

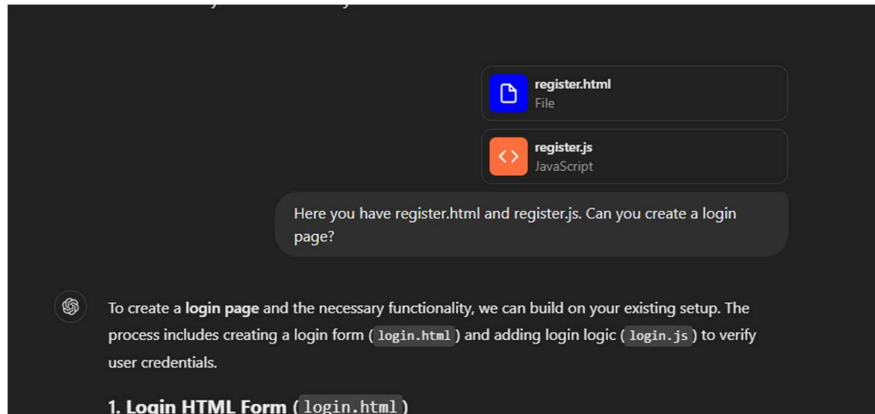
Contents

Creating the login page	2
Login page → penetration test	7
Creating the Index page	16
Index page → penetration test.....	22

Creating the login page

Prompt

Here you have register.html and register.js. Can you create a login page? (NOTE register.html and register.js files added to ChatGPT as follows)



ChatGPT

To create a login page and the necessary functionality, we can build on your existing setup. The process includes creating a login form (login.html) and adding login logic (login.js) to verify user credentials.

1. Login HTML Form (login.html)

This file will contain a form where users can input their email and password to log in.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>User Login</title>
  <link rel="stylesheet" href="/static/styles.css">
</head>
<body>
  <div class="container">
    <h1>Login</h1>
    <form action="/login" method="POST">
      <label for="username">Email:</label>
      <input type="email" id="username" name="username" placeholder="Enter your email" required>

      <label for="password">Password:</label>
      <input type="password" id="password" name="password" placeholder="Enter your password" required>

      <button type="submit">Login</button>
    </form>
  </div>
</body>
</html>
```

This form asks for the user's email and password and submits the credentials to /login via a POST request.

2. Login Logic (login.js)

Now, we'll create the backend logic to handle the login process, comparing the provided password with the hashed password stored in the database.

```
import client from "../db/db.js";
import * as bcrypt from "https://deno.land/x/bcrypt/mod.ts"; // For password comparison
import { z } from "https://deno.land/x/zod@v3.16.1/mod.ts"; // For validation

// Zod schema for login validation
const loginSchema = z.object({
  username: z.string().email({ message: "Invalid email address" }),
  password: z.string().min(8, "Password must be at least 8 characters long"),
});

// Helper function to fetch the user by email
async function getUserByEmail(email) {
  const result = await client.queryArray(
    `SELECT user_id, username, password_hash FROM abc123_users WHERE username = $1`,
    [email]
  );
  return result.rows.length > 0 ? result.rows[0] : null;
}

// Handle user login
export async function loginUser(c) {
  const body = await c.req.parseBody();
  const { username, password } = body;

  try {
    // Validate the input data using Zod
    loginSchema.parse({ username, password });

    // Fetch the user by email
    const user = await getUserByEmail(username);
    if (!user) {
      return c.text("Invalid email or password", 400);
    }

    const [userId, storedUsername, storedPasswordHash] = user;

    // Compare provided password with the stored hashed password
    const passwordMatches = await bcrypt.compare(password, storedPasswordHash);
    if (!passwordMatches) {
      return c.text("Invalid email or password", 400);
    }

    // Authentication successful, proceed to create session or token
    return c.text(`Welcome back, ${storedUsername}!`);
  } catch (error) {
    if (error instanceof z.ZodError) {
      // Handle validation errors from Zod
      return c.text(`Validation Error: ${error.errors.map(e => e.message).join(", ")}`, 400);
    }

    console.error(error);
    return c.text("Error during login", 500);
  }
}
```

Key Features of the Login System:

- **Email and Password Validation:** Zod is used to validate that the email is in a valid format and the password is at least 8 characters long.
- **Password Verification:** The login logic retrieves the stored hashed password from the database and compares it with the password provided by the user using bcrypt.
- **Response Handling:** If authentication is successful, a success message is returned; otherwise, an error is shown for invalid credentials.

Here's an example of how to create a simple user registration page for a web application using the HONO framework with JavaScript and Deno, connected to the PostgreSQL database structure we discussed earlier.

3. Hono Routes in app.js

You need to add the login route to handle the login form submission:

```
import { Hono } from "https://deno.land/x/hono/mod.ts";
import { loginUser } from "../routes/login.js"; // Import login logic
import { registerUser } from "../routes/register.js"; // Your existing register logic
import { serveStatic } from "https://deno.land/x/hono/middleware.ts";

// Create the Hono app
const app = new Hono();

// Serve static files from the /static directory
app.use('/static/*', serveStatic({ root: './static' }));

// Serve registration page
app.get('/register', async (c) => {
  return c.html(await Deno.readFile('./views/register.html'));
});

// Handle user registration
app.post('/register', registerUser);

// Serve login page
app.get('/login', async (c) => {
  return c.html(await Deno.readFile('./views/login.html')); // Use the login.html file
});

// Handle user login
app.post('/login', loginUser);

// Start the server
app.listen({ port: 3000 });
console.log('Server running on http://localhost:3000');
```

4. Testing the Login Functionality:

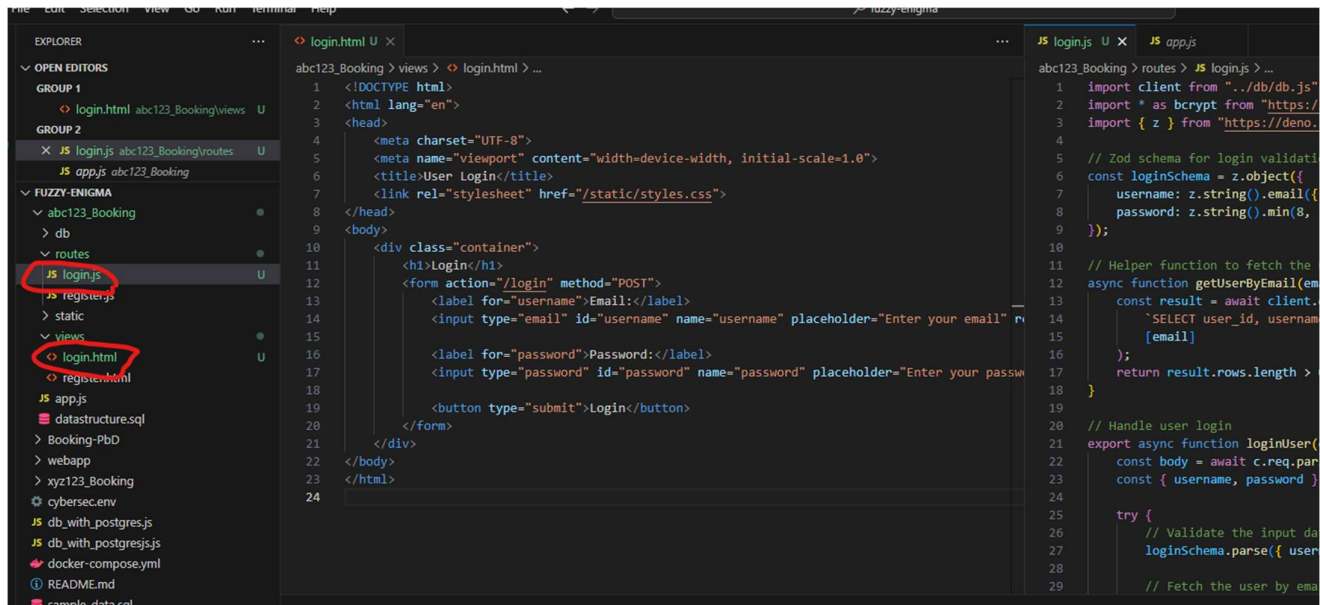
You can now test the login functionality by:

- Accessing the Login Page at <http://localhost:3000/login>.
- Submitting an email and password via the form. If the credentials match what's stored in the database, a success message is returned.

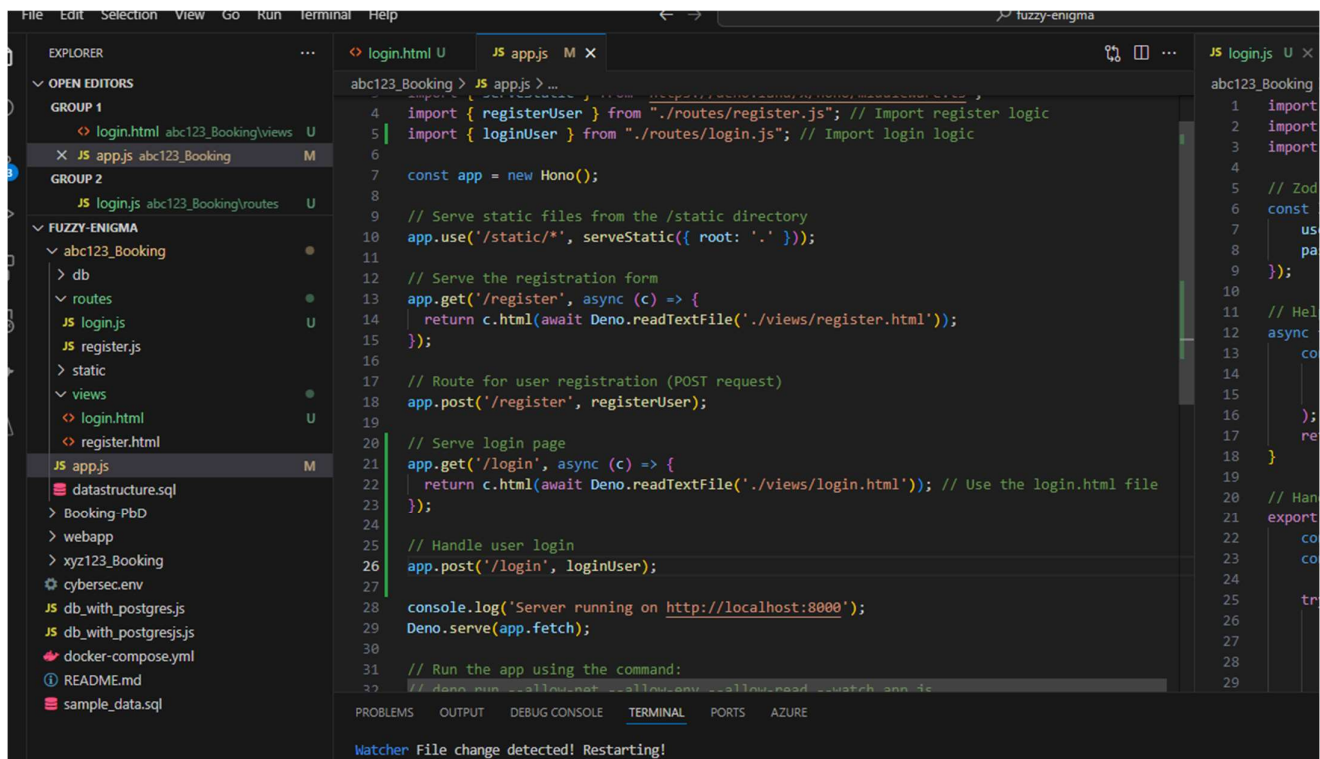
Next Steps (optional):

- **Session Management:** You can extend this by using JWT tokens or sessions to keep users logged in.
- **Redirection:** You could redirect users to a dashboard after successful login, depending on your app's needs.

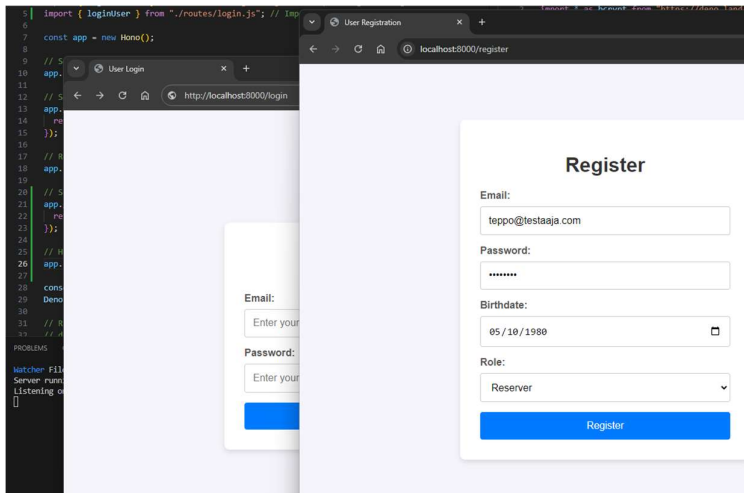
Adding files login.js and login.html



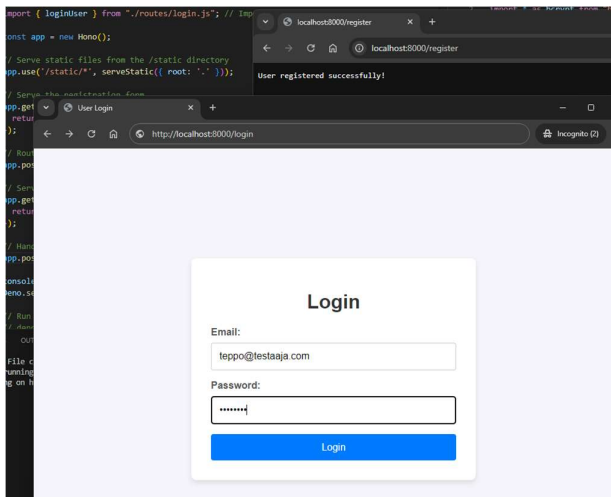
Editing the app.js file



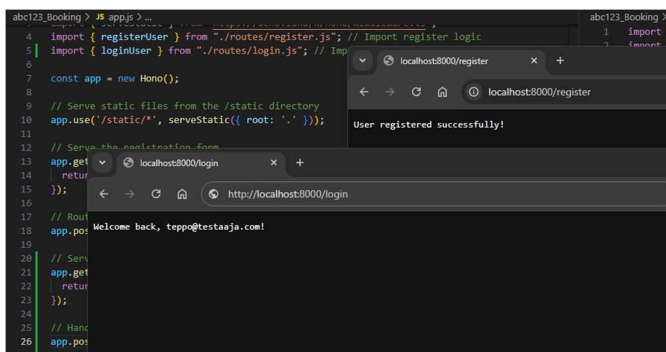
Testing the login page. Creating a known user first.



Trying to login after creation.

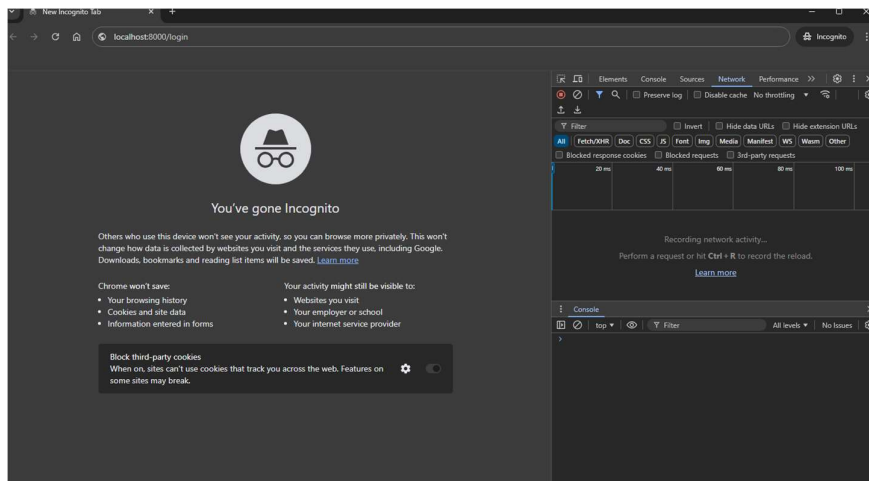


Both work

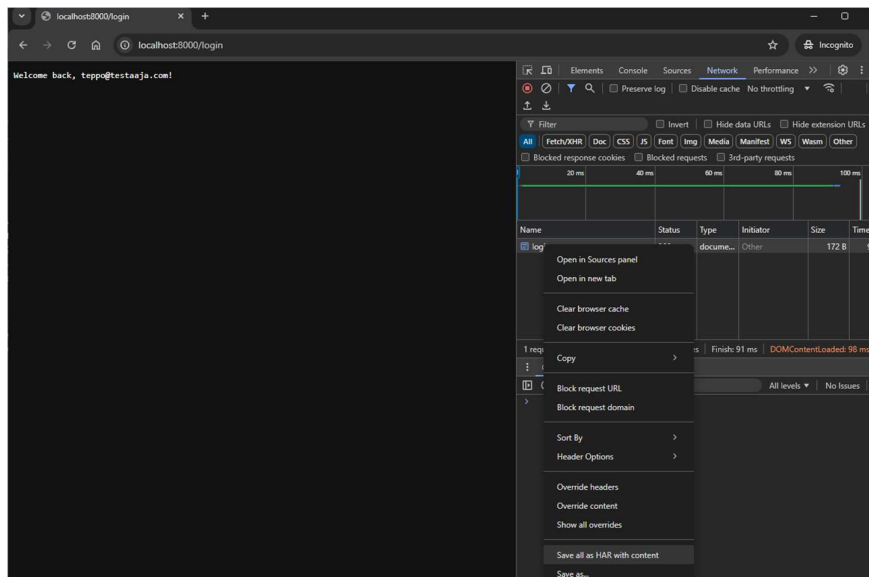


Login page → penetration test

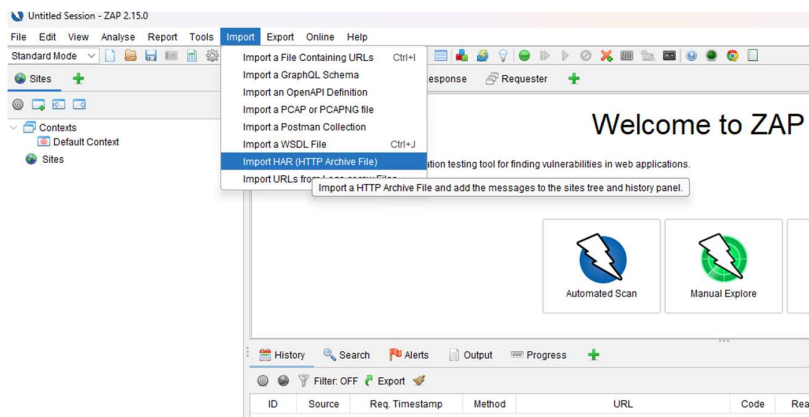
Creating a HAR file first. The first step is to open the browser with developer tools.



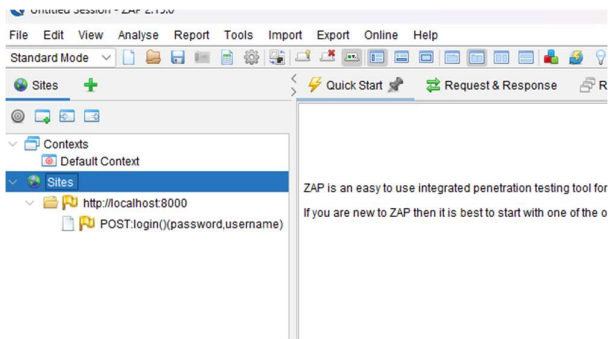
The second step is to login and save the HAR file.



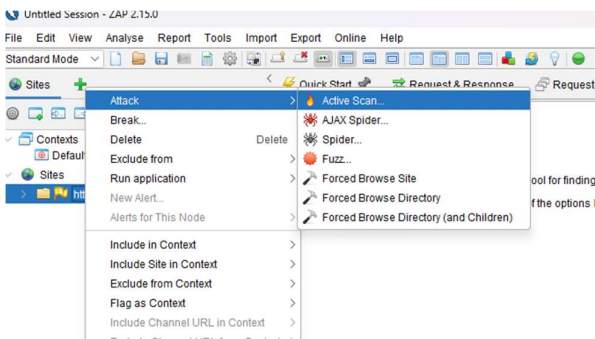
Opening ZAP and importing the HAR file.



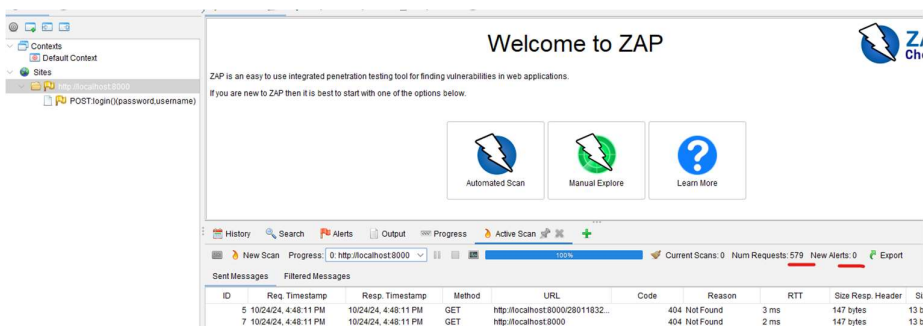
After opening, the login page should be in the Sites structure.



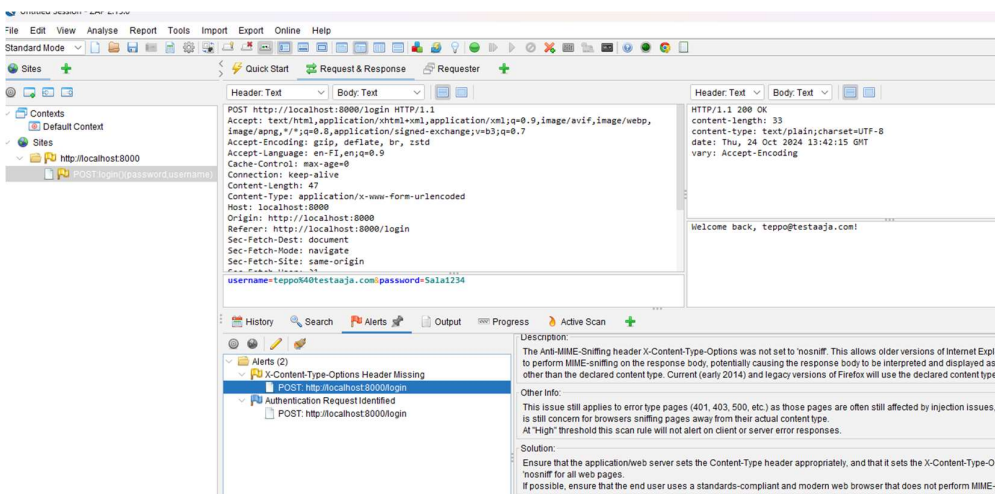
Running the first test round.



After first round no alerts.



Several rounds were performed and the situation did not change. Let's try to solve the following.



Prompt

I did a penetration test and got the following: Ensure that the application/web server sets the Content-Type header appropriately, and that it sets the X-Content-Type-Options header to 'nosniff' for all web pages.

To address the security concerns raised in the penetration test, you can enhance the Content-Type handling and add the X-Content-Type-Options header with the value 'nosniff' to prevent MIME type sniffing, which is a common attack vector.

ChatGPT

Here's how you can ensure that these headers are properly set in your Hono application:

1. Setting the Content-Type Header Appropriately:

Ensure that every response sets the appropriate Content-Type for the type of content being returned. For example, HTML pages should return Content-Type: text/html, CSS files Content-Type: text/css, and JavaScript files Content-Type: application/javascript.

2. Adding X-Content-Type-Options: nosniff Header:

This header tells the browser to not perform MIME type sniffing, which can prevent certain types of attacks where the attacker tries to trick the browser into interpreting the content differently.

Update to app.js:

You can set these headers globally for all responses by creating a middleware that applies the appropriate headers.

```
import { Hono } from "https://deno.land/x/hono/mod.ts";
import { serveStatic } from "https://deno.land/x/hono/middleware.ts";
import { loginUser } from "./routes/login.js";
import { registerUser } from "./routes/register.js";

// Create the Hono app
const app = new Hono();

// Middleware to set security headers globally
app.use('*', (c, next) => {
  // Set the Content-Type header (automatically set by Hono for HTML, CSS, JS)
  c.header('Content-Type', 'text/html'); // This will change based on your content type (text/css, application/javascript, etc.)

  // Set X-Content-Type-Options header to 'nosniff'
  c.header('X-Content-Type-Options', 'nosniff');

  return next();
});
...
```

Explanation:

Middleware:

- The middleware applies the security headers to every request. It adds:
 - Content-Type for the appropriate content being served.
 - X-Content-Type-Options: nosniff to ensure browsers don't try to guess the content type.
- Static Files: The serveStatic middleware will automatically set the appropriate Content-Type headers based on the file extension (e.g., .css, .js, .html).

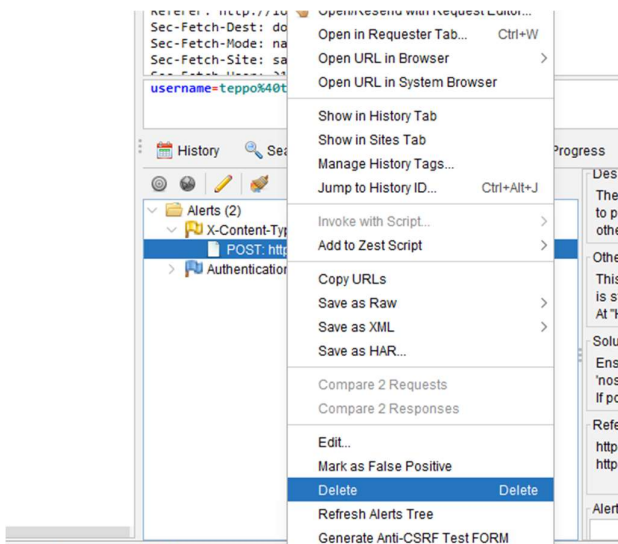
Adding the code

```

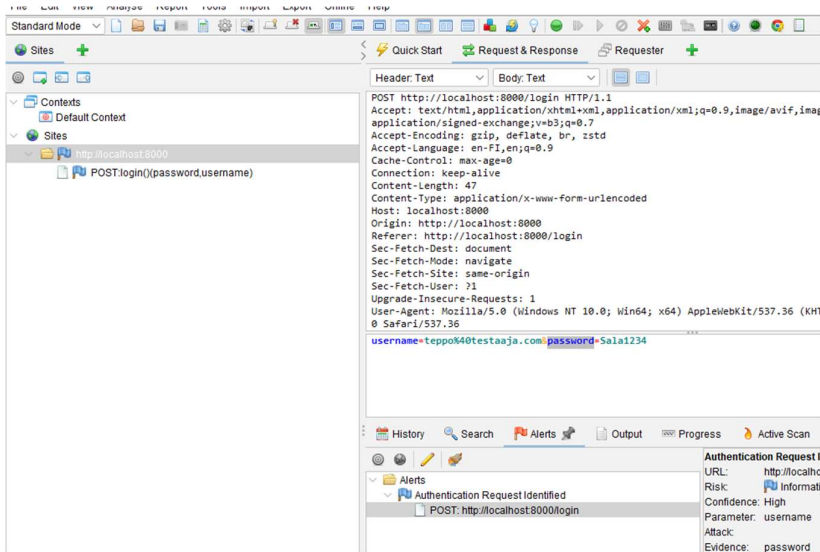
1 // app.js
2 import { Hono } from "https://deno.land/x/hono/mod.ts";
3 import { serveStatic } from "https://deno.land/x/hono/middleware.ts";
4 import { registerUser } from "../routes/register.js"; // Import register logic
5 import { loginUser } from "../routes/login.js"; // Import login logic
6
7 const app = new Hono();
8
9 // Middleware to set security headers globally
10 app.use("*", (c, next) => {
11   // Set the Content-Type header (automatically set by Hono for HTML, CSS, JS)
12   c.header('Content-Type', 'text/html'); // This will change based on your content type (text/html, application/javascript, etc.)
13
14   // Set X-Content-Type-Options header to 'nosniff'
15   c.header('X-Content-Type-Options', 'nosniff');
16
17   return next();
18 });
19
20 // Serve static files from the /static directory
21 app.use('/static/*', serveStatic({ root: '.' }));
22
23 // Serve the registration form
24 app.get('/register', async (c) => {

```

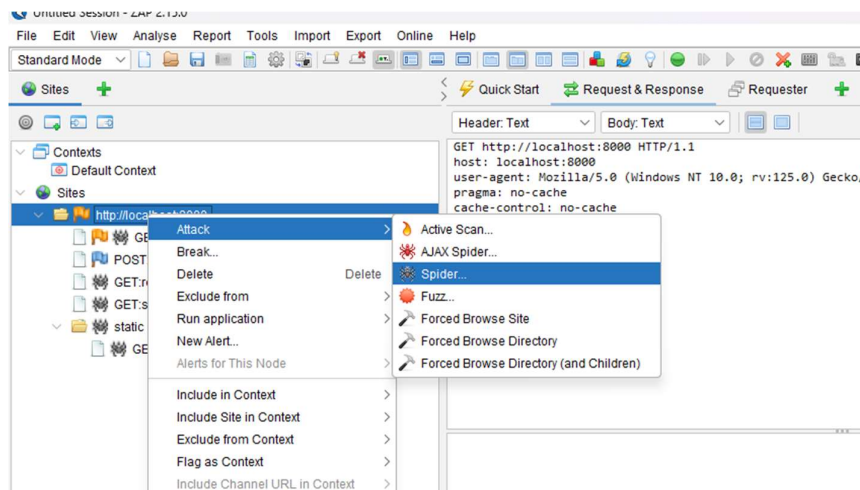
Deleting the alert and retesting.



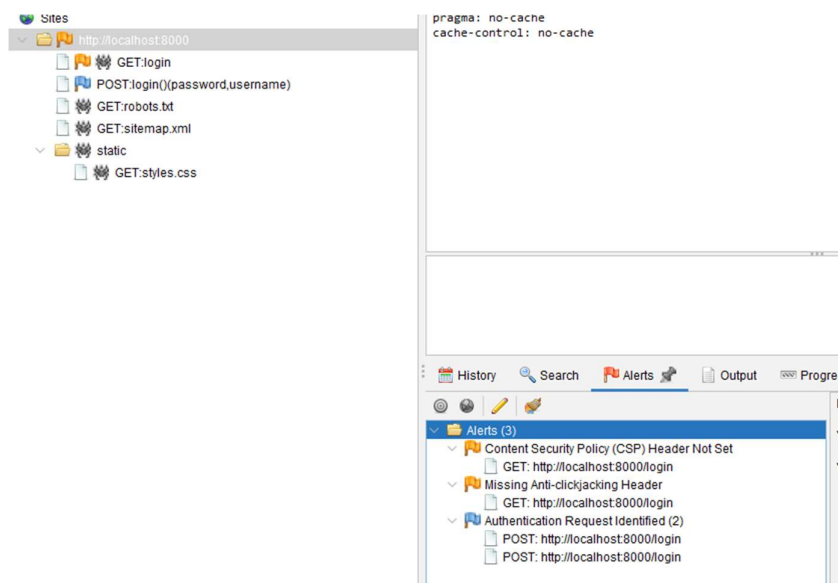
The alert did not appear



Running a Spider attack



New alerts → Maybe a never ending story



Prompt

And next alert: Ensure that your web server, application server, load balancer, etc. is configured to set the Content-Security-Policy header.

ChatGPT

To address the security alert regarding the Content-Security-Policy (CSP) header, you need to configure your web application to include this header. CSP helps protect against certain types of attacks like cross-site scripting (XSS) by specifying which sources of content are allowed to be loaded on your site.

What is a Content-Security-Policy (CSP)?

A Content-Security-Policy allows you to specify:

- The sources from which your site can load resources such as scripts, styles, images, etc.
- Rules that prevent unsafe inline scripts or styles.
- Restrictions on embedding your site in iframes or loading it in a new window.

Setting a Basic CSP Header

Here's a simple example of a CSP that:

- Only allows resources (JavaScript, CSS, images, etc.) to be loaded from your own domain (self).
- Prevents inline scripts or styles (unsafe-inline).
- Blocks content from unknown sources.

Update to app.js:

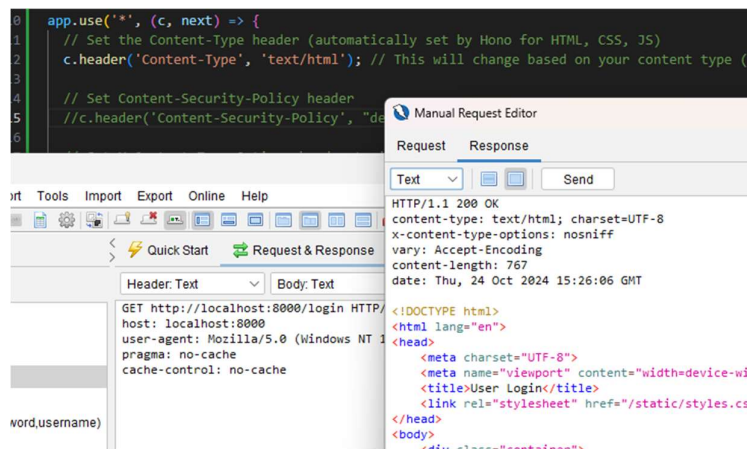
// Middleware to set security headers globally, including CSP

```
app.use('*', (c, next) => {
```

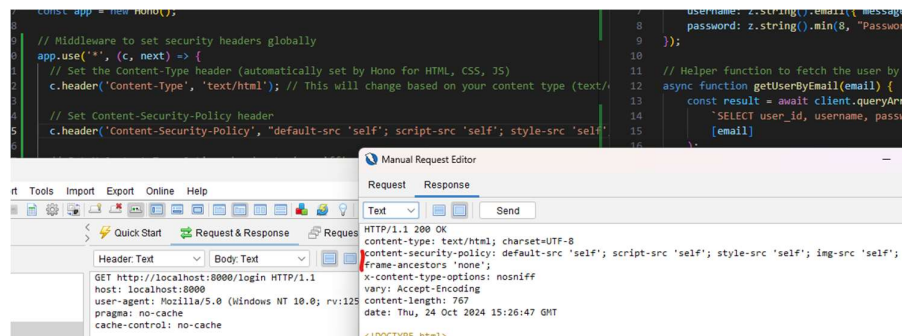
```
  // Set Content-Security-Policy header
```

```
  c.header('Content-Security-Policy', "default-src 'self'; script-src 'self'; style-src 'self'; img-src 'self'; frame-ancestors 'none';");
```

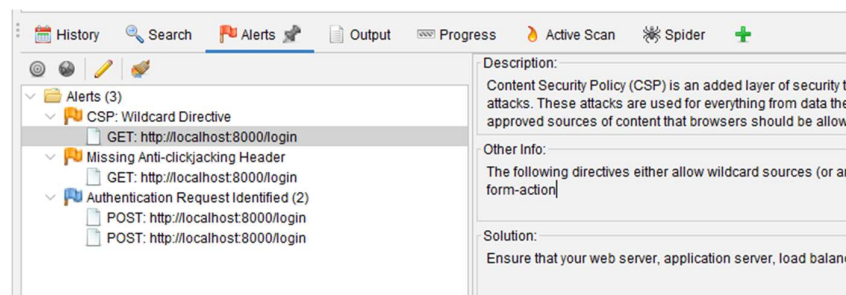
First check without



Then after



And next



Prompt

And next with CSP: The following directives either allow wildcard sources (or ancestors), are not defined, or are overly broadly defined:

form-action

ChatGPT

To address the alert regarding the form-action directive in the Content-Security-Policy (CSP), you need to explicitly define which URLs are allowed to handle form submissions. The form-action directive restricts where a form can be submitted, helping prevent Cross-Site Request Forgery (CSRF) attacks by ensuring forms can only be submitted to trusted domains.

Fixing the CSP for form-action

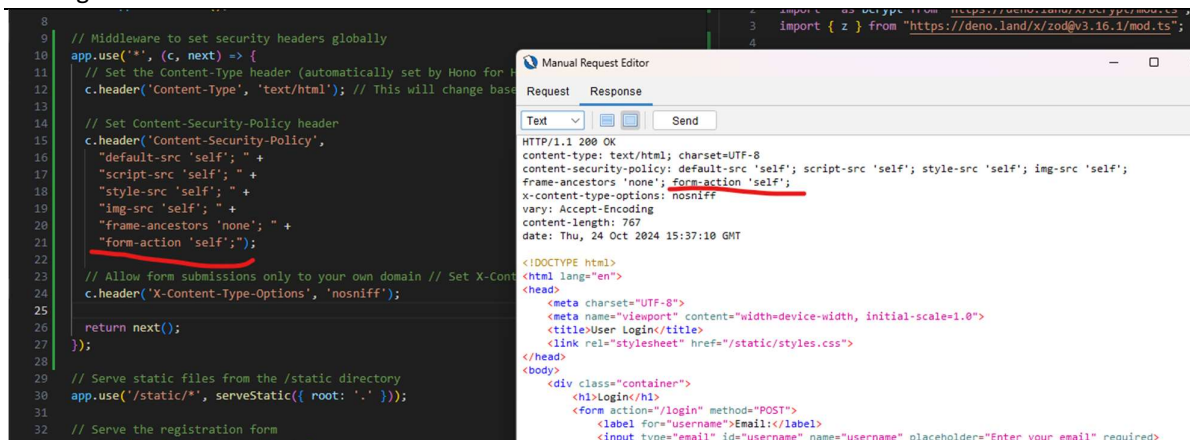
By default, the browser will allow forms to be submitted to any location if the form-action directive is not set or too broadly defined. To enhance security, you should specify that forms can only be submitted to your own domain or trusted domains.

Here's an updated version of your CSP header that includes the form-action directive:

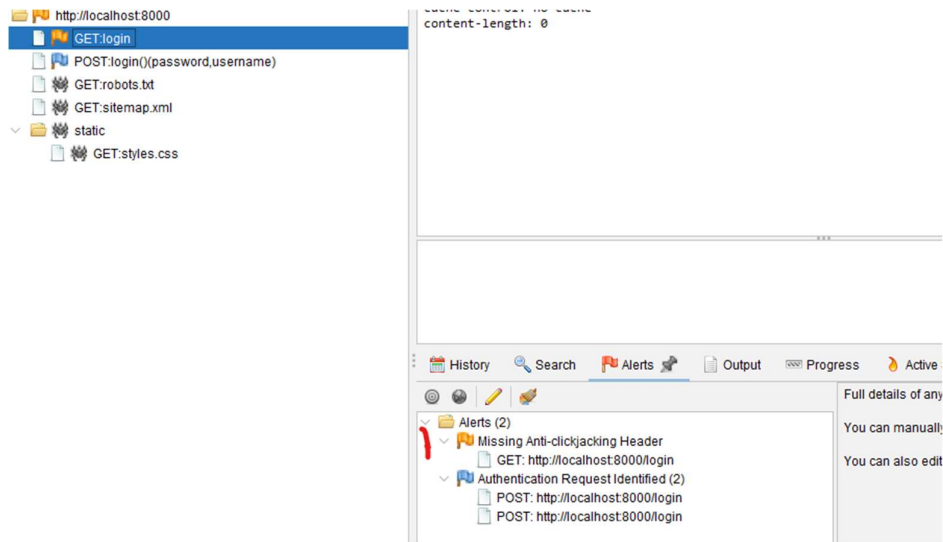
Update to app.js:

```
// Set Content-Security-Policy header
c.header('Content-Security-Policy',
  "default-src 'self'; " +
  "script-src 'self'; " +
  "style-src 'self'; " +
  "img-src 'self'; " +
  "frame-ancestors 'none'; " +
  "form-action 'self';"); // Allow form submissions only to your own domain
```

Testing



One more



Prompt

And next one: The response does not protect against 'ClickJacking' attacks. It should include either Content-Security-Policy with 'frame-ancestors' directive or X-Frame-Options.

ChatGPT

To protect your application from Clickjacking attacks, you should configure either the X-Frame-Options header or use the Content-Security-Policy (CSP) with the frame-ancestors directive.

What is Clickjacking?

Clickjacking is an attack where a malicious site embeds your application in an invisible frame or iframe and tricks users into interacting with it. By preventing your site from being embedded in iframes, you can mitigate this risk.

You can implement both protections for better compatibility across browsers:

- X-Frame-Options: This older header is supported by most browsers and is commonly used to prevent framing.
- CSP frame-ancestors: This directive in CSP is a more modern and flexible way to control which domains are allowed to embed your site.

1. Adding X-Frame-Options Header

The X-Frame-Options header has three options:

- DENY: Prevents any site from embedding your site.
- SAMEORIGIN: Only allows your own site (same origin) to embed pages.
- ALLOW-FROM: Allows a specific trusted domain to embed your pages (this is less commonly supported and deprecated in some browsers).

2. Using CSP with frame-ancestors

The frame-ancestors directive in the Content-Security-Policy header allows you to specify which domains are permitted to embed your site in an iframe.

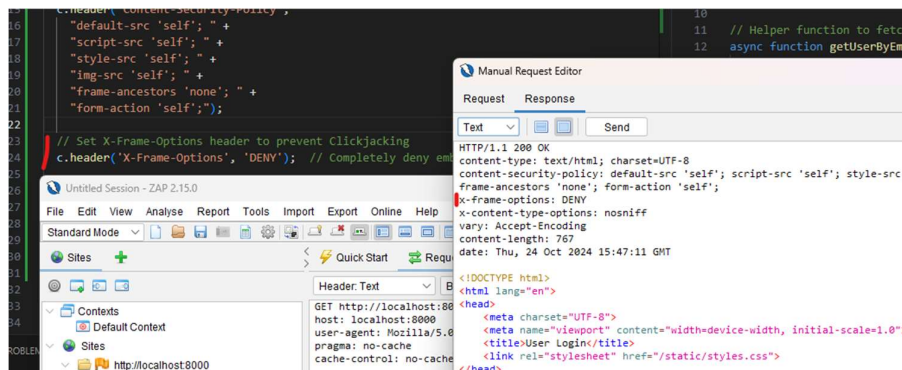
Update to app.js (Combining Both):

You can add both X-Frame-Options and CSP with frame-ancestors to your Hono application to prevent clickjacking attacks:

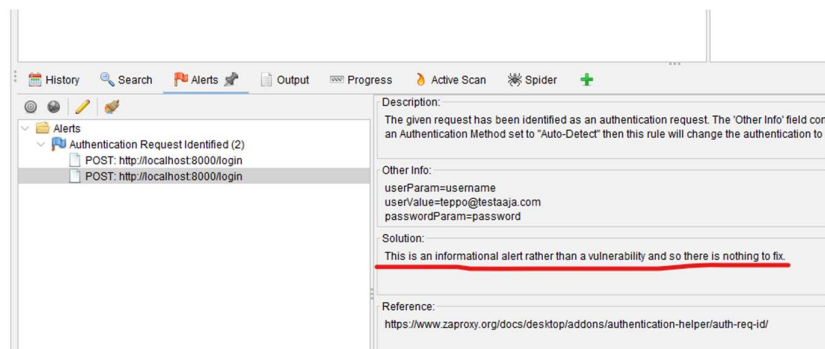
```
// Set Content-Security-Policy header to prevent embedding (Clickjacking protection)
c.header('Content-Security-Policy',
  "default-src 'self'; " +
  "script-src 'self'; " +
  "style-src 'self'; " +
  "img-src 'self'; " +
  "frame-ancestors 'none';" // Prevent all embedding BUT WHERE IS "form-action 'self';"
);
```

```
// Set X-Frame-Options header to prevent Clickjacking
c.header('X-Frame-Options', 'DENY'); // Completely deny embedding
```

Only X-Frame-Options are added



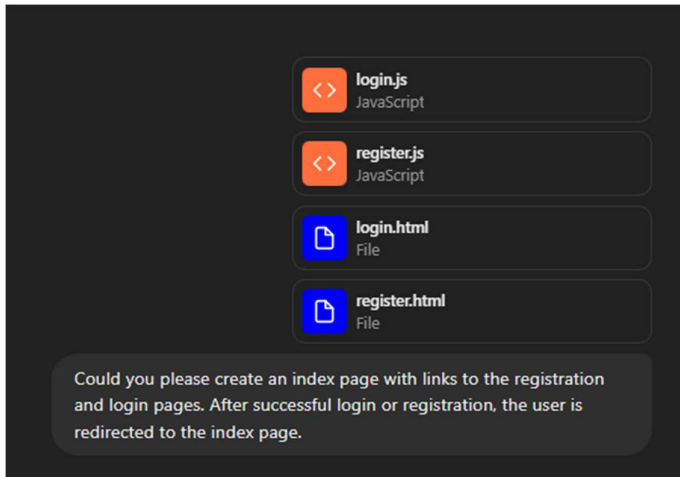
And the last alerts left → Not very severe at this point



Creating the Index page

Prompt

Could you please create an index page with links to the registration and login pages. After successful login or registration, the user is redirected to the index page.



ChatGPT

To create an index page that contains links to the registration and login pages, and to redirect users back to the index page after a successful login or registration, you can follow these steps.

1. Index Page (index.html):

Create a simple index page with links to the login and registration pages.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Home</title>
  <link rel="stylesheet" href="/static/styles.css">
</head>
<body>
  <div class="container">
    <h1>Welcome to the Booking System</h1>
    <p>Please choose one of the options below:</p>
    <ul>
      <li><a href="/login">Login</a></li>
      <li><a href="/register">Register</a></li>
    </ul>
  </div>
</body>
</html>
```

This index page simply provides two links: one to the login page and one to the registration page.

2. Modify Login and Registration Handlers (login.js, register.js):

Update the login and registration logic to redirect users back to the index page after a successful login or registration.

Updated login.js:

```
import client from "../db/db.js";
import * as bcrypt from "https://deno.land/x/bcrypt/mod.ts"; // For password comparison
import { z } from "https://deno.land/x/zod@v3.16.1/mod.ts"; // For validation

// Zod schema for login validation
const loginSchema = z.object({
  username: z.string().email({ message: "Invalid email address or password" }),
  password: z.string().min(8, "Invalid email address or password"),
});

// Helper function to fetch the user by email
async function getUserByEmail(email) {
  const result = await client.queryArray(
    `SELECT user_id, username, password_hash FROM abc123_users WHERE username = $1`,
    [email]
  );
  return result.rows.length > 0 ? result.rows[0] : null;
}

// Handle user login
export async function loginUser(c) {
  const body = await c.req.parseBody();
  const { username, password } = body;

  try {
    // Validate the input data using Zod
    loginSchema.parse({ username, password });

    // Fetch the user by email
    const user = await getUserByEmail(username);
    if (!user) {
      return c.text("Invalid email or password", 400);
    }

    const [userId, storedUsername, storedPasswordHash] = user;

    // Compare provided password with the stored hashed password
    const passwordMatches = await bcrypt.compare(password, storedPasswordHash);
    if (!passwordMatches) {
      return c.text("Invalid email or password", 400);
    }

    // Authentication successful, redirect to the index page
    return c.redirect('/');
  } catch (error) {
    if (error instanceof z.ZodError) {
      // Handle validation errors from Zod
      return c.text(`Validation Error: ${error.errors.map(e => e.message).join(", ")}`, 400);
    }
    console.error(error);
    return c.text("Error during login", 500);
  }
}
```

Updated register.js:

```

import client from "../db/db.js";
import * as bcrypt from "https://deno.land/x/bcrypt/mod.ts";
import { z } from "https://deno.land/x/zod@v3.16.1/mod.ts"; // Import Zod

// Zod schema for validating the registration form
const registerSchema = z.object({
  username: z.string().email({ message: "Invalid email address" }).max(50, "Email must not exceed 50 characters"),
  password: z.string().min(8, "Password must be at least 8 characters long"),
  birthdate: z.string().refine((date) => {
    // Ensure birthdate is a valid date (and optionally check for age)
    const birthDateObj = new Date(date);
    return !isNaN(birthDateObj.getTime()); // Check if it's a valid date
  }, { message: "Invalid birthdate" }),
  role: z.enum(["reserver", "admin"], { message: "Invalid role" }),
});

// Helper function to check if a username (email) already exists
async function isUniqueUsername(email) {
  const result = await client.queryArray(`SELECT username FROM abc123_users WHERE username = $1`, [email]);
  return result.rows.length === 0;
}

// Handle user registration
export async function registerUser(c) {
  const body = await c.req.parseBody();
  const { username, password, birthdate, role } = body;

  try {
    // Validate the input data using Zod
    registerSchema.parse({ username, password, birthdate, role });

    // Check if the email is unique
    if (!(await isUniqueUsername(username))) {
      return c.text('Email already in use', 400);
    }

    // Hash the user's password
    const salt = await bcrypt.genSalt(10);
    const hashedPassword = await bcrypt.hash(password, salt);

    // Insert the new user into the database
    await client.queryArray(
      `INSERT INTO abc123_users (username, password_hash, role, birthdate) VALUES ($1, $2, $3, $4)`
      [username, hashedPassword, role, birthdate]
    );

    // Success response, redirect to the index page
    return c.redirect('/');

  } catch (error) {
    if (error instanceof z.ZodError) {
      // Handle validation errors from Zod
      return c.text(`Validation Error: ${error.errors.map(e => e.message).join(", ")}`, 400);
    }

    console.error(error);
    return c.text('Error during registration', 500);
  }
}

```

3. Update Routes in app.js:

Ensure that the new index page and redirects are properly handled by the server.

```
import { Hono } from "https://deno.land/x/hono/mod.ts";
import { serveStatic } from "https://deno.land/x/hono/middleware.ts";
import { loginUser } from "../routes/login.js";
import { registerUser } from "../routes/register.js";

// Create the Hono app
const app = new Hono();

// Serve static files from the /static directory
app.use('/static/*', serveStatic({ root: './static' }));

// Serve the index page
app.get('/', async (c) => {
  return c.html(await Deno.readFile('./views/index.html'));
});

// Serve the login page
app.get('/login', async (c) => {
  return c.html(await Deno.readFile('./views/login.html'));
});

// Handle user login
app.post('/login', loginUser);

// Serve the registration page
app.get('/register', async (c) => {
  return c.html(await Deno.readFile('./views/register.html'));
});

// Handle user registration
app.post('/register', registerUser);

// Start the server
app.listen({ port: 3000 });
console.log('Server running on http://localhost:3000');
```

Summary:

- Index Page (index.html): Provides links to login and registration pages.
- Login and Registration Logic: Redirects users to the index page after a successful login or registration.
- Server Setup (app.js): Routes for the index, login, and registration pages are defined, and appropriate handlers are attached.

This setup will now allow users to be redirected to the index page after they successfully log in or register. Let me know if you need further customizations!

Adding the index.html file and editing the app.js file

```

// app.js
10 app.use('*', (c, next) => {
26 // Allow form submissions only to your own domain // Set X-Content-Type-Options header to 'nosniff'
27 c.header('X-Content-Type-Options', 'nosniff');
28
29 return next();
30 });
31
32 // Serve static files from the /static directory
33 app.use('/static/*', serveStatic({ root: '.' }));
34
35 // Serve the index page
36 app.get('/', async (c) => {
37   return c.html(await Deno.readFile('./views/index.html'));
38 });
39
40 // Serve the registration form
41 app.get('/register', async (c) => {
42   return c.html(await Deno.readFile('./views/register.html'));
43 });
44
45 // Route for user registration (POST request)
46 app.post('/register', registerUser);
47
48 // Serve login page
49 app.get('/login', async (c) => {
50   return c.html(await Deno.readFile('./views/login.html')); // Use the login.html file
51 });
52
53 // Handle user login

```

```

// index.html
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1">
6   <title>Home</title>
7   <link rel="stylesheet" href="/static/css/index.css">
8 </head>
9 <body>
10   <div class="container">
11     <h1>Welcome to the Reservation System</h1>
12     <p>Please choose one of the options below:</p>
13     <ul>
14       <li><a href="/login">Login</a>
15       <li><a href="/register">Register</a>
16     </ul>
17   </div>
18 </body>
19 </html>
20

```

Editing the register.js file and editing the login.js file

```

// register.js
24 export async function registerUser(c) {
30
31   // Hash the user's password
32   const salt = await bcrypt.genSalt(10);
33   const hashedPassword = await bcrypt.hash(password, salt);
34
35   // Insert the new user into the database
36   await client.queryArray(
37     `INSERT INTO abc123_users (username, password_hash, role, birthdate) VALUES ($1, $2, $3, $4)`,
38     [username, hashedPassword, role, birthdate]
39   );
40
41   // Success response, redirect to the index page
42   return c.redirect('/');
43
44   catch (error) {
45     if (error instanceof z.ZodError) {
46       // Handle validation errors from Zod
47       return c.text('Validation Error: ${error.errors.map(e => e.message).join(", ")}', 400);
48     }
49     console.error(error);
50     return c.text('Error during registration', 500);
51   }
52 }

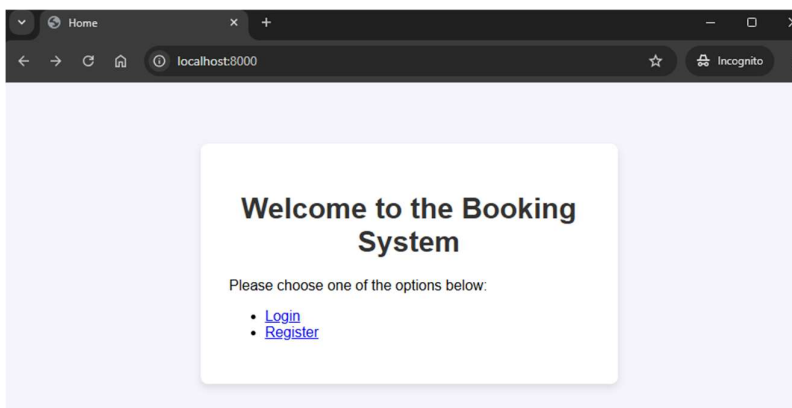
```

```

// login.js
21 export async function loginUser(c) {
22   if (!user) {
23     return c.text('Invalid email or password', 400);
24   }
25
26   const {userId, storedUsername, storedPasswordHash} = user;
27
28   // Compare provided password with the stored hashed password
29   const passwordMatches = await bcrypt.compare(password, storedPasswordHash);
30   if (!passwordMatches) {
31     return c.text('Invalid email or password', 400);
32   }
33
34   // Authentication successful, redirect to the index page
35   return c.redirect('/');
36
37   catch (error) {
38     if (error instanceof z.ZodError) {
39       // Handle validation errors from Zod
40       return c.text('Validation Error: ${error.errors.map(e => e.message).join(", ")}', 400);
41     }
42     console.error(error);
43   }
44 }

```

Testing the index page and redirects → Trying to register and login with the created account.



Registration works

User Registration

localhost:8000/register

Register

Email:

Password:

Birthdate:

Role:
 ▼

Login works

User Login

localhost:8000/login

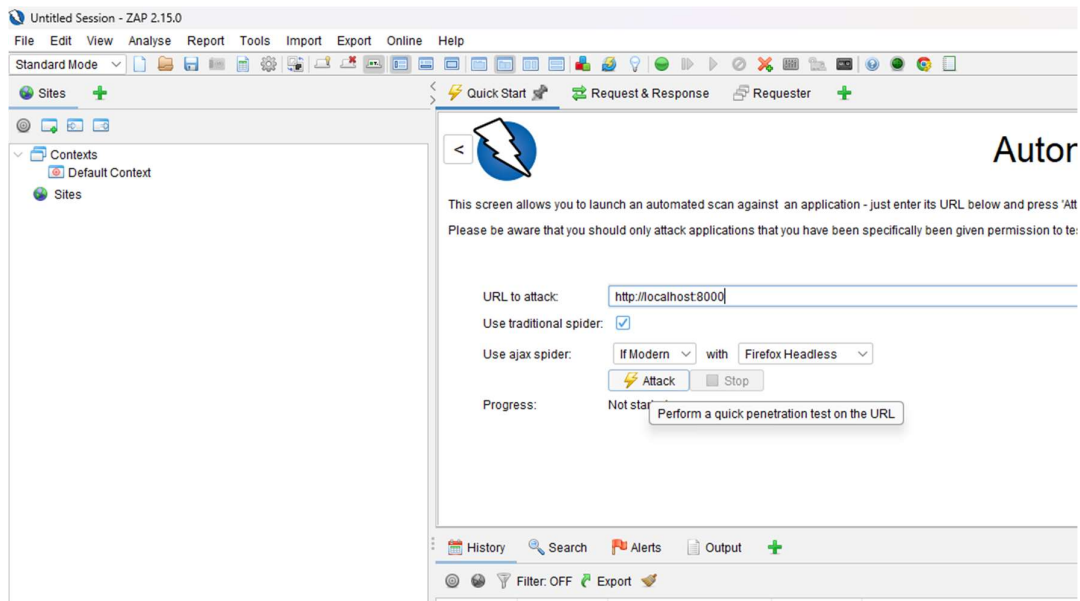
Login

Email:

Password:

Index page → penetration test

First round



And results → After the second round no changes

