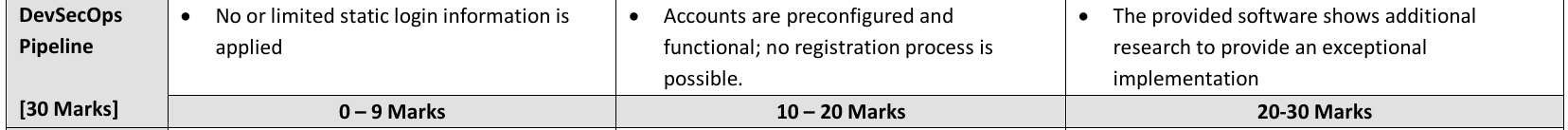
## Note:

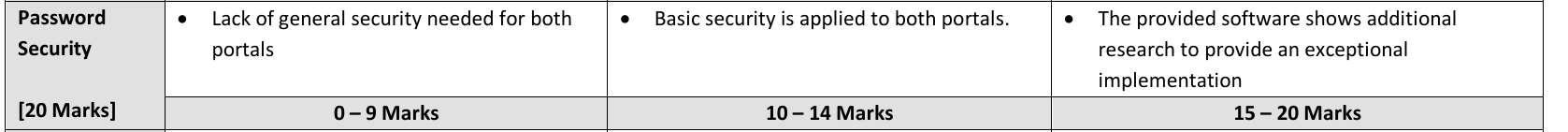
## The customer portal was revamped as well to meet part 3 standards.

## All code will be included in submission

# Project code meeting rubric requirements.



# DevSecOps Pipeline



# Password Security

### Customer Portal

1. Password Strength Validation:

* Implemented in the validateInput middleware.
* Enforces strong password rules: at least 8 characters, one uppercase letter, one lowercase letter, one number.

1. Secure Storage:

* Passwords are hashed using bcrypt with a salt before being stored in the database.

1. Brute Force Protection:

* Implemented express-brute middleware to limit login attempts and prevent brute force attacks.

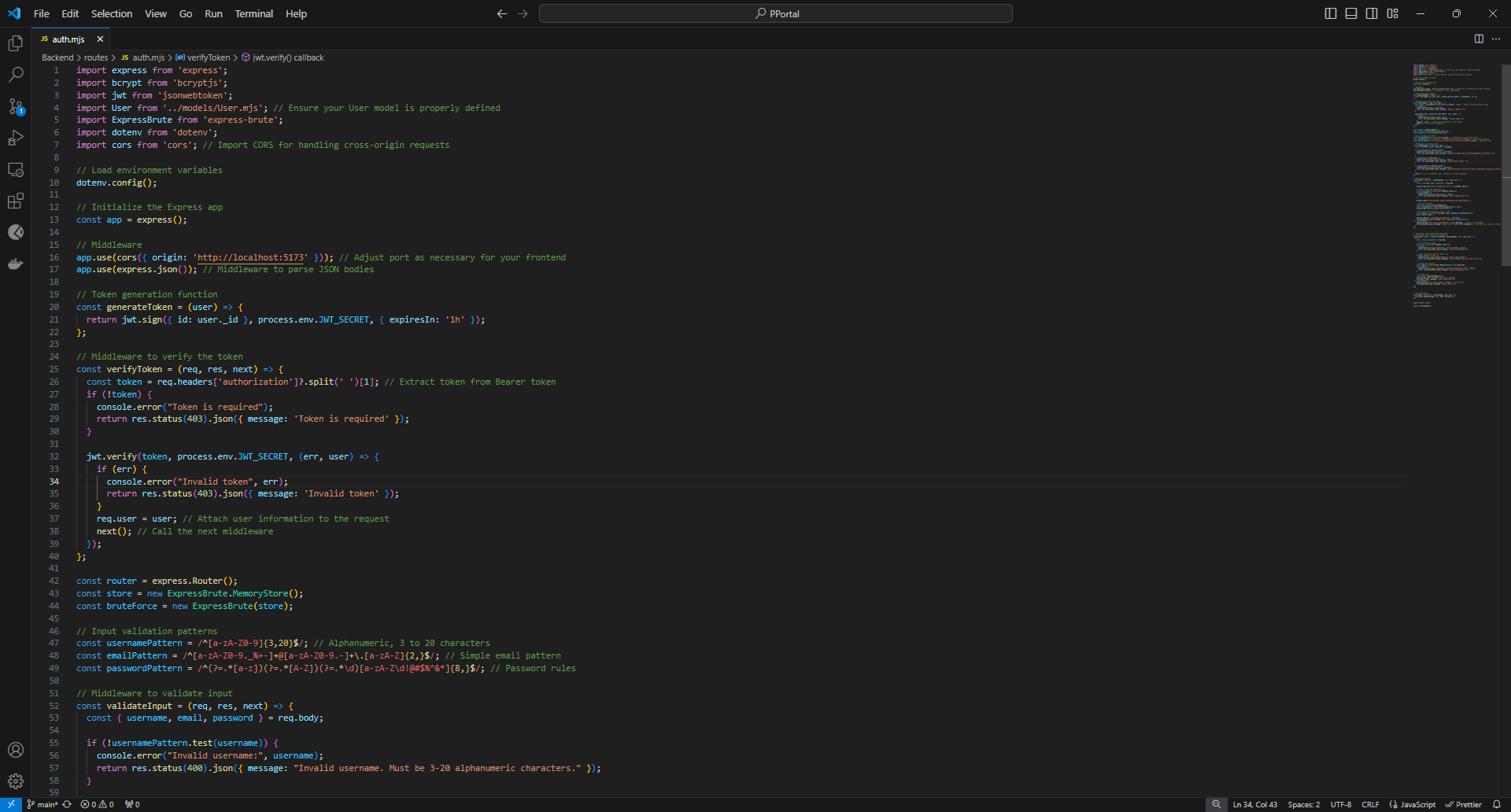
1. Secure Authentication:

* Passwords are securely verified during login using bcrypt.
* JWT tokens are signed with a secret and expire in 1 hour to reduce risk.

1. Error Handling:

* Error messages are generic to prevent user enumeration attacks.

Code Example:

The main implementation file auth.mjs:  
A screen shot of a computer

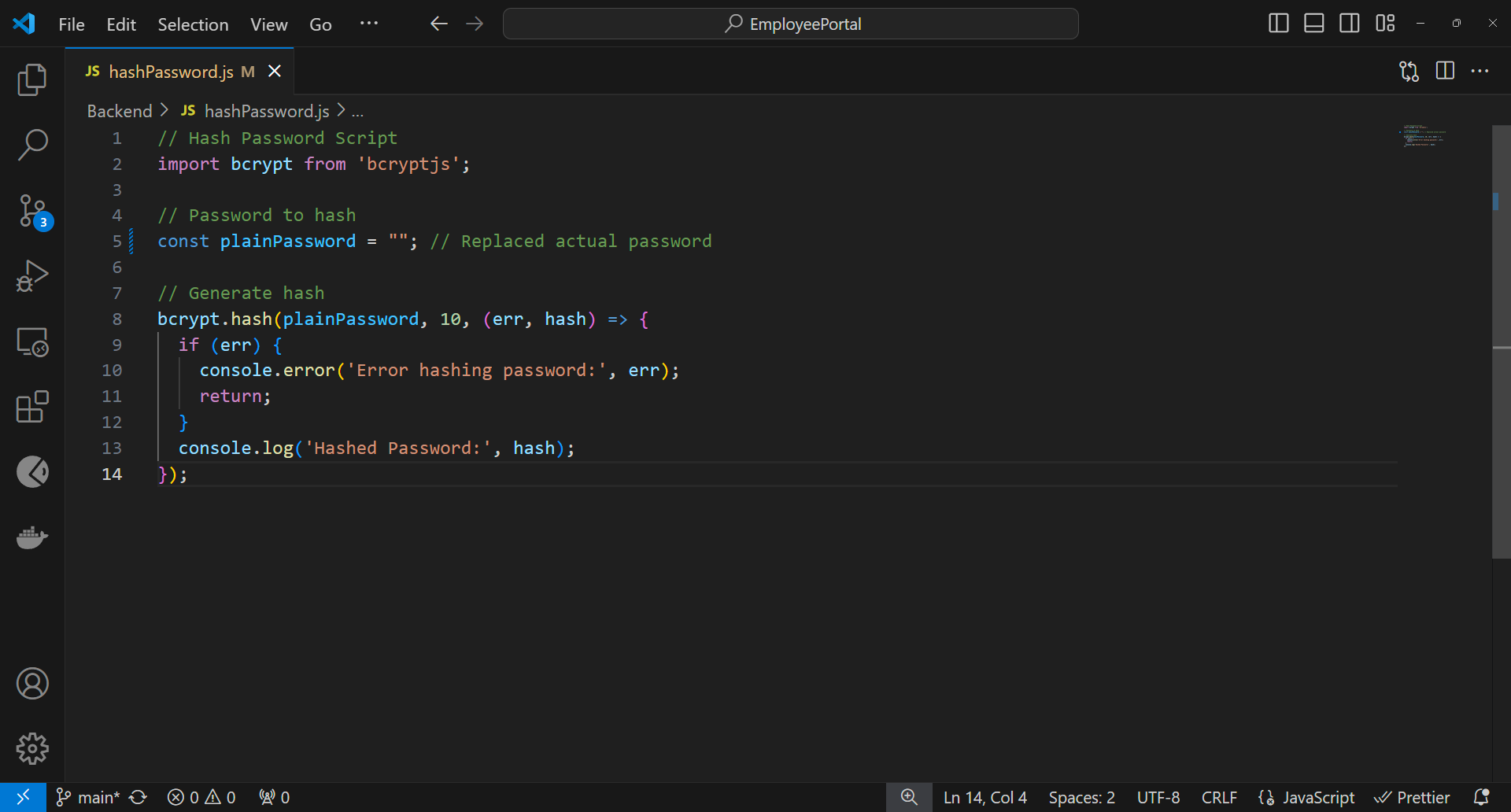
Description automatically generated

### Employee Portal

1. **Password Strength Validation:**

* The Employee Portal does not have a registration feature. Employee accounts were pre-configured and added directly to the database.
* Passwords were hashed using the hashPassword.js script, which utilizes bcrypt with a salt of 10 to ensure secure password storage.

Evidence: The following script was used for hashing passwords:  
Backend\hashPassword.js:



1. Secure Storage:

* Passwords are stored in hashed format using bcrypt with salting to enhance security.

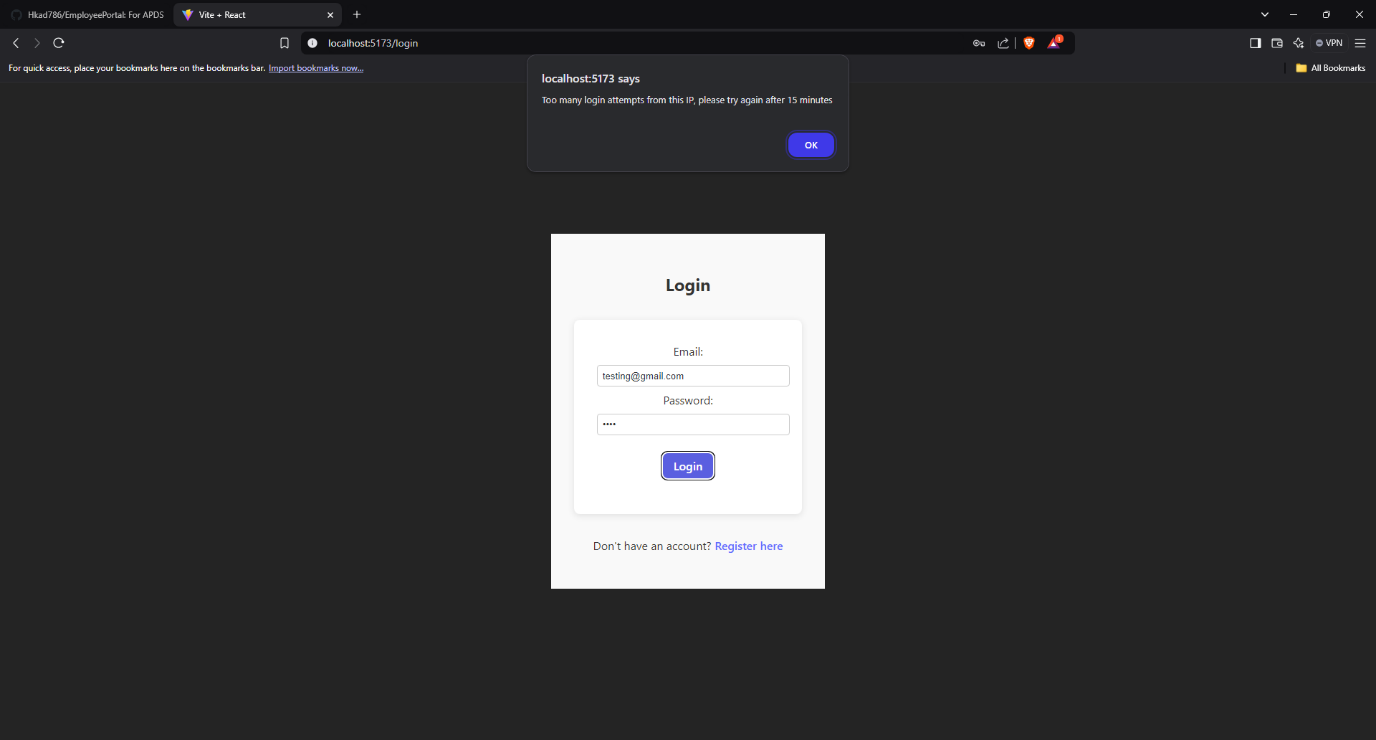
1. Brute Force Protection:

* A rate limiter restricts login attempts to 5 per 15 minutes to prevent brute force attacks.
* Evidence: auth.mjs implements this via the loginLimiter middleware.

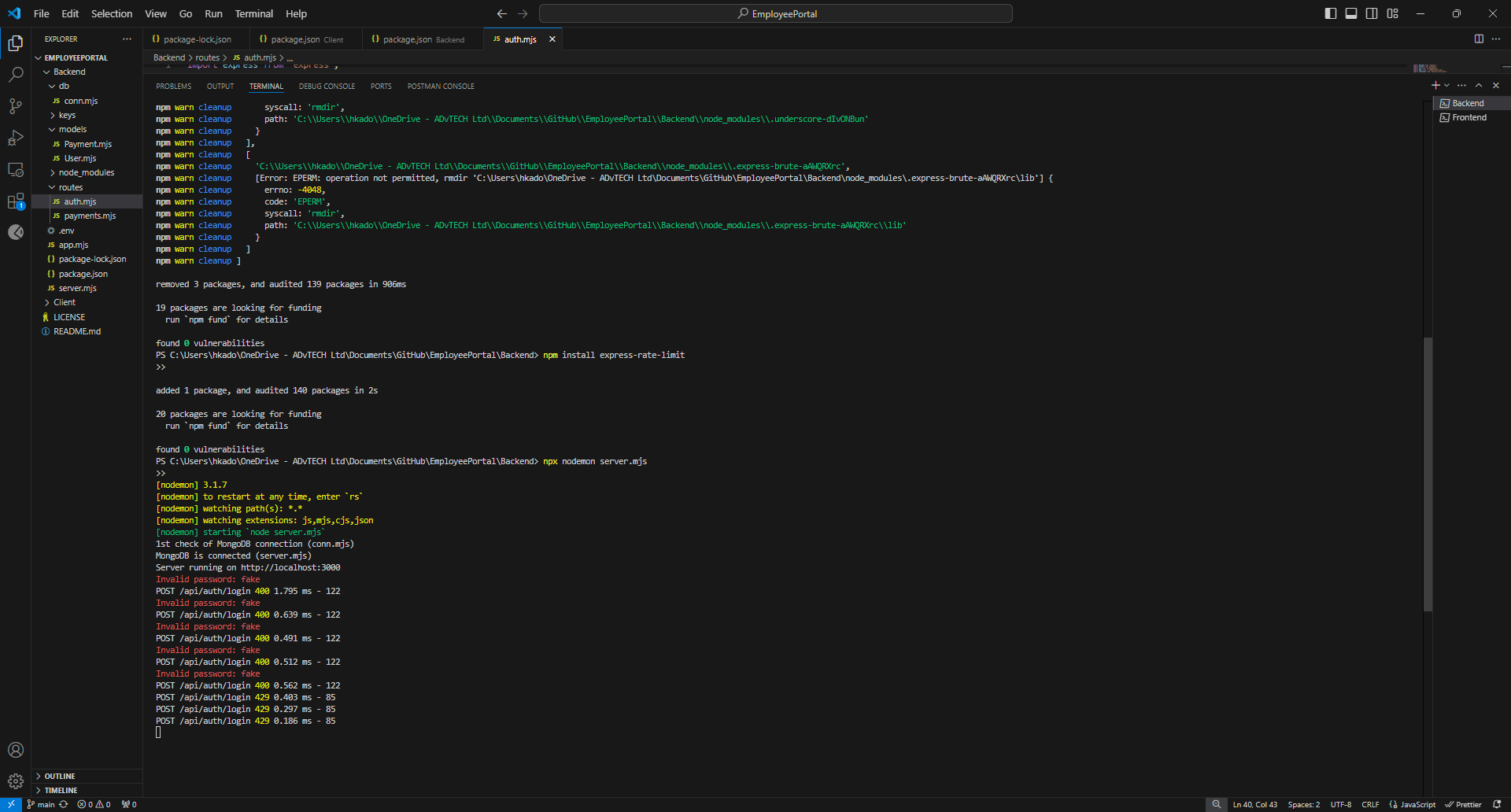
**Transition from express-brute to express-rate-limit:**

* Initial Approach: The portal initially used express-brute for brute force protection.
* Limitations Identified:
* After receiving GitHub Dependabot warnings about outdated dependencies and potential security vulnerabilities in express-brute, I reviewed its maintenance status and security risks.
* Research revealed that express-brute has limited active development and lacks modern security features (e.g., robust customization for attack patterns).
* Research and Decision:
* Explored community discussions on GitHub, Stack Overflow, and security blogs, which recommended express-rate-limit as an actively maintained and widely used alternative.
* Chose express-rate-limit because it offers:
* Simple configuration and robust performance.
* Active development and regular updates.
* More flexibility to adapt to modern attack patterns (e.g., IP-based or route-specific limits).
* Implementation:

Replaced express-brute with express-rate-limit in the authentication route to address these concerns.

Pics Showing implementation:  
(Using an incorrect account multiple times locks it out)

(backend View):



1. Secure Authentication:

* Passwords are securely verified using bcrypt.compare during login.
* JWT tokens are signed with a secret key and expire in 1 hour, reducing exposure risk in case of token theft.

1. Error Handling:

* Generic error messages prevent user enumeration attacks.
* Example: "Invalid credentials" is displayed regardless of whether the email or password is incorrect.

1. Additional Measures:

* Session Hijacking Protection: HTTPS will be used during deployment to encrypt all communication between client and server.
* Clickjacking Protection: Implemented via helmet middleware, which sets the X-Frame-Options header.
* Cross-Site Scripting (XSS) Prevention: Input validation sanitizes user inputs to prevent script injection.
* Man-in-the-Middle Attack Prevention: HTTPS ensures secure communication during data transmission.

Main Code: A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

## Reference links:

OWASP Password Storage Cheat Sheet:

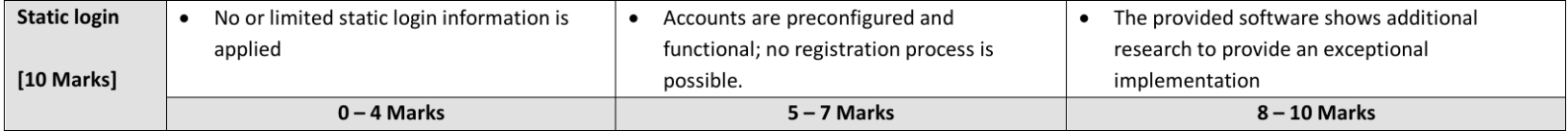
<https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html>

Express Rate Limit GitHub:

<https://github.com/nfriedly/express-rate-limit>

bcrypt GitHub Repository:

<https://github.com/kelektiv/node.bcrypt.js/>



# Static Login

## Employee Accounts

1. Preconfigured Accounts:

* Employee accounts are preconfigured directly in MongoDB with no registration feature, ensuring static login functionality.
* Passwords are securely hashed using the hashPassword.js script, leveraging bcrypt with salting for strong password storage.

Pics showing the setup: A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

1. Role-Based Access Control (RBAC):

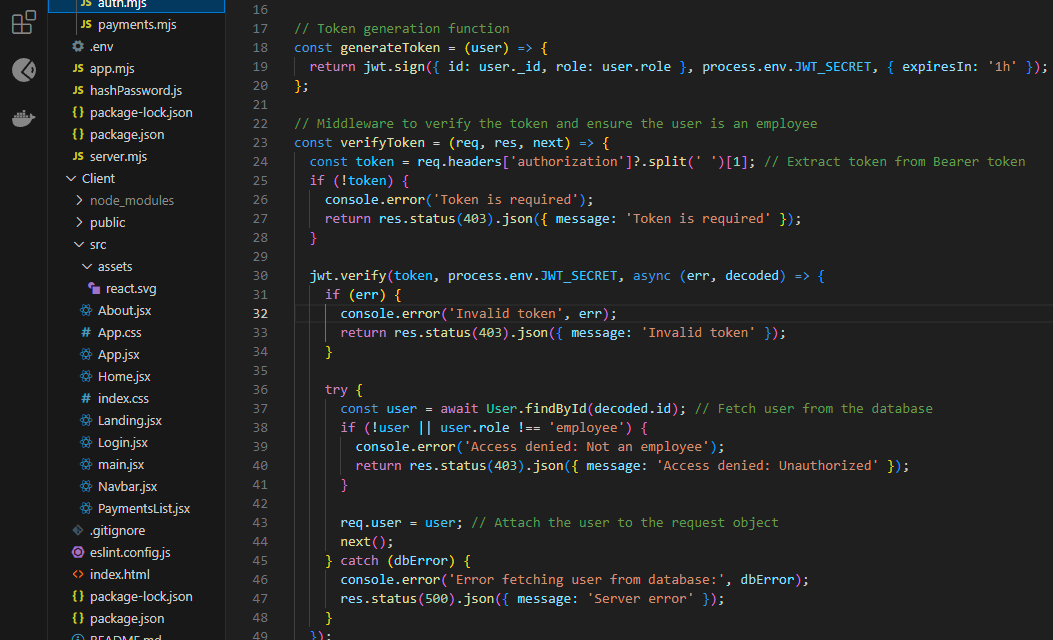
* Employee roles are enforced during login and API access.
* Only accounts with the role of employee can access sensitive endpoints like payment data.

A screenshot of a computer program

Description automatically generated

1. Secure Authentication:

* JWTs are generated upon login, ensuring secure stateless authentication.
* Tokens are validated for both identity and role, restricting access based on preconfigured roles.



## Customer Portal

Also has use roles implemented to prevent employees from accessing the user portal and vice-cersa.

1. Preconfigured Role Validation (User Role):

* Registration functionality assigns the user role by default.

The role field in the User schema for BOTH THE EMPLOYEE AND CUSTOMER PORTAL is assigned as "user" during registration so that default/non-employee accounts have the user role.

Backend\models\User.mjs:

A screen shot of a computer code

Description automatically generated

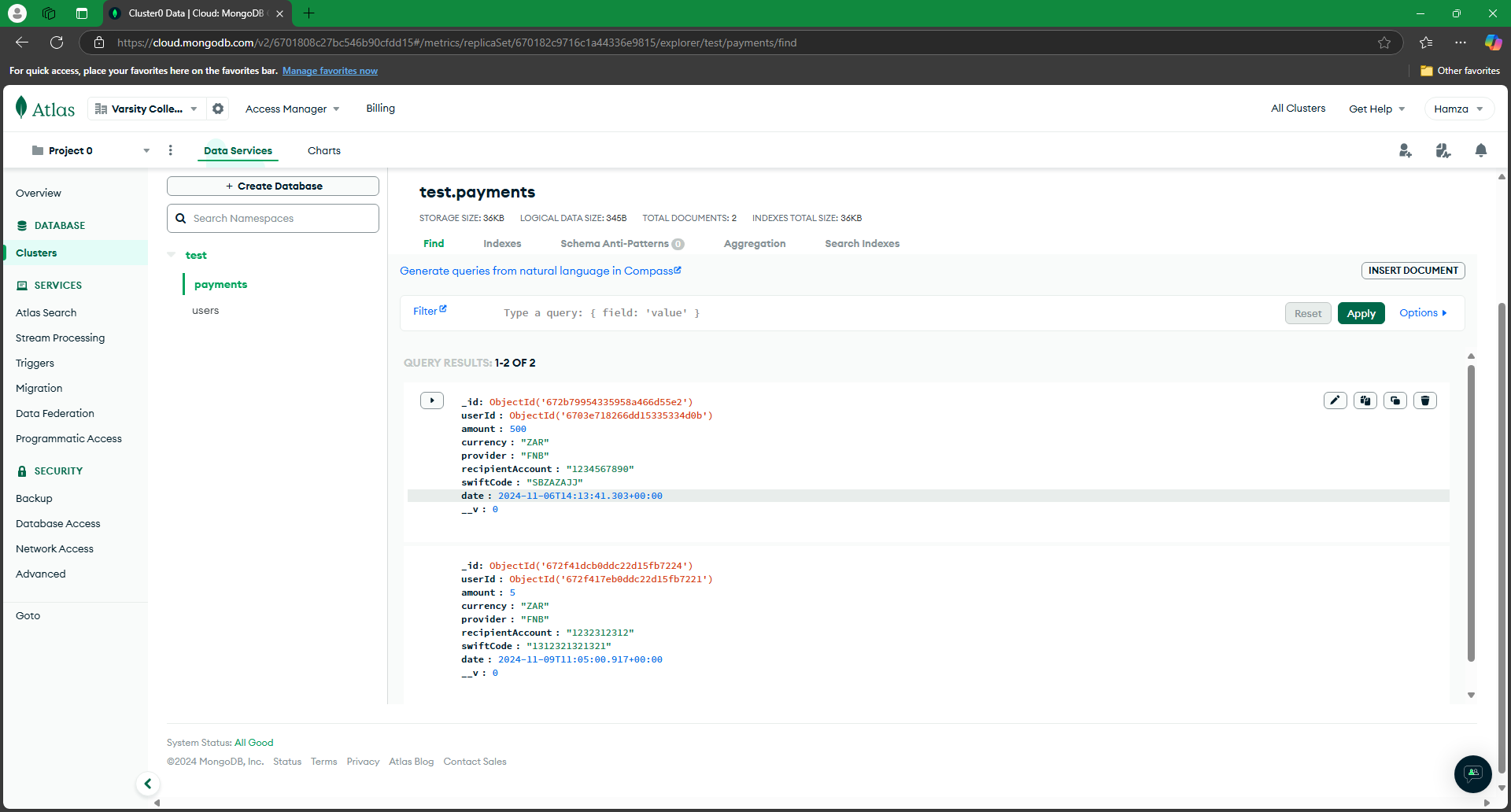
1. Role-Based Access for Payment Features:

* Role validation ensures that only accounts with the role user can access payment forms and associated routes.

1. JWT for Secure Authentication:

* JWT tokens are required for accessing all secured pages, ensuring users cannot access routes without proper authentication.

# Mongodb DatabaseA screenshot of a computer Description automatically generated

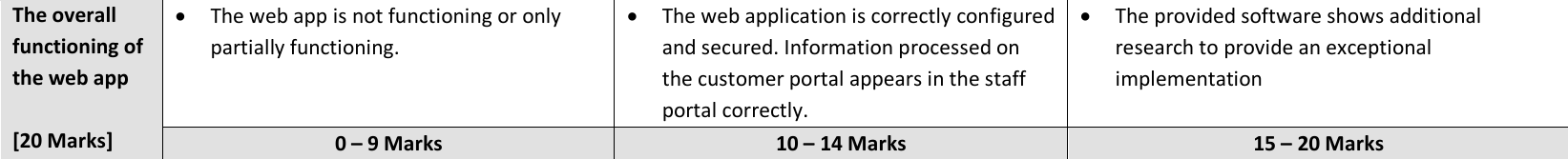


## References

* Auth0 Blog. Implement Role-Based Access Control in Node.js with Express.

Available at: <https://auth0.com/blog/role-based-access-control-rbac-and-node-js-api/>

* DigitalOcean. Building a Secure Role-Based Access Control System in Express. Available at: <https://www.digitalocean.com/community/tutorials/nodejs-role-based-access-control-api>
* JWT.io. JSON Web Tokens Introduction. Available at: <https://jwt.io/introduction>
* FreeCodeCamp. Using JSON Web Tokens (JWT) for User Authentication in Node.js. Available at: <https://www.freecodecamp.org/news/how-to-use-jwt-to-authenticate-and-authorize-users-in-node-js-d7c7e375d81e/>



# The overall functioning of the web app

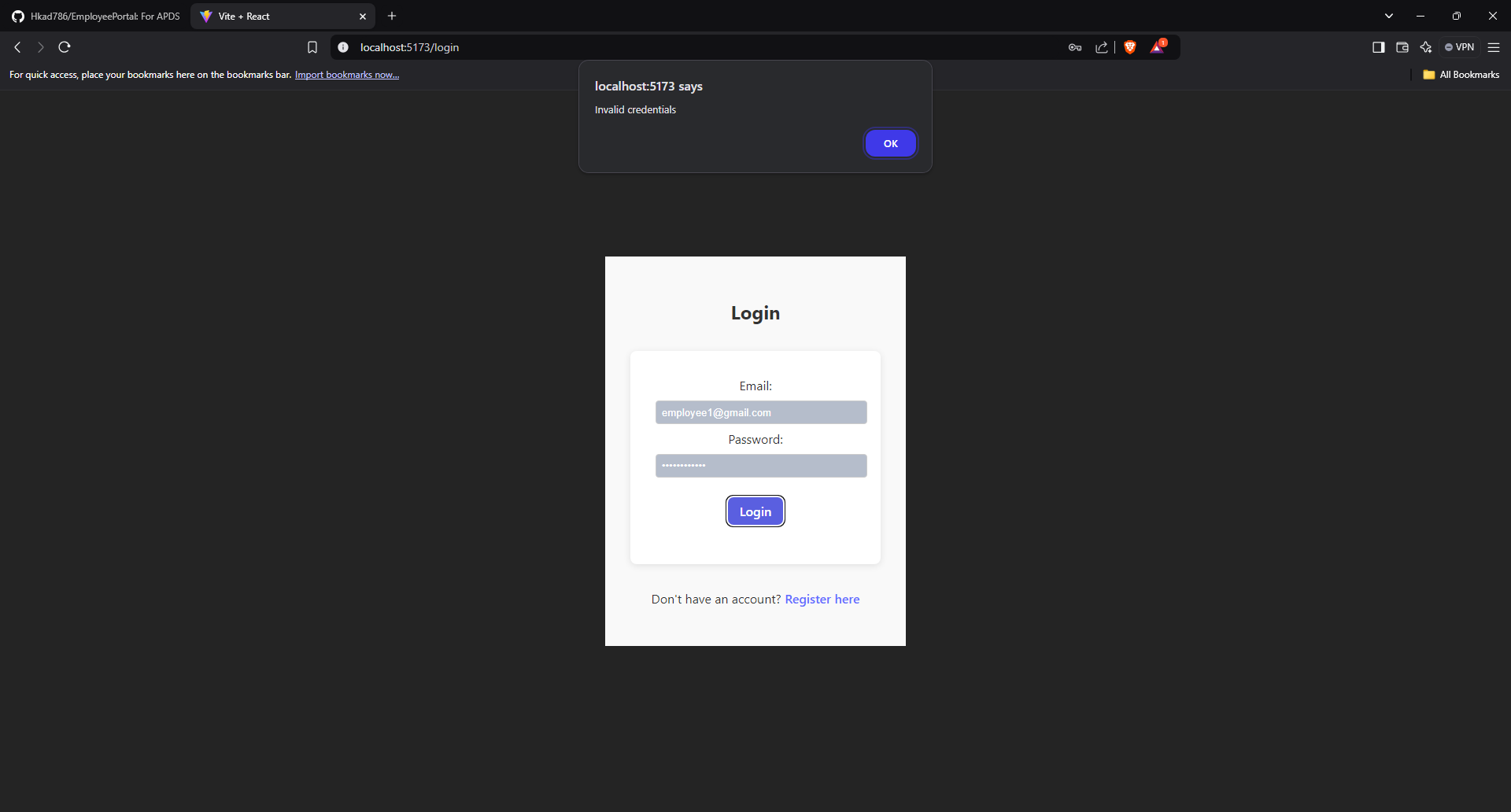
(Note: The Functionality of the app show below works hand in hand with the database shown above which would show the matching info along with a demo video showing all functionality possible in demo video)

1. Logging In and Role-Based Access Control:

The Customer portal only allows customer accounts to login and supplies them with a token and vice versa with the Employee portal.

**Customer Portal Functions shown.** A screenshot of a computer

Description automatically generated



Only users are allowed and employees are not allowed.

A screenshot of a computer

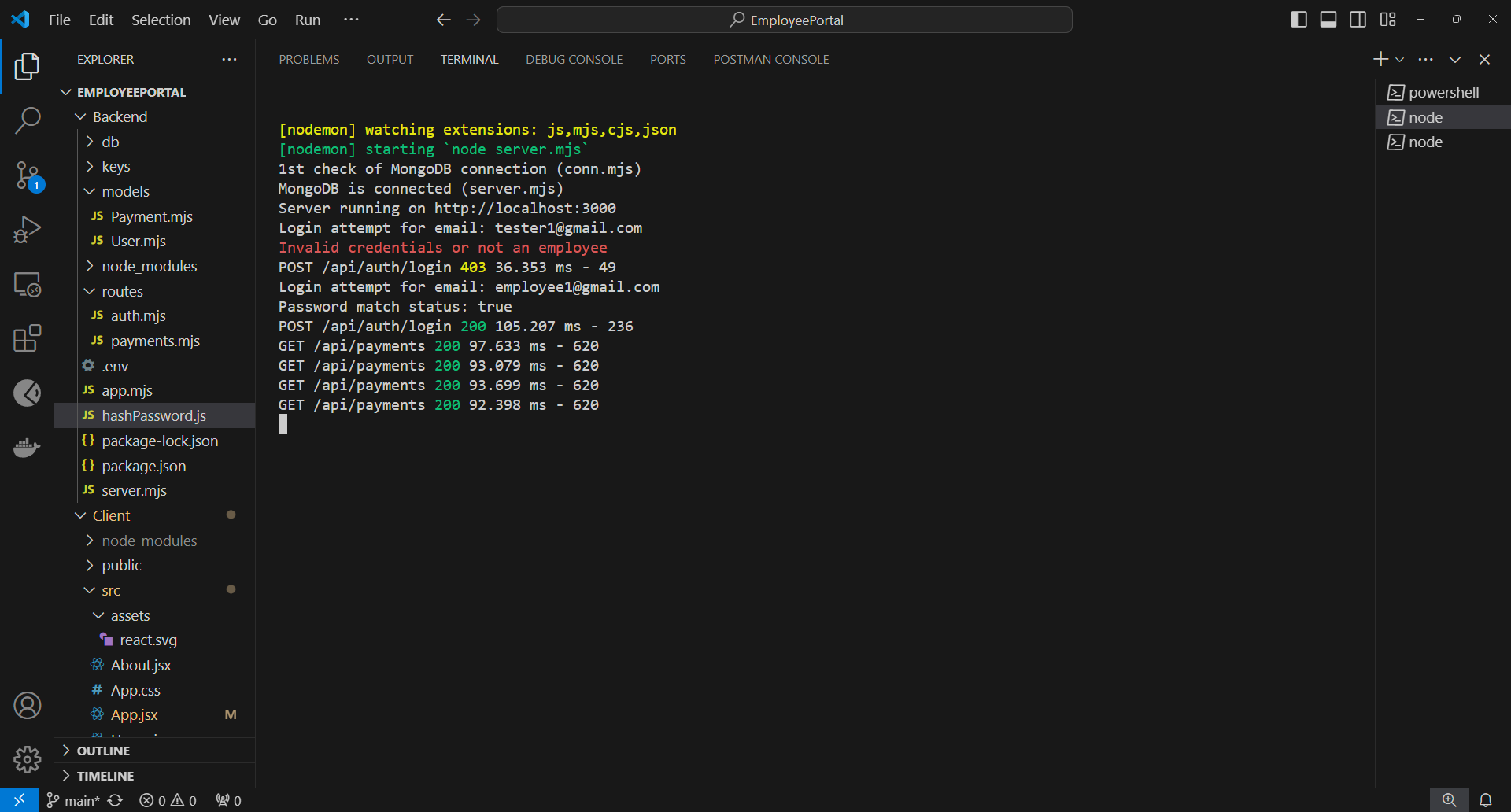
Description automatically generated

**Employee portal**

**A screenshot of a computer

Description automatically generated**

**User accounts don’t work on the employee portal which only has login and no register functionality.**

****