Security Enhancements Report

Name: Abdul Basit Rizwan
ID: DHC-251
Field: Cyber Security

Task: 2

Task 2.1 – Input Validation & Sanitization

Objective

Prevent invalid or malicious email inputs in the login route to secure the application against malformed data and injection attacks.

Implementation Steps

1. **Installed Validator Library**
   - Command: npm install validator
   - Purpose: Provides built-in functions to validate and sanitize inputs.
2. **Imported Validator in login.ts**
   - import validator from 'validator';
3. **Validated Email Input**
   - Extracted email and password from request body.
   - Checked if email is valid:
     - Invalid → rejected with error.
     - Valid → passed to next step.
4. **Sanitized Email**
   - Used normalizeEmail() to remove unwanted characters.
   - Ensured only the cleaned email was used in database queries.
5. **Testing**
   - Invalid email → Rejected with "Invalid email".
   - Valid email → Login successful.
   - Verified that only sanitized data reached the database.

Result

- Invalid inputs are blocked.
- Database only receives safe, normalized values.
- Reduced risk of injection attacks.

Task 2.2 – Password Hashing

Objective

Implement secure password storage using **bcrypt** to protect user credentials.

Theory

- **Hashing**: Converts plain text passwords into irreversible hashes.
- **Why bcrypt?**
  - Automatically salts passwords.
  - Resistant to brute-force attacks.
  - Adjustable computational cost.

Implementation Steps

1. **Installed bcryptjs** (npm install bcryptjs).
2. **Created helper functions** for hashing and comparing passwords.
3. **Updated User Model**:
   - Passwords automatically hashed before being stored.
4. **Updated Login Logic**:
   - User-provided password checked against stored hash using compareSync.

Result

- Plain passwords never stored in the database.
- Login works using original password (verified against hash).
- Database security significantly improved.

Task 2.3 – Token-Based Authentication (JWT)

Objective

Enhance login system using **JSON Web Tokens (JWT)** for stateless authentication.

Theory

- **JWT**: A signed token containing user data, used for secure session management.
- **Flow**:
    1. User logs in with email & password.
    2. Password verified (bcrypt).
    3. JWT generated with user info.
    4. Token sent to client.
    5. Client includes token in headers for subsequent requests.
    6. Server verifies token before granting access.

Benefits

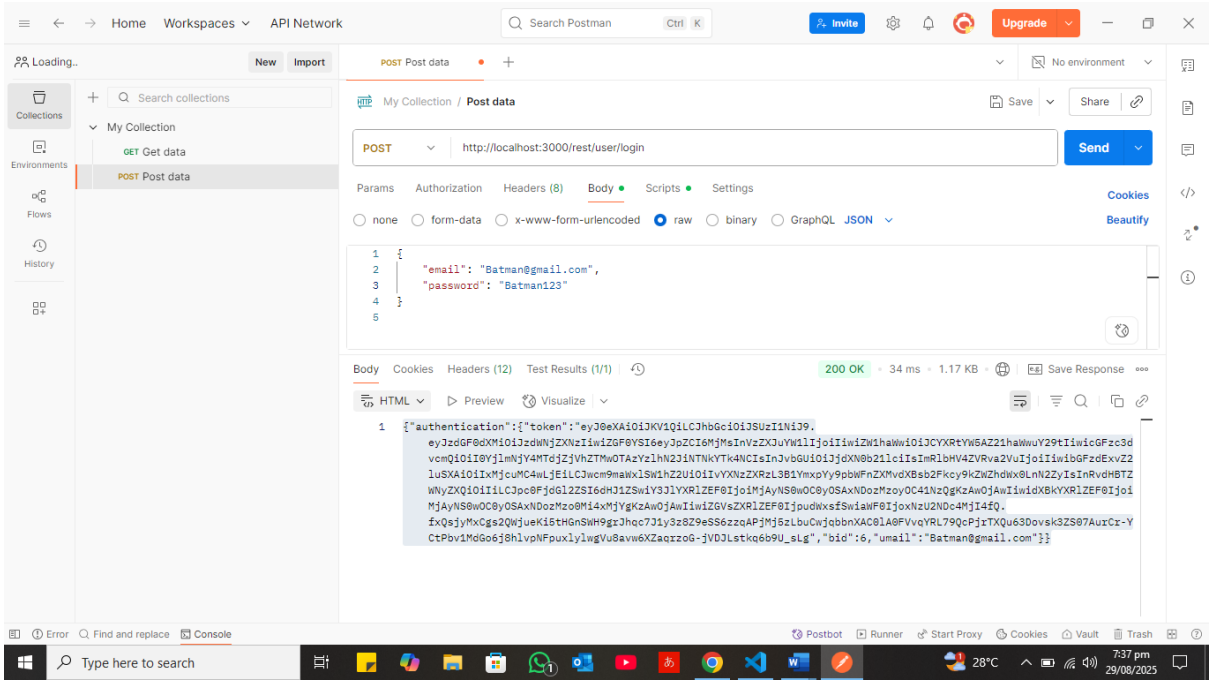
- No need to store sessions on server.
- Tokens can expire (e.g., 6 hours).
- Scales well for APIs and microservices.

Implementation Steps

1. **Installed jsonwebtoken & jws**.
2. **Generated JWT** after successful login.
3. **Verified JWT** for protected routes.
4. **Integrated with login.ts** to issue tokens.

Result

- Users receive JWT after login.
- Protected routes only accessible with valid token.
- Authentication is secure, stateless, and scalable.



Task 2.4 – Helmet.js Security Headers

Objective

Enhance HTTP response security by adding **Helmet.js** middleware.

Implementation Steps

1. **Installed Helmet** (npm install helmet).
2. **Integrated Helmet in server.ts** with minimal config:
    - Disabled CSP & COEP (to avoid breaking Juice Shop features).
3. **Restarted application** and verified headers.

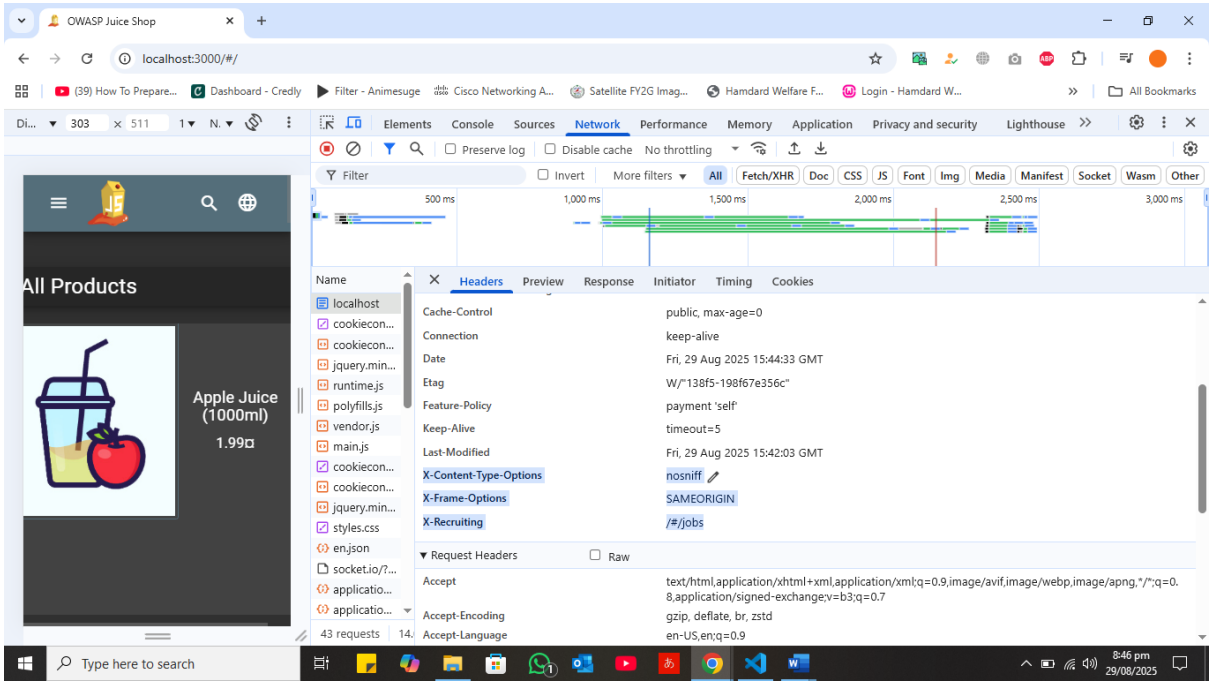
Testing

- **Browser DevTools** and **PowerShell** used to check headers.
- Confirmed headers like:
    - X-Content-Type-Options: nosniff
    - X-Frame-Options: SAMEORIGIN

✅ Result

- Helmet applied multiple security headers.

- Reduced risk of clickjacking, MIME-type sniffing, and framing attacks.



Final Conclusion

Through a step-by-step security enhancement process:

- **Input Validation & Sanitization** ensured only safe data is processed.
- **Password Hashing** with bcrypt secured stored credentials.
- **JWT Authentication** enabled stateless, token-based access control.
- **Helmet.js** strengthened HTTP response headers for secure communication.
