API Integration Project Documentation

1. Requirement Gathering

Functional Aspects:

- **User Registration**:
  - Users can register by providing a username, password, gender, and location.
  - Validation ensures all fields are filled and the password is at least 8 characters long.
- **User Login**:
  - Users can log in by providing their username and password.
  - If the credentials are correct, a JWT token is generated and returned.
- **Get Users**:
  - Authenticated users can retrieve a list of all users, including their ID, username, gender, and location.
- **Update User**:
  - Authenticated users can update their profile details (name, gender, location).
- **Delete User**:
  - Authenticated users can delete their account if it exists.

Non-Functional Aspects:

- **Performance**:
  - The database connection is initialized once and used for all queries.
  - Prepared statements are used to enhance query performance and security.
- **Scalability**:
  - The API uses Express, which is scalable and suitable for handling multiple requests.
- **Security**:
  - JWT-based authentication is implemented to secure endpoints.
  - Passwords are hashed using bcrypt before storing in the database.
  - Validation ensures proper input data.

2. Create a Simple API

Endpoints for CRUD Operations:

- **Create User**: `POST /users/`
  - Allows creating a new user account with validation for required fields and password length.
- **Read Users**: `GET /users`
  - Retrieves a list of all users (authenticated endpoint).
- **Update User**: `PUT /users/:userId`
  - Updates user details (name, gender, location) for an authenticated user.
- **Delete User**: `DELETE /users/:userId`
  - Deletes an existing user account if the user exists (authenticated endpoint).

3. Implement API Authentication

Security Measures:

- **JWT Authentication**:
    - The `authenticateToken` middleware ensures that only authenticated requests can access the endpoints.
    - JWT tokens are generated on successful login and must be included in the Authorization header for protected routes.
- **Data Validation**:
    - The `validateRegisterUser` middleware checks that all required fields are provided and the password meets the minimum length requirement.
    - Prevents SQL injection by using parameterized queries.

Detailed Explanation:

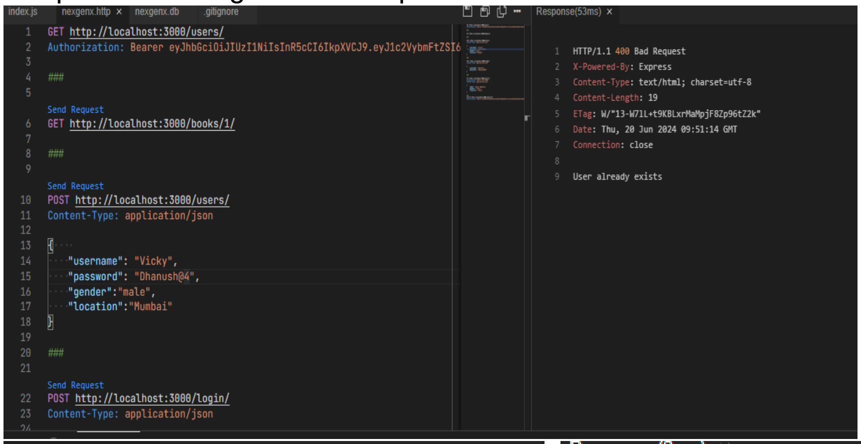
- **Database Initialization**:
    - The database connection is established once when the server starts, ensuring efficient handling of subsequent queries.
- **Token-Based Authentication**:
    - JWT tokens are used to secure endpoints, ensuring that only authenticated users can access or modify data.
    - Tokens are verified for each protected route, and the username is extracted from the token for subsequent operations.
- **Validation Middleware**:
    - Ensures all registration fields are provided and the password is at least 8 characters long.
    - Prevents user creation with missing or invalid data.
- **User Management**:
    - Users can register, log in, view all user profiles, update their own profile, and delete their account.
    - The API responds with appropriate status codes and messages for successful and failed operations.

5. Error Handling in APIs

- **Objective:** Implement robust error handling in the API.

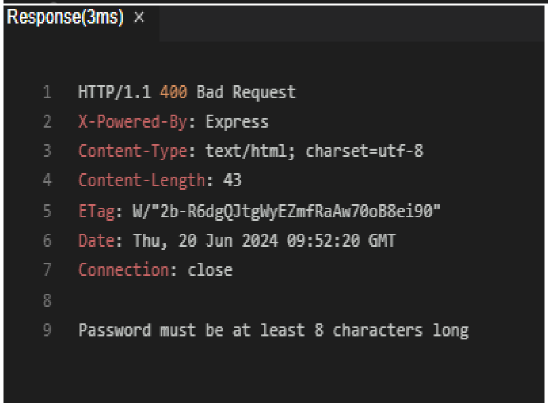
Instructions: Add proper error handling to the API created in Task 2. Ensure that appropriate HTTP status codes are returned for different error scenarios, such as 404 for not found, 400 for bad requests, and 500 for server errors.

Responses In Register User Api:

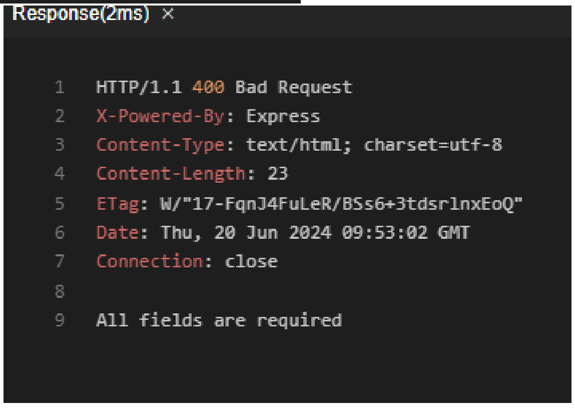


```
GET http://localhost:3000/users/
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6

###

Send Request
GET http://localhost:3000/books/1/

###

Send Request
POST http://localhost:3000/users/
Content-Type: application/json

{
    "username": "Vicky",
    "password": "Dhanush@4",
    "gender":"male",
    "location":"Mumbai"
}

###

Send Request
POST http://localhost:3000/login/
Content-Type: application/json
```

```
HTTP/1.1 400 Bad Request
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 19
ETag: W/"13-W71L+t9KBLxrMaMpjF8Zp96tZ2k"
Date: Thu, 20 Jun 2024 09:51:14 GMT
Connection: close

User already exists
```

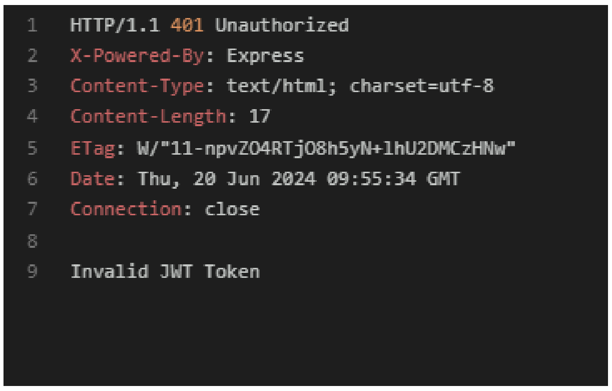


```
HTTP/1.1 400 Bad Request
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 43
ETag: W/"2b-R6dgQJtgWyEZmfRaAw70oB8ei90"
Date: Thu, 20 Jun 2024 09:52:20 GMT
Connection: close

Password must be at least 8 characters long
```



```
HTTP/1.1 400 Bad Request
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 23
ETag: W/"17-FqnJ4FuLeR/BSs6+3tdsrlnxEoQ"
Date: Thu, 20 Jun 2024 09:53:02 GMT
Connection: close

All fields are required
```

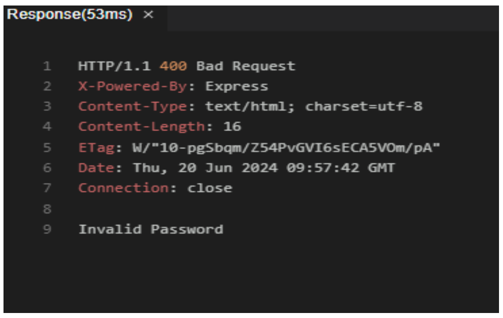
GET USERS API:



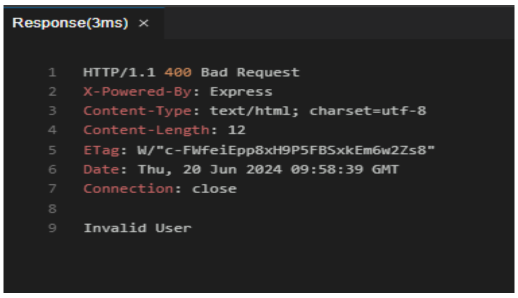
```
HTTP/1.1 401 Unauthorized
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 17
ETag: W/"11-npvZO4RTjO8h5yN+1hU2DMCzHNw"
Date: Thu, 20 Jun 2024 09:55:34 GMT
Connection: close

Invalid JWT Token
```

Getting response as 401 when jwt token is invalid.

LOGIN USER API:



```
HTTP/1.1 400 Bad Request
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 16
ETag: W/"10-pgSbqm/Z54PvGVI6sECA5VOm/pA"
Date: Thu, 20 Jun 2024 09:57:42 GMT
Connection: close

Invalid Password
```



```
HTTP/1.1 400 Bad Request
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 12
ETag: W/"c-FWfeiEpp8xH9P5FBSxkEm6w2Zs8"
Date: Thu, 20 Jun 2024 09:58:39 GMT
Connection: close

Invalid User
```

10. Security Audit

- **Objective:** Perform a security audit of the API.
- **Instructions:** Conduct a basic security audit of the API created in Task 2. Check for common vulnerabilities, such as SQL injection, cross-site scripting (XSS), and improper authentication. Document your findings and suggest improvements.

With respect to the above task the ascpects I have implemented in my code are

Common Vulnerabilities Checked:

1. **SQL Injection**
2. **Improper Authentication**

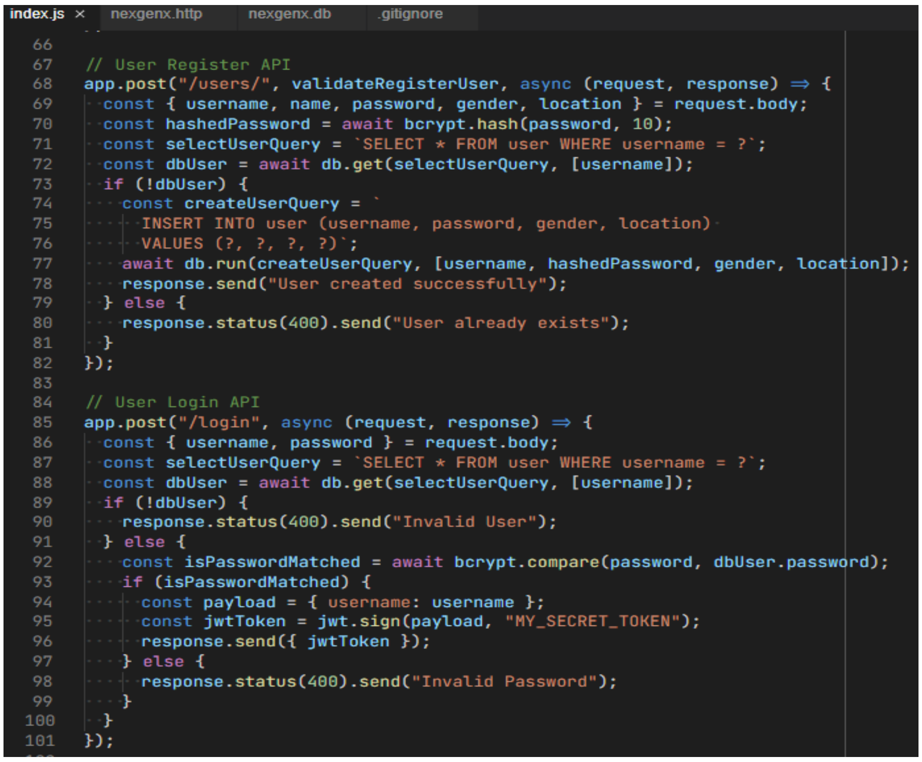
Findings and Implementations:

1. **SQL Injection**:

   - ❖ **Vulnerability**: SQL injection can allow attackers to manipulate SQL queries by inserting malicious SQL code through input fields.
   - ❖ **Implementation**:

     - ➢ **Prepared Statements**: The API uses parameterized queries to safely inject user inputs into SQL queries, preventing SQL injection attacks.
     - ➢ **Improvement**: Continue using parameterized queries for all SQL operations to maintain protection against SQL injection.

Below image represent the Parametrised way of giving sql query, in order to prevent sql injection.



```javascript

// User Register API
app.post("/users/", validateRegisterUser, async (request, response) => {
  const { username, name, password, gender, location } = request.body;
  const hashedPassword = await bcrypt.hash(password, 10);
  const selectUserQuery = `SELECT * FROM user WHERE username = ?`;
  const dbUser = await db.get(selectUserQuery, [username]);
  if (!dbUser) {
    const createUserQuery = `
      INSERT INTO user (username, password, gender, location)
      VALUES (?, ?, ?, ?)`;
    await db.run(createUserQuery, [username, hashedPassword, gender, location]);
    response.send("User created successfully");
  } else {
    response.status(400).send("User already exists");
  }
});

// User Login API
app.post("/login", async (request, response) => {
  const { username, password } = request.body;
  const selectUserQuery = `SELECT * FROM user WHERE username = ?`;
  const dbUser = await db.get(selectUserQuery, [username]);
  if (!dbUser) {
    response.status(400).send("Invalid User");
  } else {
    const isPasswordMatched = await bcrypt.compare(password, dbUser.password);
    if (isPasswordMatched) {
      const payload = { username: username };
      const jwtToken = jwt.sign(payload, "MY_SECRET_TOKEN");
      response.send({ jwtToken });
    } else {
      response.status(400).send("Invalid Password");
    }
  }
});
```

2. **Improper Authentication**:

   - **Vulnerability**: Inadequate authentication can allow unauthorized access to protected resources.
   - **Implementation**:

     JWT Authentication: The API employs JSON Web Tokens (JWT) for authentication. Tokens are issued upon successful login and must be included in the Authorization header for accessing protected endpoints.

     Token Verification: A middleware function verifies the JWT for each protected route and extracts the username from the token.

     Improvement:

       Store the JWT secret securely, using environment variables instead of hardcoding it in the source code.

       Implement token expiration and refresh mechanisms to enhance security.

3. **Input Validation**:
   - **Vulnerability**: Insufficient input validation can lead to attacks such as SQL injection and data corruption.
   - **Implementation**:
     - **Validation Middleware**: A middleware function checks that all required fields are provided and that the password meets the minimum length requirement.
     - **Improvement**:
       - Implement additional validation rules for other input fields, such as username format and location validity.
       - Use a validation library for more comprehensive and maintainable validation rules.
4. **Endpoint Security**:
   - **Vulnerability**: Unauthorized access to sensitive endpoints can compromise user data.
   - **Implementation**:
     - **Protected Endpoints**: All endpoints that modify or access user data are protected by a middleware function ensuring that only authenticated users can access them.
     - **Improvement**:
       - Ensure robust authorization checks to verify that authenticated users can only access and modify their own data.
       - Regularly review and update security policies to adapt to emerging threats.