Daniel Jones
John Habakus

# Sprint 3: Multi-Layered Authentication and Testing

**User Registration and Login:**

The registration and login forms were implemented as two separate components named Register.js and Login.js. Each form uses React to track username, password, and errors. When the forms are submitted, HTTP Post requests are sent to the backend, where either an account is created (register) or an existing account is authenticated (login). The backend routes for register and login are implemented using Express. They're handled in authController.js in the client-service folder, which validates the input, interacts with the sqlite database, and returns the correct responses. The authentication functions are mounted within server.js under /api/auth, separating them from the rest of the API. Within authController.js, we hash passwords using bcrypt.hash(password, 10), and save the resulting hash within the sqlite database, along with the user ID and the username. We do not save passwords themselves, as doing so is a security risk. When a login is attempted, the input password is compared with the hash using bcrypt.compare, and if it's correct the user is admitted. Passwords are not stored in plaintext anywhere. Once the user is verified and the login is successful, a JWT is created with the user ID and username. User information is defined within the User.js file, which contains a basic insertion function for the sqlite database, as well as a getter function for User names.

**Logout & Session Handling:**

On the main page of the website (App.js) is a logout button, just below the header. Clicking it will log the user out (by setting the current user to null), and redirect the user to the Login page. The Login page also has a Register button for ease of use, which redirects the user to the Register page. There is a small display at the top of the website's main page which displays the username of the currently active user. We have a file AuthContext.js which defines the interaction between the frontend login/register/logout functions and their backend implementations. The index.js file has been changed to wrap App.js in two layers of protection: the first is ProtectedRoute.js, which replaces things with the login page, as well as AuthContext.js, which sets the user to whatever is most fitting for the situation (null, user, etc). The html and handling for logging in and registering are handled in Login.js and Register.js, respectively.

**Token-Based Authentication and Password Hashing:**

The JWT created in the backend is a token created by jsonwebtoken. It's sent back to the user as an HTTP-only cookie to improve security. The token is verified by sensitive routes in the backend, such as user profiles, bookings, and purchasings, so that these operations cannot be performed without a valid token. Automatic expiration is implemented in authController.js. The token is given an expiration date of thirty minutes. After that, the user has to log in again.

In the authController.js file, within the register function, the two lines below exist on lines 17 and 19. We hash the password, then make the new User with the hash. There are a few reasons for this. If we stored raw passwords, then someone hacking our database would be able to access every registered account. By hashing, we transform the password into a string in a way that can't be easily undone. Even if two users have the same password, their hashes will be different from one another. Later, on line 36 in the login function, we compare the input password to the password hash to check that they're the same.

```
17        const passHash = await bcrypt.hash(password, 10);

19        const user = await createUser(username, passHash);
 ...
36        const valid = await bcrypt.compare(password, user.password_hash);
```

As seen below, we've implemented token-based authentication in our login function, as well as in the middleware and the frontend (Lines 45-62 in authController and the entirety of AuthContext.js in the frontend). The token authentication is used to check whether or not a user is actually signed in. If they want to access something protected, or something that's sensitive, the middleware can check the token's signature and expiration before we let the request proceed. We can also (not yet implemented) record which tokens were used to make which requests on the site. With tokens, only someone who is logged in and has a valid token can make requests. This makes it more difficult for someone to mess with the system, or overload the system with purchase requests.

JWT HTTP only cookie creation:

```
        res.cookie("token", token, {
            httpOnly: true,
            secure: process.env.NODE_ENV === "production",
            sameSite: "strict",
            maxAGE: JWT_EXPIRES_IN,
        });
...
```

Clearing cookie on logout:

```
export const logout = async (req, res) => {
    res.clearCookie("token");
    res.json({message: "Logged out."});
};
```

An example of token authentication in clientRoutes.js, this can also be found in authRoutes.js:

```
import {verifyToken} from '../middleware/authMiddleware.js';
const router = express.Router();

// Client routes
router.get('/events', listEvents);
router.post('/events/:id/purchase', verifyToken, buyTicket);

// Booking routes
router.post('/prepare-booking', verifyToken, prepareBooking);
router.post('/confirm-booking', verifyToken, confirmBooking);
```

The verifyToken function called in Routes (implementation in authMiddleware.js):

```
export const verifyToken = (req, res, next) => {
    const token = req.cookies?.token || req.headers['authorization']?.split('
')[1];
    if (!token) {
        return res.status(401).json({message: "Unauthorized: no token
provided."});
    }
    try {
        const decoded = jwt.verify(token, JWT_SECRET);
        req.user = decoded;
        next();
    }catch (err) {
        return res.status(401).json({message: "Session expired. Please log in
again."});
    }
};
```

Even with hashed passwords and JWT authentication, the software still has vulnerabilities. If a user has an easily guessable password, their account can be accessed. If the database is compromised, the hashes can be brute forced. People can perform brute force attacks on the login, since we haven't limited the number of attempts there, and we don't have a CAPTCHA. Attackers might also find a way to make their own tokens, and if tokens are stolen from the client machine, they can be used until they expire.

**Testing**

Unit, integration, and end-to-end tests are all important elements of the testing process. This is due to the fact that they all cover different scenarios. Unit testing is useful for testing elements efficiently by using mock functions to test logic, for example simulating user registration without actually registering a new user. Integration testing is more time and resource intensive but also more accurate, since it involves actually utilizing the active resources. For example, checking if a real test account exists. Finally, end-to-end tests cover the full length of the project, making sure that every element is working from start to finish.

**Testing Strategy**

Utilize unit, integration, and end-to-end testing to ensure user authentication is functional and secure.

**Backend Tests - Utilizing Jest**

- userModel
    - Creates user and retrieves it by username
- authController
    - Creates new user
    - Does not create if user exists
    - Logs in with correct credentials
    - Does not log in with incorrect credentials

    - Does not log in if user does not exist
- authIntegration
    - Returns current user if it exists
    - Returns 401 if no token
    - Returns 401 if token is invalid

**Frontend Tests - Utilizing Jest**

- authE2E
    - Logs in from frontend with correct credentials
    - Does not log in from frontend with incorrect credentials

**Manual Tests**

- Test 1 - logging in with incorrect credentials
    - Input invalid credentials
    - Validate that login fails
- Test 2 - log in with expired token
    - Attempt to login after token expires
    - Validate that login fails
- Test 3 - log out
    - Attempt to log out from account
    - Validate that account logs out

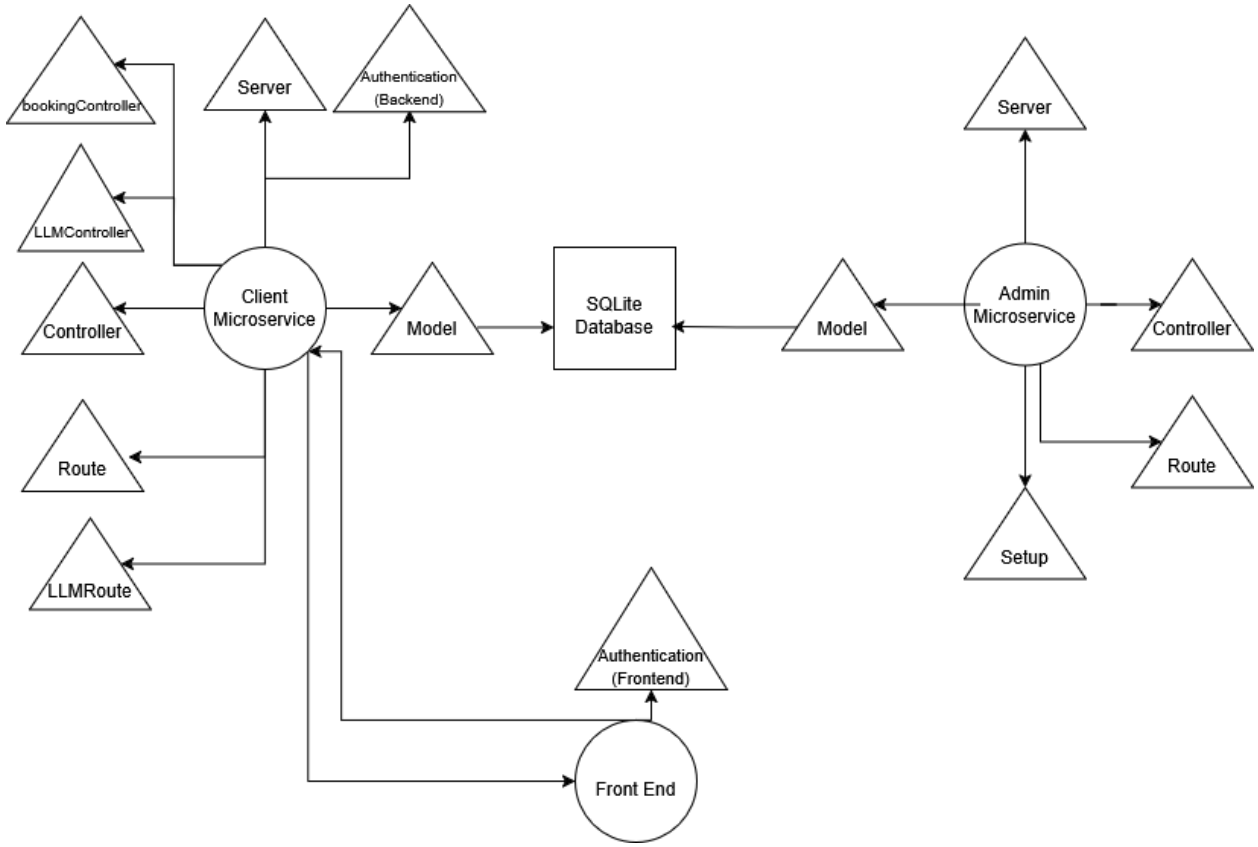
**Test Log**





Test 1: Expected “Invalid username or password.”, Found “Invalid username or password.”
Test 2: Expected “Invalid or expired token.”, Found “Invalid or expired token.”
Test 3: Returned to log in screen when attempting to log out

**Architecture Diagram**




**Accessibility Description**

- Fully keyboard-navigable UI
- Screen reader support for interactive elements
- Voice Input Capture
- LLM Chat Integration through Voice Input Capture and LLM Parsing

**Code Quality**

- **Function Documentation & Comments**
  - Function headers used
  - Ex. voiceChat.js, lines 3-5
  - In-line comments used
- **Variable Naming & Code Readability**
  - Lines are kept short
  - Consistent capitalization of variables and functions
  - Variable and function names are clear
- **Modularization**
  - Helper functions are used
  - Small functions are preferred
- **Error Handling**
  - Inputs are validated

    - Proper error codes are used
    - If incorrect inputs, call is canceled and error is retired
    - Ex. lines 46-51, bookingController.js
- **Consistent Formatting**
    - Consistent formatting is used throughout the code