TigerTix - Sprint 3 Report

---

1. Summary of Sprint 3

The goal was to implement a multi-layered authentication microservice and comprehensive regression testing covering backend, frontend, LLm booking, voice interaction, and accessibility. The idea of user authentication is to deliver secure login, token based protection for sensitive routes, and logout handling. We covered the majority of testing in sprint two but added a few new tests to handle the new updates.

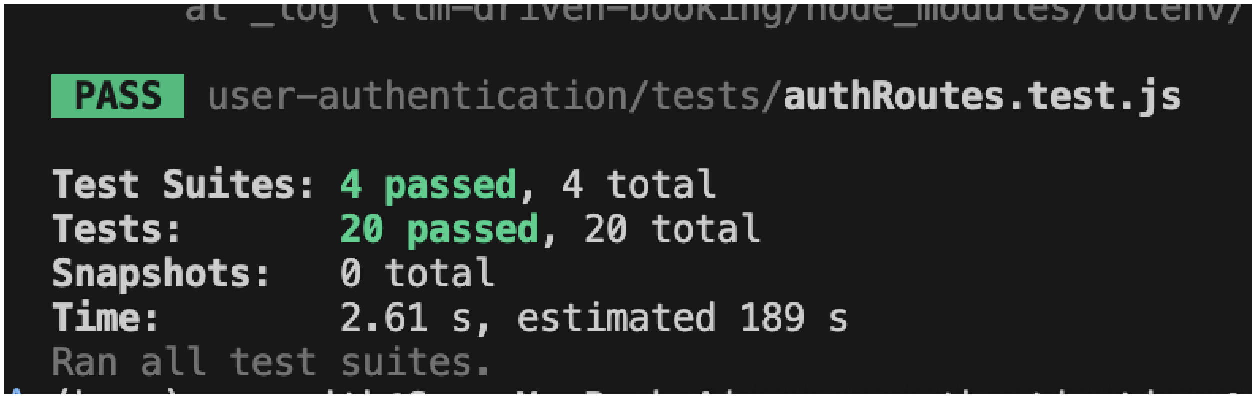
2. Implementation

2.1 Frontend

- login.js and frontend/src/componenets/register.js is what forms client-side validation, accessibility labels, and visible user state indicator “logged in as user@example”
- Token handling: on a successful login the frontend receives the JWT in an HTTP only cookie, stores it in in-memory React and never in localstorage
- Logout button clears cookie via backend endpoint and resets the React auth context

2.2 Backend

- server.js: Express app exposing POST / register, POST /login, POST /logout
- authController.js: registration uses bcrypt.hash. Login uses bcrypt.compare
- authMiddleware.js: verifies JWT using jsonwebtoken.verify; rejects with 401 on invalid or expired token
- Token expiration is set to 30 minutes. On expired token frontend redirects to login screen when receiving 401 or when token refresh fails



```
    at _log (ttm-driven-booking/node_modules/dotenv/

 PASS  user-authentication/tests/authRoutes.test.js

Test Suites: 4 passed, 4 total
Tests:       20 passed, 20 total
Snapshots:   0 total
Time:        2.61 s, estimated 189 s
Ran all test suites.
```

(ran all of our tests but shows that all 4 of the authRoutes.tests passed)

3. Testing

Testing and testing documentation was completed in sprint 2. Additional tests for user-authentication will be added here. Tools used were Jest, and Supertest.

3.1 Tests for user-Authentication

- Test 1: Registration Creates a user and Sets JWT
  - Purpose: Ensure new users can register successfully
  - What it Verifies
    - A new user row is inserted into the test database
    - Response includes the created user object
    - A valid JWT auth cookie is sent back
- Test 2: Duplicate email returns 409
  - Purpose: ensure duplicate emails are rejected
  - What it verifies
    - First registration succeeds
    - Second attempt returns 409 conflict
    - Error message contains “email already registered”
- Test 3: Login Authenticates User and Issues JWT Cookie]
  - Purpose: Confirm Login works with hashed passwords
  - What it verifies
    - Login succeeds only with correct password
    - Response includes the user object]
    - A fresh JWT cookie is set
- Test 4: Logout clears Authentication cookie
  - Purpose: Ensure sessions can be safely terminated
  - What it verifies
    - Logout returns a success message
    - The token cookie is cleared or set to an expired value

3.2 Manual Tests

- Test 1: Login with incorrect credentials
  - Purpose: validate proper error handling for failed authentication
  - Steps
    - Navigate to login page
    - Enter a valid email but incorrect password
    - Expected result: red error message displaying invalid credentials
- Test 2: Token Expiration Behavior
  - Purpose: Ensure expired JWT forces the user back to login page
  - Steps
    - Log in and verify login
    - Manually modify the browser cookie or wait 30m
    - Attempt to navigate to a protected route
    - Expected Result: User is redirected to login screen
- Test 3: Logout Flow
  - Purpose: Ensure that logout clears authentication cookies and sessions state
  - Steps
    - Log in
    - Click logout button
    - Attempt to visit /profile again
    - Expected results: JWT Cookie removed and returns 401
- Test 4: Registration input validation
  - Purpose: Ensure form errors are readable and accessible
  - Steps
    - Open registration app
    - Submit form with missing fields
    - Navigate with keyboard only
    - Expected results: Screen reader announces error messages

4. Accessibility

- Semantic HTML
  - Used <form>, <label>, <input> appropriately
  - Each input has a <label> with html for connected ot the inputs id
- ARIA and focus handling
  - Error messages are placed where screen readers can reach them
  - Focus automatically returns to the first invalid field on failed submit
  - Keyboard users can tab through fields and buttons in logical order
- VisualContrast
  - Text and buttons on the auth screens use colors with sufficient contrast against the background

    - Focus state is visible for interactive elements

5. Security

- Why hashed Passwords
    - If the database is leaked, attackers only see hashes, not raw passwords
    - Bycrpt applies a salt and multiple rounds, slowing down brute force attacks
    - This significantly reduces the impact of a database breach
- Why use JWTs?
    - Allows stateless authentication across microservices without a shared session store
    - Each request carries its own proof of authentication via token
    - Tokens have an expiration to limit the time window if a token were leaked
    - The token is stored only in an http cookie which helps protect it from the javascript based attacks
- Remaining Security Vulnerabilities
    - We still need to ensure we sanitize any user input rendered in the frontend
    - Currently, repeated failed attempts are not limited
    - When using cookies, cross site requests could be an issue

6. Why Unit, integration, and end to end tests?

- Unit tests
    - Test small functions in isolation
    - Fast and easy to run, good for catching logic bugs early
    - When it's used in Project

```
// backend/user-authentication/controllers/authController.js
const bcrypt = require('bcryptjs');

async function hashPassword(password, rounds = 10) {
  return bcrypt.hash(password, rounds);
}

module.exports = { hashPassword };

// backend/user-authentication/tests/hashHelper.test.js
const bcrypt = require('bcryptjs');
jest.mock('bcryptjs');

const { hashPassword } =
require('../../user-authentication/controllers/authController');
```

```
describe('hashPassword helper', () => {
  it('calls bcrypt.hash with provided rounds and returns hash',
async () => {
    bcrypt.hash.mockResolvedValue('mocked-hash-value');

    const result = await hashPassword('plainPassword', 12);

    expect(bcrypt.hash).toHaveBeenCalledWith('plainPassword',
12);
    expect(result).toBe('mocked-hash-value');
  });
});
```

This unit test isolates the hashing helper; it runs fast and verifies we call bcryot.hash correctly without starting the server or touching the db

- Integration tests
    - They ensure the controller, route, and database logic work together not just in isolation
    - In the project example

```
// backend/client-service/tests/purchase.integration.test.js
const request = require('supertest');
const express = require('express');
const bodyParser = require('body-parser');
const fs = require('fs');
const path = require('path');
const sqlite3 = require('sqlite3').verbose();

const clientRoutes = require('../routes/clientRoutes');

// Mock auth middleware to simulate logged-in user
jest.mock("../../user-authentication/middleware/authMiddleware", ()
=> {
  return (req, res, next) => {
    req.user = { id: 1, email: "test@example.com" };
    next();
  };
});

const TEST_DB = path.join(__dirname, 'test_simple.sqlite');

// Reset test DB with one event before each test
beforeEach((done) => {
  if (fs.existsSync(TEST_DB)) fs.unlinkSync(TEST_DB);
  const db = new sqlite3.Database(TEST_DB);
  db.serialize(() => {
    db.run(`
      CREATE TABLE events (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT NOT NULL,
        date TEXT NOT NULL,
        tickets_total INTEGER NOT NULL,
        tickets_sold INTEGER NOT NULL DEFAULT 0
      );
    `, () => {
      db.run(`INSERT INTO events (name, date, tickets_total,
tickets_sold) VALUES ('Test Event', '2025-12-01', 10, 0)`, done);
    });
  });
});

const app = express();
app.use(bodyParser.json());
app.use('/api', clientRoutes);

test('POST /api/events/:id/purchase increments tickets_sold',
async () => {
  const res = await request(app).post('/api/events/1/purchase');
  expect(res.status).toBe(200);
  expect(res.body.event).toHaveProperty('tickets_sold', 1);

  // verify the DB state via the list endpoint
  const list = await request(app).get('/api/events');
  expect(list.body[0].tickets_sold).toBe(1);
});
```

This integration test exercises the HTTP route, route handlers, model lviv that reads/writes the DB, and auth middleware - al together. It catches issues that unit tests miss

- End to end tests
    - Simulate real user behavior: open app -> fill something -> submit -> access -> log out
    - These catch issues that only appear when all layers are involved
    - Example in Project

```
// e2e/login.spec.js
const { test, expect } = require('@playwright/test');
```

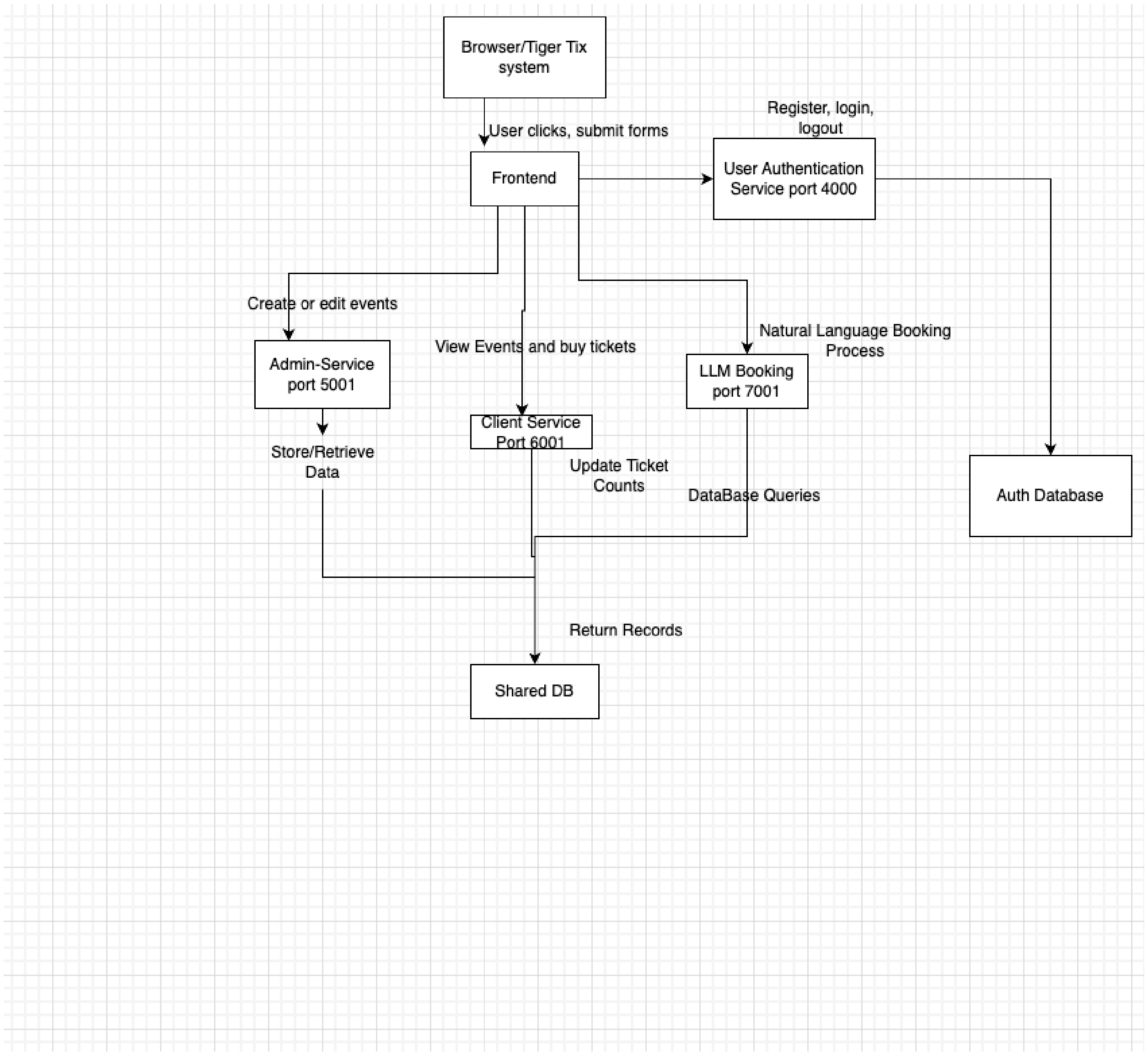
```
test('login -> access protected profile -> logout', async ({ page })
=> {
  await page.goto('http://localhost:3000');

  // open login form (adjust selectors to your UI)
  await page.click('text=Login');

  // fill and submit the login form
  await page.fill('input[name="email"]', 'test@example.com');
  await page.fill('input[name="password"]', 'password123');
  await page.click('button[type="submit"]');

  // verify UI shows logged-in state (adjust text to your app)
  await expect(page.locator('text=Logged in as
test@example.com')).toBeVisible();

  // visit protected route and assert we can access it
  await page.goto('http://localhost:3000/profile');
  await expect(page).toHaveURL(/\/profile/);
  await expect(page.locator('h1')).toContainText('Profile');

  // logout and verify returned to logged-out UI
  await page.click('text=Logout');
  await expect(page.locator('text=Login')).toBeVisible();
});
```

This end to end test verifies the full stack behavior. That the front end stores receive the JWT, protect routes are enforced, and the logout flow clears session slate.

7. Code Quality Standards

- Modular structure
  - Separated routes, controllers, middleware, and config/db
- Meaningful names
  - Functions like register, login, getme, and signtoken are description
- Error handling
  - Always send clear status codes and JSON error messages
- Comments
  - Added short doc comments above key functions and tests
- Environment configuration
  - Database path and JWT secret come from .env and are not hard coded
- Consistency

- Code has followed same format across the different sprints


Browser/Tiger Tix system
User clicks, submit forms
Frontend
Register, login, logout
User Authentication Service port 4000
Create or edit events
View Events and buy tickets
Natural Language Booking Process
Admin-Service port 5001
Client Service Port 6001
LLM Booking port 7001
Store/Retrieve Data
Update Ticket Counts
DataBase Queries
Auth Database
Return Records
Shared DB