

## Sprint 3 Report

**Team Name:** Schizos Resurrection

**Team Members:** Rodrigo Villalobos, Ronak Bhattal, and David Misuk

---

## Testing Strategy

Sprint 3 introduced the **login-service** microservice with full user authentication (registration, login, JWT-based session handling). Testing was done at multiple levels.

### Manual Testing

- **Backend (login-service):**
  - Registration: verified uniqueness of username and email, password hashing using [bcrypt](#).
  - Login: tested both email and username as identifiers.
  - JWT token: validated creation, expiration (30 minutes), and protected route access ([/me](#)).
  - Logout: ensured tokens cleared and unauthorized access rejected.
- **Frontend (React):**
  - Registration and login forms correctly submitted data.
  - Error messages displayed for missing fields, duplicate usernames/emails, and invalid credentials.
  - Token handling: stored securely in local storage, auto-login on refresh, redirect to login if expired.
  - Logout functionality resets user state and removes stored token.

### Automated End-to-End Testing (Selenium)

- Simulated full user workflows through the UI:
    - [Open login page](#) -> submit registration -> switch to login.
    - [Login with valid credentials](#) -> verify [/me](#) endpoint access.
    - Attempt access to protected routes without token -> confirm access denied.
    - Logout -> confirm user redirected to login and token removed.
  - Verified backend token validation and front-end state updates simultaneously.
- 

## Final Assessment of Testing Coverage

- All core functionalities of **login-service** were tested:
  - Registration, login, JWT token generation, and expiration handling.
  - Frontend form validation and proper error messages.
  - Logout and session reset.
- Selenium tests confirm end-to-end coverage of user authentication flows.  
No known defects remain in authentication functionality.

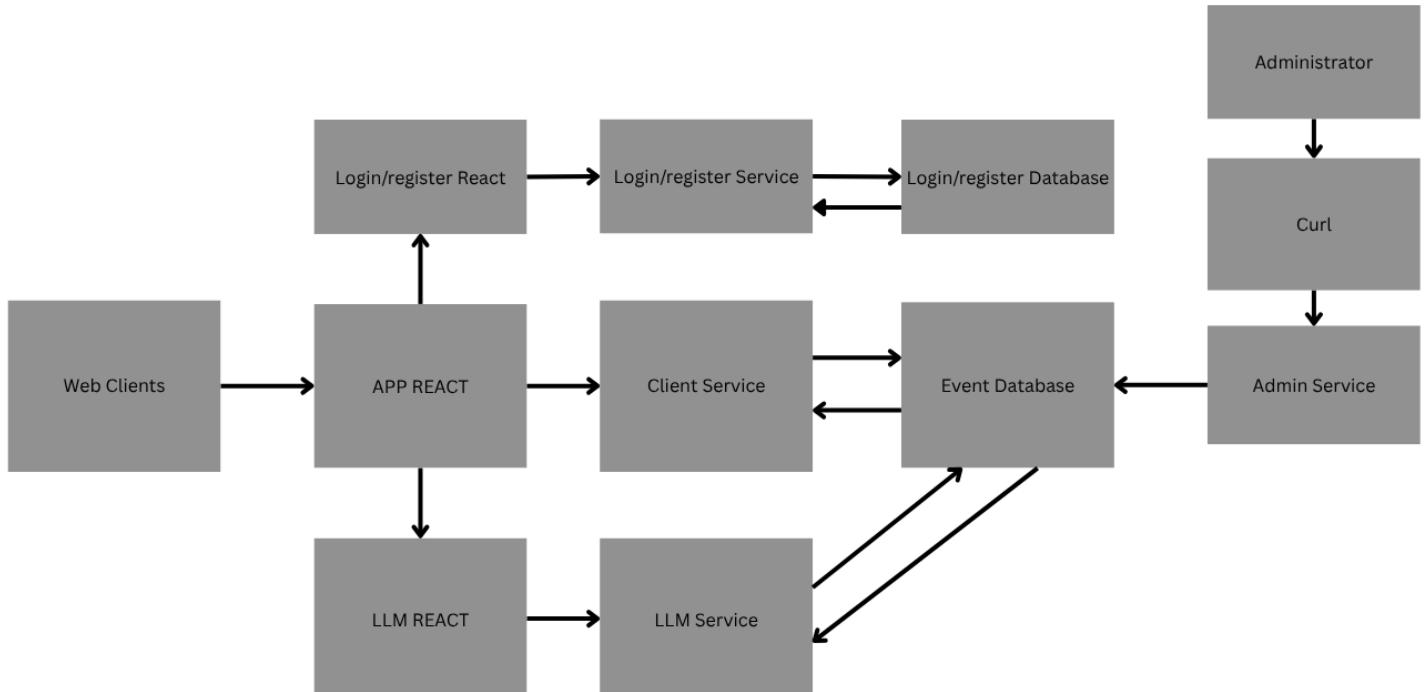
## Architecture

- **React Frontend:** handles login and registration UI, communicates with login-service via REST API.
- **login-service microservice:**
  - Endpoints: [/api/register](#), [/api/login](#), [/api/me](#).
  - Own SQLite database: [login.sqlite](#) (user credentials stored securely with hashed passwords).
  - Issues JWTs for authenticated sessions.
- **Client microservice:** consumes JWTs from login-service for protected actions (ticket purchase)
- **Databases:**
  - [login.sqlite](#) -> authentication data (users).
  - [events.sqlite](#) (or main database) -> events and ticket counts (used by Client microservice).

Communication flow:

[React Frontend <-> login-service \(login.sqlite\)](#)

[React Frontend <-> Client microservice \(events database, JWT-protected\)](#)



---

## Accessibility Features

- Registration and login forms include descriptive **ARIA labels** for screen readers.
  - Error messages are visible and accessible via screen reader.
  - Headings and form controls structured semantically to avoid misreading (e.g., hidden headings via `aria-hidden`).
  - Keyboard navigation fully supported for login and registration workflows.
- 

## Code Quality and Standards

- **Function Documentation:** JSDoc headers added for all controller, model, and utility functions.
  - **Variable Naming & Readability:** consistent and descriptive naming conventions; lines <100 characters where possible.
  - **Modularization:** separation of concerns:
    - `/models/loginModel.js` -> database queries
    - `/controllers/loginController.js` -> registration, login, JWT middleware
    - `/utils/hash.js` -> password hashing and comparison
    - `/routes` -> route definitions
  - **Error Handling:**
    - Backend: try/catch with meaningful messages for registration, login, and token verification.
    - Frontend: fetch and form submission errors properly displayed to the user.
  - **Consistent Formatting:** standardized indentation, spacing, and logical grouping of functions.
  - **Input/Output Validation:** backend ensures required fields, uniqueness, and hashed passwords; frontend validates required inputs before submission.
- 

## Security Rationale

- **Password Hashing:**
  - In `loginController.js`, the password is hashed using `bcryptjs` before storage (`hashPassword(password)`) to prevent plaintext password exposure.
  - Even if the database is compromised, hashed passwords cannot be trivially reversed, protecting user accounts.
- **Token-Based Authentication:**
  - JWTs are generated on login (`jwt.sign({ id, username, email }, JWT_SECRET, { expiresIn: '30m' })`) and used for session validation.

- Tokens allow stateless authentication, secure access to protected routes, and enforce session expiration.

### **Remaining Security Considerations:**

- Tokens are verified on every protected request.
- Potential vulnerabilities: tokens stored in localStorage could be susceptible to XSS attacks.

### **Why three types instead of just unit tests**

- Unit tests alone cannot ensure database interactions, route handling, or full front-end integration works correctly.
- Integration tests catch issues between the backend code and database.
- End-to-end tests verify the entire user workflow in a real browser, catching frontend/backend interaction issues that unit or integration tests alone cannot detect.