Sprint 1 Report

Team Name: Schizos Resurrection

Team Members: Rodrigo Villalobos, Ronak Bhattal and David Misyuk

1. Overview

The primary objectives was to split the backend into two functional microservices (Admin and Client), establish a shared SQLite database for consistent event and ticket data, integrate this database with both services, and replace mock frontend data with live event data. Additionally, accessibility features such as ARIA labels were added to ensure TigerTix is usable by all students, including those with some disabilities.

---

2. Sprint Goals(Completed)

Backend Architecture

- **Admin Service (Port 5001):** Handles event creation, management, and ticket updates.
- **Client Service (Port 6001):** Allows users to browse events and purchase tickets.
- Both services communicate with the shared SQLite database via dedicated model files to ensure clean separation of concerns.

Database Integration

- A **SQLite** database was set up to store consistent data about events and tickets.
- The ***Event*** table stores event details such as name, date, tickets available, and location.
- The ***`Ticket`*** table references events through a foreign key, supporting relationships between event listings and ticket sales.

Frontend Updates

- The **React** frontend was updated to fetch live data from the Client Service
- Implemented an interactive interface displaying event information and ticket availability.

Accessibility Enhancements

- Implemented **ARIA** attributes for improved accessibility technology use.

---

3. Sprint Goals(Uncompleted/Extra)

- For future use, the ***createTicket*** function inside adminModel.js was implemented to demonstrate ticket price, ticket type and the event the ticket belongs to.
- For future use, the ***getAnEvent*** function inside clientModel.js was implemented to access an individual event by eventId if necessary

---

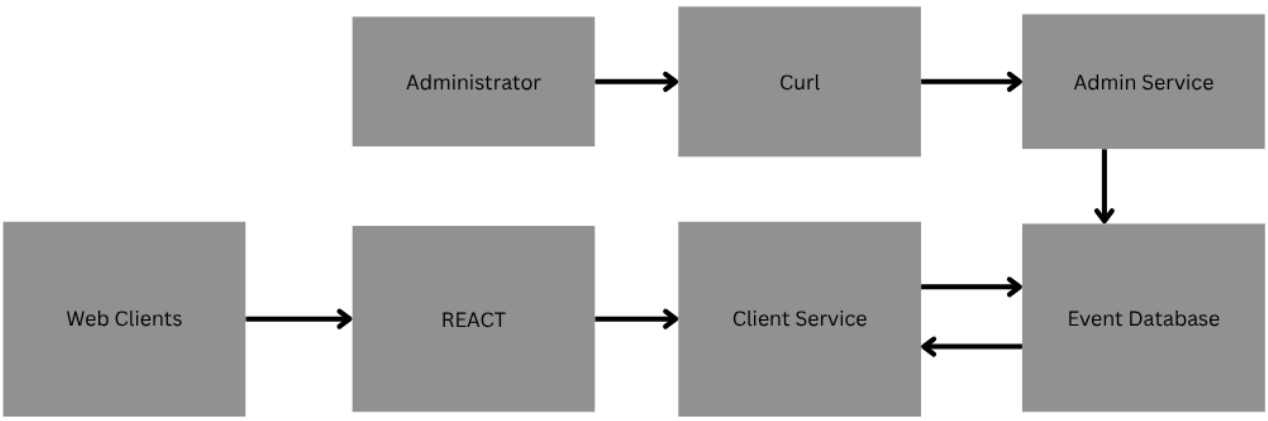
4. Database Synchronization & Concurrency

- To ensure safe concurrent updates and prevent race conditions in the ticketing system, the application logic was designed so that all ticket purchases are executed as atomic operations within the SQLite database. This ensures that a ticket can only be purchased if one is available, preventing the possibility of overselling even when multiple requests are sent simultaneously. The SQLite engine internally handles these updates atomically, guaranteeing database consistency.
- Our testing using cURL and Postman to simulate back-to-back and concurrent purchase requests to the same event confirmed that the ticket count was correctly decremented without allowing negative availability. No race conditions or data corruption were detected, demonstrating that the system is reliable.

---

5. Sprint 1 Architecture

A web client interacts with the react app interface. React talks to the client service which gets the events from the database. If a ticket is purchased in the user interface, react talks to the client which talks to the database to decrement the ticket amount.
An administrator can add an event by accessing the admin service through curl. The admin service then posts event data to the database.



6. Code Quality and Standards

1. **Function Documentation and Comments**
    - In-line comments and functions headers (following JSDoc style) with purpose, inputs and outputs were implemented in each file.
    - File headers were implemented in clientRoute and adminRoute.
2. **Variable Naming and Code Readability**
    - No lines of code are greater than 100, indentation and spacing are consistent and variable names are clear and descriptive.
3. **Modularization**
    - There is no duplicate code, and code is small, reusable and in the MVC pattern.
4. **Error Handling**
    - Try/Catch is used App.js, adminController, clientController.
    - adminController uses input validation.
    - HTTP status codes are used appropriately and error messages are used to help users/developers understand errors. Ex. adminController line 23
5. **Consistent Formatting**
    - Indentation is consistent across all files, blank lines are used to separate functions, and similar statements are grouped together.
6. **Input and Output Handling**
    - Expected input and output is documented above every function where applicable.
    - Database calls and API responses are handled with promises and await /async patterns in admin-service, client-service and in the front end. Ex. adminModel line 28, adminController line 15 and 69.