Sprint 1 Overview

The goal of sprint 1 was to transform the example TigerTix ticketing system into a system with microservices, and integrate it with React on the frontend. The new design separated the website into two backend services, a client and admin service, that share a single SQLite database.

Architecture Design and Flow

The admin microservice is designed to handle event and management operations such as creating, updating and deleting events, and will run on port 5001. The client microservice is designed to read event information and process ticket purchases, and will run on port 6001. Both services connect to the same SQLite database which is found in the backend/shared-db directory.

Each microservice is set up following the MVC pattern:

- Models: Used to define the database schema and connect to SQLite
- Controllers: We added error checking and logic
- Routes: Creates RESTful endpoints for the frontend or admin to get or post data

When the frontend requests data, it communicates only with the client service. The client service queries the database and returns JSON responses such as a list of events or confirmation of a ticket purchase. The admin is isolated for administrative use, improving security by preventing the client from directly modifying any event data.

Example data flow

1. The frontend fetches events from the client service using GET
2. The client service retrieves event data requested from the shares database
3. When a user buys a ticket, the frontend sends a request to client service, and the client service uses a specified purchaseTicket function from the database, which decrements the available ticket count if possible
4. In the same function, it updates the ticket count, and the information is sent back to the frontend and its displayed immediately

Database and Concurrency

The shared SQLite database ensures consistent data between both of the microservices. To maintain the accuracy under multiple concurrent processes, the transactions use a method called serialize(), which ensures atomic operations. This prevents overselling or race conditions if several users attempt to buy the same ticket simultaneously. Each transaction includes proper error handling if an update fails.

Frontend Integration

The frontend was updated to replace hard coded mock data from the original example with live data from the client service. The main App.js now fetches events from `http://localhost:6001/api/events` and displays them dynamically. Each event card shows the event name, data, and available tickets remaining. The "Buy Ticket" button triggers a POST request to the purchase endpoint, and the ticket immediately updates if successful. The

UI code aligns with the code-quality standards such as clear variable naming, consistent formatting, and proper asynchronous I/O handling using async

Accessibility Implementation

To make the system inclusive for visually impaired users, the frontend was upgraded to add ARIA attributes. We added success messages that use aria-live = polite (reads after the user is idle), and aria-live = assertive (reads immediately). Each “Buy Ticket” button also includes an aria-label which describes the exact action. For example, the whole context “Buy ticket for Homecoming game” is pronounced so that the user knows which ticket specifically the cursor is on.

We also added semantic structure. Generic <div> containers were replaced with <main>, <header>, <section>, <ul>, <li>, and <article> elements. This now provides a logical document outline that screen readers can interpret, improving navigation of the system.

Lastly, we improved the focus indicators to make it easier to navigate the interface using only keyboard commands. The active elements are now much more highlighted with different colors, ensuring that it's still easy to use without a mouse.

Code Quality

1. **Function documentation**
    a. Function headers have specific comments that explain what they need to do, and how the function works
2. **Variable naming**
    a. We used descriptive naming (buyTicket, getAllEvents)
3. **Modularization**
    a. By using separate routing, controllers, and models, everything is very modularized
4. **Error handling**
    a. We used try and catch blocks to attempt error handling, and added informative messages/error codes for why some requests are failing
5. **Consistent formatting**
    a. All code is properly indented, and white space is utilized
6. **Input and Output handling**
    a. I/O is handled using async