TigerTix: Sprint 1 Report

1. Introduction

TigerTix is a ticketing system that is designed to simulate real-world event management platforms. It is a web-based ticketing system developed for the students of Clemson University. It brings all the event tickets, football games, concerts, and campus events into a single, simple system.

The primary goals of Sprint 1 were:

- Split a monolithic backend into Admin and Client microservices.
- Implement persistent storage via a shared database.
- Integrate the frontend with live APIs.
- Ensure accessibility for visually impaired users.
- Maintain code quality and proper concurrency control.

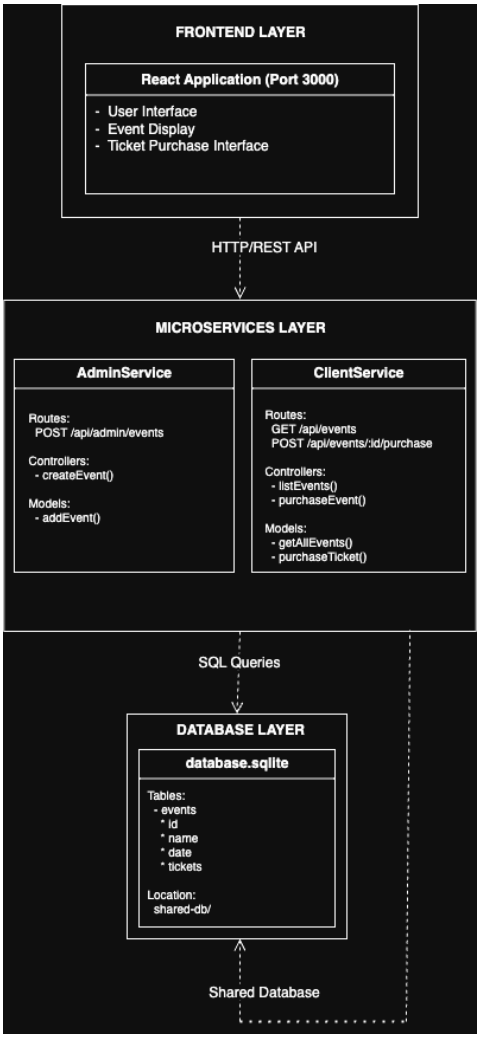
2. System Architecture

Flow Explanation

1. **Frontend → Client Service:**
    - GET /api/events: Fetches and displays events.
    - POST /api/events/:id/purchase: Simulates ticket purchase.
2. **Admin Service → Database:**
    - POST /api/admin/events: Adds new events.
3. **Client Service → Database:**
    - SELECT queries for events.
    - UPDATE queries for ticket purchases (transactional).

This design ensures **separation of concerns**: admins manage data, clients consume it.

Architecture Diagram:



3. Concurrency Handling

Concurrency is managed at the **database and service layers**:

- **Atomic transactions** in clientModel.js (BEGIN, COMMIT, ROLLBACK) ensure consistency.
- **Conditional updates** (WHERE tickets > 0) prevent overselling tickets.
- **Optimistic locking checks** (this.changes) confirm updates were successful.
- **Serialization** with db.serialize() enforces sequential execution, preventing race conditions under simultaneous requests.

4. Accessibility

Accessibility features were implemented in the React frontend to ensure inclusivity:

- **Semantic HTML** (<ul>, <button>, <header>) improves screen reader navigation.
- **ARIA attributes** (e.g., aria-label="Buy Ticket") provide descriptive context.
- **Keyboard navigation:** All interactive elements are accessible via the Tab key.
- **Visible focus indicators:** Users can clearly see which element is selected.
- **Dynamic updates:** Success/failure messages are announced to assistive technologies using aria-live.

5. Code Quality

- ➢ **Separation of Concerns (MVC)**
  - **Models:** Handle database logic (adminModel.js, clientModel.js).
  - **Controllers:** Manage business logic (adminController.js, clientController.js).
  - **Routes:** Define REST endpoints (adminRoutes.js, clientRoutes.js).
- ➢ **Error Handling**
  - **Admin Controller:** Returns 400 for invalid inputs, 500 for server errors.
  - **Client Controller:** Differentiates between event not found (404), sold out (409), and success (200).
  - **Models:** Roll back on transaction errors.
- ➢ **Validation & Data Integrity**
  - Input validation ensures required fields are present (adminController.js, lines 5–8).
  - Event ID validation using parseInt prevents invalid queries.
  - Business rules (tickets > 0) enforced at DB level.
- ➢ **RESTful API Design**
  - GET /api/events → read operations.
  - POST /api/events/:id/purchase → action on resource.
  - Consistent JSON responses across endpoints.
- ➢ **Documentation & Comments**
  - JSDoc used for function headers (e.g., getAllEvents(), purchaseTicket()).
  - Inline comments explain transaction logic and concurrency safeguards.
  - Route files include endpoint purpose descriptions.