1. Code Quality and Structure

1.1 Function Documentation & Comments

Explanation: All functions have headers describing purpose, parameters, outputs, and side effects. Inline comments explain why a particular logic exists (not what it does). Some files also include file headers.

Examples:

- clientController.js (lines 3-30):
    - buyTicket function header documents input (eventId), output (JSON response), and side effect (decrements ticket count).
- App.js (lines 19-24):
    - buyTicket function header describes parameters, return, and state updates.

1.2 Variable Naming & Code Readability

Descriptive, meaningful names are used for variables and functions. Consistent camelCase is used for functions and variables, PascalCase for components. Lines kept reasonably short.

Examples:

- purchaseTicket in clientModel.js (lines 30-45) describes its purpose.

1.3 Modularization

Code is split into small, reusable functions, separated by layer: models for DB, controllers for business logic, routes for API endpoints. No duplicated code; helpers used where needed.

Example:

- clientController.js handles business logic, clientModel.js handles DB.
- clientRoutes.js delegates to controller functions (lines 10–14).

1. 4 Error Handling

Inputs validated, API responses use proper HTTP status codes, clear error messages returned, and errors handled gracefully.

Example:

- clientController.js (lines 37-43): returns 400 if tickets sold out.
- adminController.js (lines 29-34): returns 400 if event info incomplete and 500 if DB insert fails

1.5 Consistent Formatting
Code uses 2-space indentation, logical blocks separated with blank lines, grouped statements. ESLint used in frontend for consistent style.

Example:

- frontend/src/App.js (lines 1–60) follows consistent formatting for React functional component.
- clientModel.js (lines 3–45) uses consistent spacing and line breaks between DB operations.

1.6 Input and Output Handling
APIs expect JSON with defined properties; responses include JSON objects or status codes. All DB calls handled asynchronously with callbacks.

Example:

- clientController.js (lines 10–25): buyTicket receives eventId param and responds with JSON object indicating success/failure.
- App.js (lines 15–50): fetches events from API and updates state asynchronously.

2. Accessibility

The frontend for TigerTix (app.js) follows the rubrics best accessibility practices.
Semantic HTML: app.js uses the following Semantic HTML: <h1>, <h2>, <ul>, <li> and <button> for clear structure.
ARIA Attributes: Implemented aria-label which allows screen readers to read the event name, and aria-live=”polite” which ensures dynamic ticket updates

3. Architecture Design and Explanation

Frontend to Client Service:

1. Fetches list of events from Client Service using GET /api/events.
2. When a ticket is bought, the front end sends a POST request to the Client Service

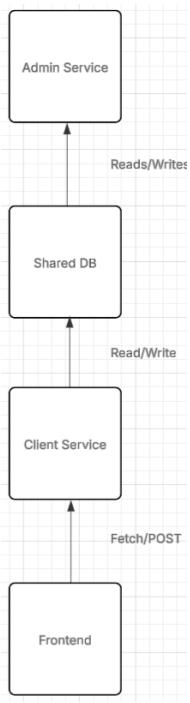
Client Service to shared DB:

1. The Client Service reads from the shared DB to get the current events, and when a ticket is purchased it will decrement the ticket count in the database and reflect that sale.

Admin Services to shared DB:

1. The admin Service manages the events table in the DB, adds new events, updates info and deletes events
2. Changes made by the Admin Service are reflected in the shared DB.

Diagram:



4. Concurrency Handling

To ensure database integrity when multiple users attempt to purchase tickets simultaneously, the client microservice implements basic concurrency control using SQLite's built-in locking mechanism.

Implementation Details:

All ticket purchases occur inside a database transaction.

The transaction:

Read the current ticket count for the event.
Verifies that tickets are available (tickets > 0).
Decrements the count by 1.
Commits the change atomically.