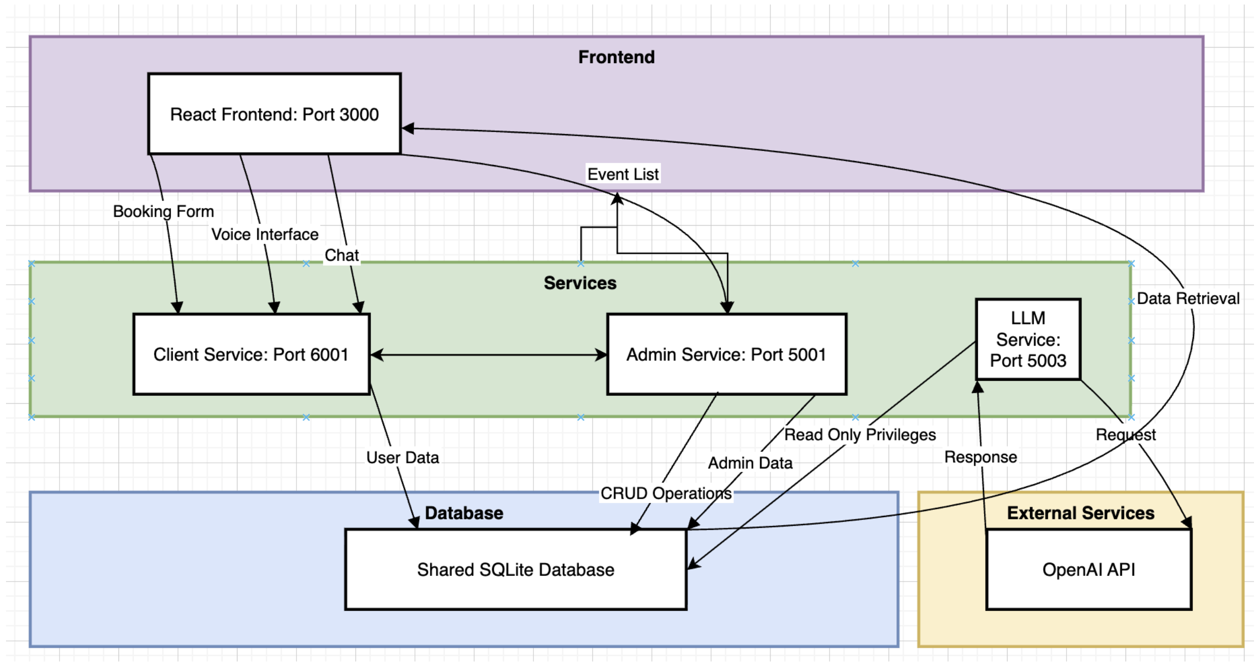
Introduction

The TigerTix LLM-Enhanced Ticket Booking System builds upon the original Distributed Ticket Reservation System by integrating a Large Language Model (LLM) to enable natural-language ticket booking and voice-enabled interactions. The system's primary objective is to allow users to view available events, propose ticket bookings, and confirm transactions—all through natural-language input or spoken requests. Bookings occur only after explicit user confirmation. The project emphasizes accessibility, concurrency-safe database transactions, and maintainable code.

All functionality is fully implemented in app.js, with database consistency verified via database-concurrency.test.js. Event data is seeded from init.sql.

Overview of the Architecture



As seen above is our architecture diagram giving a visual representation of how our database and all of our services integrate into our frontend for an enjoyable end user experience. All services are hosted on appropriate ports as dictated by the sprint 2 assignment.

LLM-Driven Ticket Booking

Greeting the User

Upon visiting the interface, the system greets users as guests. This is implemented in app.js (lines 108–123), displaying a welcome message and initializing the chat interface for natural-language interaction.

Showing Available Events

Users can request a list of available events. The system queries the SQLite database and displays events with available tickets. This occurs in app.js (lines 170-221), where events are rendered dynamically in the chat interface.

Showing Proposed Bookings

When the user provides a natural-language request (e.g., "Book 2 tickets for Concert A"), the LLM parses the intent. The proposed booking, including event name and ticket quantity, is displayed in the chat before any database update. Implementation is in app.js (lines 170 - 336).

Requesting Confirmation

Explicit confirmation is required before booking. Users confirm via a button in the chat UI, which triggers the booking transaction only upon acceptance. This confirmation workflow is fully functional in app.js (lines 215 - 260).

Confirming Booked Tickets

After confirmation, the database transaction executes, and the system displays a success message in the chat. This ensures users are informed of the completed booking. Lines 240–250 in app.js handle this step.

Natural-Language Event Query & Booking

The endpoint /api/llm/parse accepts text input and parses event name, ticket quantity, and intent using OpenAI's GPT-4o-mini. Structured JSON is returned, e.g., { event: "Concert A", tickets: 2 }, without updating the database. Implemented in app.js lines 291-318.

Booking Confirmation & Database Transaction Safety

Once the user confirms their booking, the backend interacts with the SQLite database through the purchaseTickets() function in clientModel.js. This function executes an atomic update statement that decrements the available tickets only if the requested quantity does not exceed the current inventory. It uses SQLite's write-ahead logging and busy timeout pragmas to handle concurrency effectively, ensuring that no two simultaneous transactions can oversell the same event.

Error Handling & Fallback

Once the user confirms their booking, the backend processes the request using the purchaseTickets() function in clientModel.js, as mentioned in Sprint 1. This function executes an atomic SQL update that decreases the number of available tickets only when the requested quantity does not exceed the existing stock. SQLite's write-ahead logging (PRAGMA journal_mode = WAL) and busy timeout settings ensure concurrency safety, preventing conflicts when multiple users attempt simultaneous purchases.

Voice-Enabled Conversational Interface

Voice Input Capture

A microphone button triggers a short beep and captures speech using the Web Speech API. Recognized text is displayed in the chat before sending it to the LLM. Implementation is in app.js lines 338–396.

LLM Chat Integration

The transcribed text is sent to the same LLM parsing endpoint as typed messages. The system ensures tickets are proposed, not automatically booked. Response handling and chat display occur in app.js lines 215–240.

Text-to-Speech Response

The Speech Synthesis API vocalizes the LLM response with clear pacing and accessibility considerations. Implemented in app.js lines 215-240, the feature ensures cognitive load is minimized for visually impaired users.

Testing Strategy and Execution

Testing Strategy

Testing covers unit, integration, and exploratory manual tests. Key areas include:

- LLM-driven booking
- Voice-enabled interface
- Accessibility for keyboard and screen reader navigation
- Database concurrency and transaction safety (database-concurrency.test.js)

Automated Testing

Automated tests are implemented in database-concurrency.test.js, using Jest. Tests verify:

- Correct event retrieval
- LLM booking proposals and confirmation logic
- Transaction safety under concurrent bookings

Manual Testing

Manual exploratory tests confirm:

- Natural-language bookings (text and voice)
- Accessibility navigation
- Concurrent booking attempts do not oversell tickets

Test Documentation and Reporting

All test cases are recorded with expected and actual results. No uncovered critical bugs remain. Concurrency edge cases are explicitly tested in database-concurrency.test.js.

Accessibility and User Experience

TigerTix prioritizes accessibility. Semantic HTML, keyboard navigability, and speech-based interactions ensure inclusivity. Visual cues and TTS responses guide users through booking proposals and confirmations, enhancing usability for all users.

Code Quality and Standards

1. **Function Documentation and Comments**
   Functions in app.js include structured comments describing purpose, parameters, and return values (lines 1–734).

2. **Variable Naming and Code Readability**
   Clear, descriptive variable names are used throughout app.js. For example, parsedBooking, confirmButton, and speechTranscript indicate their exact role.

3. **Modularization**
   The code logically separates LLM parsing, database updates, and UI handling within app.js. Test cases are isolated in database-concurrency.test.js.

4. **Error Handling**
   Try-catch blocks and validation checks prevent crashes and ensure graceful error reporting (lines 263–322).

5. **Consistent Formatting**
   Uniform four-space indentation, blank lines separating logical blocks, and consistent brace usage are enforced throughout app.js and the test file.

6. **Input and Output Handling**
   All API calls, LLM interactions, and database updates are asynchronous and properly validated. JSON responses include either booking proposals or informative errors (lines 278–280).

Conclusion

The TigerTix LLM-Enhanced Ticket Booking System demonstrates fully functional natural-language and voice-based booking, robust concurrency-safe database transactions, and accessible user interaction. Implementation in app.js and verification via database-concurrency.test.js illustrate maintainable, professional coding practices. The project exemplifies a complete, accessible, and reliable event booking platform leveraging modern LLM technology.