Sprint 4 - 3720 TigerTix

Deployment
https://3720-tigertix.vercel.app/

Github:
https://github.com/ethanmarq/3720_tigertix.git

README Copy Paste:

TigerTix - Campus Event Booking System

Project Overview

TigerTix is a microservices-based event booking platform designed for university campus events. It features a unified web interface where users can view events, register/login, and book tickets either through standard UI controls or via an AI-powered conversational assistant.

The system is architected as a Unified Gateway backend that routes traffic to four distinct microservices: Admin Service (Event Management), Client Service (Event booking), Auth Service (User registration), LLM Service (AI Chatbot for booking).

Tech Stack:

- Frontend: React.js, CSS Modules
- Backend: Node.js, Express.js (Gateway & Microservices)
- Database: SQLite (Shared file-based database)
- AI/LLM: Ollama (Llama 3.1) or Mock Fallback for deployment
- Testing: Jest (Backend Unit), Playwright (Frontend E2E)
- DevOps: GitHub Actions (CI/CD Pipeline)

Architecture Summary

The system uses a Unified Gateway pattern. A single Express server (backend/server.js) runs on port 10000 and routes requests to specific internal microservices based on the URL path. `/auth/*` is User Authentication Services, `/api/admin/*` is Admin Service, `/api/events` is Client Service, `/api/llm/*/` is LLM Service. The Frontend runs on Port 3000. All services share a single database.sqlite file location in `backend/shared-db` to ensure data consistency across the distributed services.

Installation and Setup Instructions

Required: Node.js (v18 or higher), npm (Node Package Manager), Git

1. Clone the repo

```
git clone https://github.com/ethanmarq/3720_tigertix.git
cd 3720_tigertix
```

2. Backend Setup The backend uses native modules (sqlite3) which must be compiled for your specific machine architecture (Apple Silicon vs Intel vs Windows).

```
cd backend
npm install
npm run install-all
```

If you encounter `ERR_DLOPEN_FAILED` or wrong architecture errors when starting the server, you must clean and rebuild the dependencies for your specific chip. Run this command from the backend/ folder:

```
rm -rf node_modules package-lock.json
npm install

cd admin-service && rm -rf node_modules package-lock.json && npm install && cd ..
cd client-service && rm -rf node_modules package-lock.json && npm install && cd ..
cd user-authentication && rm -rf node_modules package-lock.json && npm install && cd ..
cd llm-service && rm -rf node_modules package-lock.json && npm install && cd ..
```

3. Database Initialization Before running the app, initialize the shared database

```
node backend/shared-db/setup/js
```

4. Frontend Setup

```
cd Frontend
npm install
```

Environment Variables Setup

Frontend (frontend/.env) Create a file named .env inside the frontend directory:

```
REACT_APP_API_URL=http://localhost:10000
```

Backend No specific .env file is requried for local deployment, but you can configure ports if needed in `backend/server.js`

Running the Application

1. Start the Unified Backend

cd backend
npm start

2. Start the Frontend

cd frontend
npm start

Adding events

curl -X POST http://localhost:10000/api/admin/events \
-H "Content-Type: application/json" \
-d '{"name": "TigerTix Launch Party", "date": "2025-10-20", "tickets": 150}'

How to run regression tests

Backend Unit Tests (Jest)

Run all backend tests
cd backend/admin-service && npm test
cd ../user-authentication && npm test

Frontend E2E Tests (Playwright)

Run from project root
npx playwright test

Team

- Ethan Marquez
- Michael Ellis

Instructor: Dr. Brinkley

License

Importance of CI/CD

Continuous Integration and Continuous Deployment (CI/CD) is very important for software development because it automates the integration and checking of code changes. By

automatically running tests (Jest/Playwright) on every push, we ensure that new code does not break existing functionality (regression testing), maintaining high software quality. Automated deployment removes the manual steps of moving code to production servers, allowing for faster, more reliable release cycles and immediate feedback on build failures.

Learning Reflection

Throughout this project, we learned to think like software engineers and to understand broader infrastructure concepts. We learned how to decompose a monolithic application into distinct services (Auth, Admin, Client, LLM) behind a unified API Gateway, which improved modularity but increased routing complexity. On the infrastructure side, we built an understanding of Network Security, specifically regarding Cross-Origin Resource Sharing (CORS) policies and how browsers block requests between different domains unless explicitly allow-listed headers are present. We also advanced our DevOps skills by configuring GitHub Actions for automated testing and managing complex dependencies across a monorepo structure using npm ci. We also encountered the practical challenges of containerization and dependency management, particularly the limitations of deploying file-based databases like SQLite on cloud hosting and the necessity of rebuilding native modules like sqlite3 when migrating between different chip architectures.

Deployment Challenges

One significant problem we faced was a "Network Error" caused by CORS policies when connecting our Vercel frontend to the Render backend. Initially, our backend was hardcoded to only accept requests from localhost:3000, causing the browser to block production requests; we resolved this by dynamically configuring the allowed origins in Express. We also encountered build failures on Render due to corrupted node_modules from nested installations, which required us to write a "clean install" build command that wiped and reinstalled dependencies for every microservice before starting the server.