Event Management System - Technical Implementation Report

Author: Hritik
Date: December 10, 2025
Version: 1.0

---

1. Executive Summary

The Event Management System is a robust full-stack web application designed to facilitate the creation, management, and booking of events. It serves three distinct user roles: **Admins** (platform oversight), **Organizers** (event management), and **Attendees** (discovery and booking).

Key capabilities include:

- **Role-Based Access Control (RBAC)**: secure access for different user types.
- **Dynamic Dashboard**: Visual analytics for admins and organizers.
- **Real-time Booking**: Inventory management for ticket sales.
- **Review System**: Post-event feedback mechanism.

---

2. System Architecture

The application follows a classic **Client-Server** architecture, utilizing a RESTful API for communication between the decoupled frontend and backend.

2.1 Technology Stack

Layer	Technology	Description
Frontend	**React (Vite)**	High-performance UI library with fast build tooling.
	TypeScript	Static typing for enhanced code reliability.
	Tailwind CSS v4	Utility-first CSS framework for styling.
	Shadcn UI	Accessible, reusable component library.
	Recharts	Data visualization for dashboards.
Backend	**Node.js + Express**	Scalable server-side runtime and framework.
	TypeORM	ORM for database interaction (Data Mapper pattern).
	PostgreSQL	Relational database management system.
	JWT	Stateless authentication mechanism.

---

3. Database Design

The database is built on **PostgreSQL** and managed via **TypeORM**. The schema features robust relationships to ensure data integrity.

3.1 Entity Relationship Diagram (ERD) Overview

- **User**: Central entity linking to Events, Bookings, and Reviews.
- **Event**: Owned by a User (Organizer), contains TicketTypes and Reviews.
- **TicketType**: Defines pricing and inventory for an Event.
- **Booking**: Links a User (Attendee) to a TicketType.
- **Review**: Links a User to an Event.

3.2 Schema Definition

User Entity

Field	Type	Description
`id`	UUID	Primary Key.
`username`	String	Unique identifier.
`email`	String	Unique contact email.
`password`	String	BCHRYPT hashed password.
`role`	Enum	`ADMIN`, `ORGANIZER`, `ATTENDEE`.
`isApproved`	Boolean	For approving Organizer accounts.

Event Entity

Field	Type	Description
`id`	UUID	Primary Key.
`title`	String	Event Name.
`organizer`	FK (User)	Relationship to the creator.
`ticketTypes`	OneToMany	Linked ticket categories.
`date`	Date	Event timestamp.

TicketType Entity

Field	Type	Description
`name`	String	e.g., "VIP", "General".
`price`	Decimal	Cost per ticket.
`quantity`	Int	Total inventory.
`event`	FK (Event)	Parent event.

Booking Entity

Field	Type	Description
`status`	Enum	`CONFIRMED`, `CANCELLED`.

`user`	FK (User)	The attendee.
`ticketType`	FK (TicketType)	The specific ticket purchased.

---

4. Backend Implementation

The backend resides in `fullstack-app/server` and exposes a REST API at `/api`.

4.1 API Endpoints

Authentication (`/api/auth`)

- `POST /register` : Create a new user account.
- `POST /login` : Authenticate and receive a JWT.

Events (`/api/events`)

- `GET /` : List all events (Public).
- `GET /:id` : Get detailed event info (Public).
- `POST /` : Create Event (**Organizer/Admin**).
- `PUT /:id` : Update Event (**Organizer/Admin**).
- `DELETE /:id` : Delete Event (**Organizer/Admin**).
- `GET /stats/organizer` : specific analytics for the logged-in organizer.

Bookings (`/api/bookings`)

- Typically includes creation and list endpoints (inferred from schema).

Users (`/api/users`)

- User profile management.

4.2 Security Mechanisms

- **Stateless Auth**: Validated via `auth.middleware.ts` which extracts the token from the `Authorization: Bearer <token>` header.
- **Authorization**: Middleware guards (e.g., `authorize([UserRole.ADMIN])`) ensure only specific roles can access sensitive endpoints.
- **Validation**: Input validation using `class-validator` (implied by typical TypeORM setups/package.json).

---

5. Frontend Implementation

The frontend is a Single Page Application (SPA) located in `fullstack-app/client`.

5.1 Project Structure

- `src/pages` : Main views corresponding to routes or features.
- `src/components` : Reusable UI elements (Buttons, Inputs, Charts).
- `src/context` : Global state providers.
- `src/lib` : Utilities and API clients.

5.2 Routing System (`App.tsx`)

The app uses `react-router-dom` with a `ProtectedRoute` wrapper to handle access control.

- **Public**: `/` (Landing), `/login`, `/register`, `/events`.
- **Admin Only**: `/admin`.
- **Organizer Only**: `/organizer`.
- **Attendee Only**: `/my-bookings`.

5.3 State Management

- **AuthContext**: Manages the user's login state and holds the `user` object (including role) and `token`. It re-hydrates state from `localStorage` on page refresh.

5.4 Key Interfaces

- **Organizer Dashboard**: A complex view (`OrganizerDashboard.tsx`) likely displaying event stats, bookings, and revenue charts using `Recharts`.
- **Event Details**: Shows event metadata and allows booking interactions.

5.5 API Integration

- **Axios Instance (`api.ts`)**: accessible via `src/lib/api.ts`.
- **Interceptors**: Automatically attaches the JWT from `localStorage` to every outgoing request, simplifying authenticated calls across the app.

---

6. Setup & Deployment

1. **Database**: Ensure a local PostgreSQL instance is running on port 5432 (default).
2. **Environment Variables**: Create `.env` files in both server and client directories.
3. **Backend Start**:

   ```
   cd server
   npm install
   npm start
   ```

4. **Frontend Start**:

   ```
   cd client
   npm install
   npm run dev
   ```

End of Report