Project Report: IIT Jammu Cultural Fest Management System

Date: November 14, 2025

Project ID: PR14

1. Introduction

1.1. Purpose

This document details the design, architecture, and functionality of the **IIT Jammu Cultural Fest Management System**. The primary goal is to provide a robust, scalable, and efficient full-stack application for managing all core aspects of the cultural festival.

1.2. Scope

The system is a full-stack MERN-stack (MySQL, Express, React, Node.js) application designed to handle two distinct user experiences: a public-facing website and a secure administrative panel.

The system's scope covers:

- **Public Website:** Event browsing, participant registration, participant login, and management of personal event registrations and tickets.
- **Admin Panel:** A secure, role-based system for managing all 14 database tables, including:
  - Event and schedule management
  - Venue and performer logistics
  - Sponsor and budget coordination
  - Management of organizing teams and student members
  - Participant and registration oversight

---

2. System Architecture

The application is built on a modern, decoupled MERN-stack (with MySQL).

- **Backend (Server):** A Node.js and Express.js RESTful API. This server handles all business logic, database queries, and authentication.
- **Frontend (Client):** A dynamic single-page application (SPA) built with React.js (using Vite for tooling). The client communicates with the backend API to send and receive data.
- **Database:** A relational MySQL database, hosted on a cloud service (Railway), to ensure data integrity and persistence.

Data Flow:

All communication follows a standard API flow. The React frontend does not have access to the database. It sends HTTP requests (e.g., GET, POST) to the Express backend, which then validates the request, queries the MySQL database, and returns a JSON response.

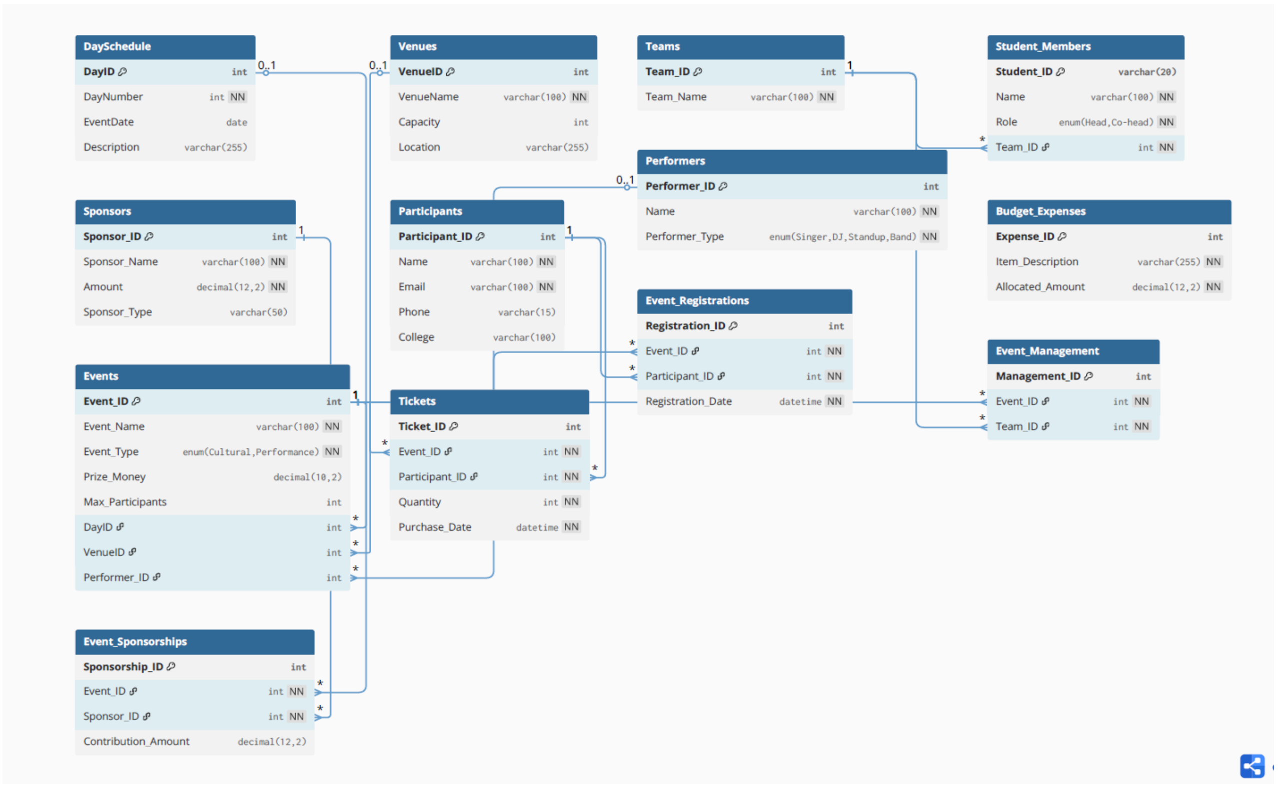
3. Database Design & Normalization

The database design is the foundation of the project. It is centered around the Events table, which is connected to most other entities in the system.

3.1. Final Schema

The initial schema provided in the design document was updated during development to meet advanced functional requirements. The final, implemented schema includes these critical changes:

- **Student_Members Table:**
  - The Role ENUM was expanded from ('Head', 'Co-head') to **('Head', 'Co-head', 'SuperAdmin', 'Member')** to allow for a clearer and more secure permission structure.
  - The Team_ID was made **NULL (optional)**, as a SuperAdmin is not part of any single team.
  - A Password column (VARCHAR(255) NOT NULL) was added to enable admin authentication.
- **Participants Table:**
  - A Password column (VARCHAR(255) NOT NULL) was added to allow for public participant login and authentication.



3.2. Normalization Analysis

The database schema is highly normalized and adheres to the **Boyce-Codd Normal Form (BCNF)**, as detailed in the original design report. This eliminates data redundancy and protects data integrity.

- **1NF (First Normal Form):** All tables are in 1NF. All attributes are atomic (e.g., `varchar`, `int`) and there are no repeating groups or multi-valued attributes.
- **2NF (Second Normal Form):** All tables are in 2NF. All core tables use a single-column primary key (e.g., `Event_ID`, `Participant_ID`), which means they trivially satisfy 2NF as no non-key attribute can be dependent on *part* of a key.
- **3NF (Third Normal Form):** The schema avoids transitive dependencies. For example, the `Events` table stores `VenueID`, not the `VenueName` or `Location`. This prevents a transitive dependency where `Event_ID -> VenueID -> Location`. This pattern is repeated for all foreign key relationships (e.g., `Student_Members` stores `Team_ID`, not `Team_Name`).
- **BCNF (Boyce-Codd Normal Form):** The schema is in BCNF. For every non-trivial dependency X->Y, X is a superkey (a candidate key). For example, in the `Participants` table, both `Participant_ID` and `Email` are candidate keys. All dependencies, like `Participant_ID -> Name` or `Email -> Name`, originate from a superkey, satisfying BCNF .

---

4. Core Functionality & API Flow

This section explains *how* the frontend and backend communicate, addressing the "how we are sending and receiving requests" question.

4.1. Dual Authentication System (JWT)

The application has two independent authentication systems, both using **JSON Web Tokens (JWT)**.

1. **Admin Authentication:**
   - An admin (e.g., `SuperAdmin`) submits their `Student_ID` and `Password` via the `/login` page.
   - This hits the `POST /api/auth/member/login` endpoint.
   - The `authController.js` on the server uses `bcrypt.compare` to check the hashed password.
   - If valid, the server signs a JWT containing the admin's `Student_ID`, `Role`, and `Team_ID` and sends it back.
   - The React client saves this token in `localStorage` via the `AuthContext` as `adminInfo`.
2. **Participant Authentication:**
   - A participant submits their `Email` and `Password` via the `/participant-login` page.
   - This hits the `POST /api/auth/participant/login` endpoint.
   - The `authController.js` validates them against the `Participants` table.
   - If valid, a *separate* JWT is signed (with `Role: 'Participant'`) and sent back.
   - The React client saves this token in `localStorage` as `participantInfo`.

4.2. Authenticated CRUD Request (A Full-Stack Example)

The following is the full lifecycle of a single request, using "Delete a Venue" as an example:

1. **Frontend (React):** A logged-in `SuperAdmin` on the `VenueManagement.jsx` page clicks the "Delete" button.
2. **Frontend (onClick):** The `handleDeleteVenue` function is called. It calls `apiClient.delete('/venues/1')`.
3. **Frontend (`apiClient.js`):** Our custom `apiClient` intercepts this request. It reads the `adminInfo` object from `localStorage`, gets the `token`, and adds it to the request header: `Authorization: Bearer <jwt_token>`.
4. **Backend (Route):** The request arrives at the server and is matched by `server/routes/venueRoutes.js`. The route `router.delete('/:id', protect, isSuperAdmin, deleteVenue)` is triggered.
5. **Backend (Middleware 1: `protect`):** The `protect` middleware runs. It checks the `Authorization` header, verifies the JWT is valid, and decodes it, attaching the user's info (e.g., `{ id: 'superadmin', role: 'SuperAdmin' }`) to the `req.user` object.
6. **Backend (Middleware 2: `isSuperAdmin`):** The `isSuperAdmin` middleware runs. It checks `req.user.role`. Since it is `'SuperAdmin'`, the check passes.
7. **Backend (Controller):** The request is finally passed to the `deleteVenue` function in `server/controllers/venueController.js`.
8. **Backend (Database):** The controller runs the SQL command `DELETE FROM Venues WHERE VenueID = 1`.
9. **Response:** The database confirms the deletion. The server sends back a `200 OK` JSON response (e.g., `{"message": "Venue deleted successfully"}`).
10. **Frontend (Refresh):** The `handleDeleteVenue` function receives the successful response and calls `fetchVenues()` again, which re-downloads the list and updates the UI.

---

5. Project File Structure

This section details "what's there in each file" for the main project directories.

5.1. Backend (`server/`)

The server is a structured Node.js/Express application.

File / Folder	Purpose
`index.js`	The main entry point. It starts the Express server, applies global middleware (like `cors` and `express.json`), and "glues" all the route files to their API prefixes (e.g., `app.use('/api/venues', venueRoutes)`).

`config/db.js`	Configures and exports the `mysql2` connection pool using environment variables from `.env`. This is how we connect to our database.
`controllers/`	Contains the "business logic." Each file (e.g., `venueController.js`, `eventController.js`) contains the functions that run the SQL queries for a specific database table.
`routes/`	Defines the API URLs. Each file (e.g., `venueRoutes.js`) maps a URL and HTTP method (like `GET /`) to a specific controller function (like `getAllVenues`). This is also where we apply our security middleware.
`middleware/`	Contains our "gatekeepers." `authMiddleware.js` has the `protect` function (to check JWTs) and the role-check functions (`isAdmin`, `isSuperAdmin`).
`utils/`	Contains helper functions. `generateToken.js` holds the `jsonwebtoken.sign` logic to create new JWTs.
`.env`	A **private** file that stores secret keys, such as `DB_HOST`, `DB_PASSWORD`, and `JWT_SECRET`.

5.2. Frontend (`client/`)

The client is a structured React application built with Vite.

File / Folder	**Purpose**
`src/main.jsx`	The main entry point for the React app. It renders the `App` component and wraps it in our `<AuthProvider>` (from Context API) and `<BrowserRouter>` (from React Router).

src/App.jsx	The top-level component. Its only job is to define the application's **page routing** using <Routes> and <Route>. It defines which URLs lead to the public site, the admin login, or the protected admin panel.
src/layout/	Contains the "wrapper" components. PublicLayout.jsx has the public navbar. AdminLayout.jsx has the admin sidebar and header. Both use the <Outlet /> component.
src/components/	Contains small, reusable UI pieces. Modal.jsx, Navbar.jsx, ProtectedRoute.jsx, and ParticipantProtectedRoute.jsx are key examples.
src/pages/	Contains the main page components. Each file (e.g., EventManagement.jsx, HomePage.jsx) represents a full page in the application.
src/context/	Manages global application state. AuthContext.jsx provides the admin and participant objects, along with the login and logout functions, to the entire application.
src/api/	Contains apiClient.js. This is our custom axios instance that automatically attaches the admin's auth token (Authorization: Bearer ...) to every request.
tailwind.config.js	Configures the Tailwind CSS utility-first styling framework.