IIT JAMMU CULTURAL FESTIVITIES MANAGEMENT SYSTEM

Submitted to :
Dr. Mrinmoy Bhattacharjee

Submitted By:
Kanika (2023ucs0095)
Isha Singh (2023uma0214)
Japnoor Kaur (2023uma0215)
Saumya Gupta (2023ucs0111)
Vinayak Soni(2023ucs0119)

ABOUT THE PROJECT

The IIT Jammu Cultural Fest Management System is a full-stack application designed to manage all aspects of the annual cultural festival.

It provides two interfaces:

✓ Public portal for participants

✓ Admin panel for organizers

Why This System?

- To automate registrations, sponsorships, event management, budgeting & logistics.
- To ensure data accuracy, security, and scalability.

WHAT THE SYSTEM COVERS

1. **Public Website**: Event browsing, participant registration, participant login, and management of personal event registrations and tickets.
2. **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

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.

RELATIONAL SCHEMA

. GROUP 1: CORE ENTITIES

Table DaySchedule { DayID int [pk, increment] DayNumber int [not null] EventDate date Description varchar(255) }

Table Venues { VenueID int [pk, increment] VenueName varchar(100) [not null, unique] Capacity int Location varchar(255) }

Table Teams { Team_ID int [pk, increment] Team_Name varchar(100) [not null, unique] }

Table Student_Members { Student_ID varchar(20) [pk] Name varchar(100) [not null] Role enum('Head', 'Co-head', “Super Admin”, “member”) [not null] Team_ID int [not null, ref: > Teams.Team_ID] }

Table Sponsors { Sponsor_ID int [pk, increment] Sponsor_Name varchar(100) [not null, unique] Amount decimal(12, 2) [not null] Sponsor_Type varchar(50) }

Table Participants { Participant_ID int [pk, increment] Name varchar(100) [not null] Email varchar(100) [not null, unique] Phone varchar(15) College varchar(100) }

Table Performers { Performer_ID int [pk, increment] Name varchar(100) [not null] Performer_Type enum('Singer', 'DJ', 'Standup', 'Band') [not null] }

Table Budget_Expenses { Expense_ID int [pk, increment] Item_Description varchar(255) [not null] Allocated_Amount decimal(12, 2) [not null] }

RELATIONAL 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.

GROUP 2: THE CENTRAL EVENTS TABLE

Table Events { Event_ID int [pk, increment] Event_Name varchar(100) [not null] Event_Type enum('Cultural', 'Performance') [not null] Prize_Money decimal(10, 2) Max_Participants int DayID int [ref: > DaySchedule.DayID] VenueID int [ref: > Venues.VenueID] Performer_ID int [null, ref: > Performers.Performer_ID] }

. GROUP 3: JUNCTION & LINKING TABLES

Table Tickets { Ticket_ID int [pk, increment] Event_ID int [not null, ref: > Events.Event_ID] Participant_ID int [not null, ref: > Participants.Participant_ID] Quantity int [not null, default: 1] Purchase_Date datetime [not null] }

Table Event_Registrations { Registration_ID int [pk, increment] Event_ID int [not null, ref: > Events.Event_ID] Participant_ID int [not null, ref: > Participants.Participant_ID] Registration_Date datetime [not null] _index [Event_ID, Participant_ID] [unique] }

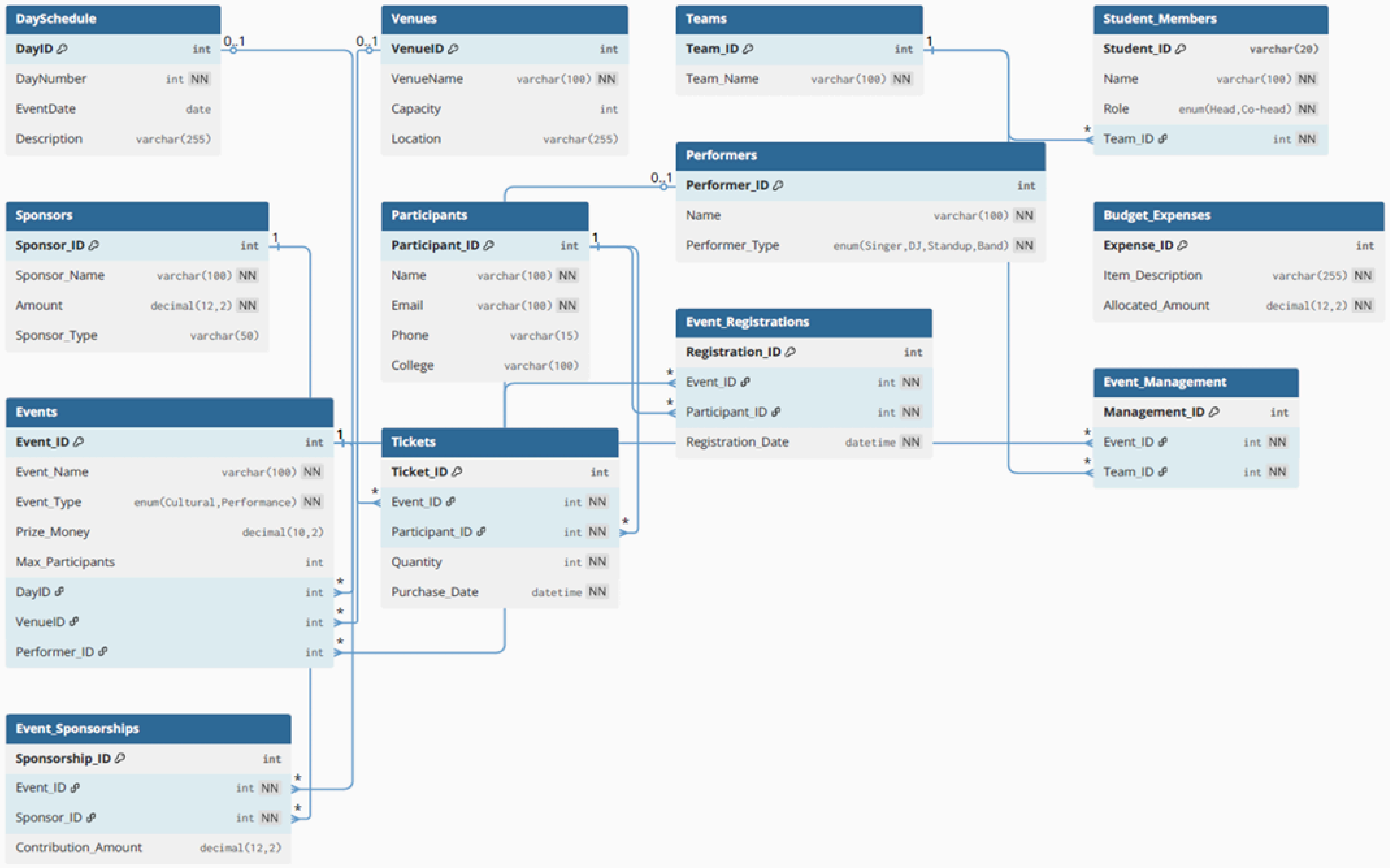
Table Event_Management { Management_ID int [pk, increment] Event_ID int [not null, ref: > Events.Event_ID] Team_ID int [not null, ref: > Teams.Team_ID] _index [Event_ID, Team_ID] [unique] }

Table Event_Sponsorships { Sponsorship_ID int [pk, increment] Event_ID int [not null, ref: > Events.Event_ID] Sponsor_ID int [not null, ref: > Sponsors.Sponsor_ID] Contribution_Amount decimal(12,2) _index [Event_ID, Sponsor_ID] [unique] }

ER DIAGRAM

The design is centered around the Events table, which connects to virtually all other parts of the system.

- Core Entities: These are the primary "nouns" of the system: Events , Participants , Sponsors , Venues , Teams , Performers , and DaySchedule .
- Junction Tables (Many-to-Many): To avoid data redundancy and adhere to normalization principles, several many-to-many (M:N) relationships are resolved using junction tables: ○ Event_Registrations : Links Events and Participants (a participant can register for many events; an event has many participants). ○ Event_Management : Links Events and Teams (a team can manage multiple events; an event can be managed by multiple teams). ○ Event_Sponsorships : Links Events and Sponsors (a sponsor can fund multiple events; an event can have multiple sponsors).
- One-to-Many (1:N) Relationships: These are handled by foreign keys. For example, many Student_Members belong to one Team . One DaySchedule can have many Events .



DaySchedule		
DayID	int	
DayNumber	int	NN
EventDate	date	
Description	varchar(255)	

Venues		
VenueID	int	
VenueName	varchar(100)	NN
Capacity	int	
Location	varchar(255)	

Teams		
Team_ID	int	
Team_Name	varchar(100)	NN

Student_Members		
Student_ID	varchar(20)	
Name	varchar(100)	NN
Role	enum(Head,Co-head)	NN
Team_ID	int	NN

Performers		
Performer_ID	int	
Name	varchar(100)	NN
Performer_Type	enum(Singer,DJ,Standup,Band)	NN

Sponsors		
Sponsor_ID	int	
Sponsor_Name	varchar(100)	NN
Amount	decimal(12,2)	NN
Sponsor_Type	varchar(50)	

Participants		
Participant_ID	int	
Name	varchar(100)	NN
Email	varchar(100)	NN
Phone	varchar(15)	
College	varchar(100)	

Budget_Expenses		
Expense_ID	int	
Item_Description	varchar(255)	NN
Allocated_Amount	decimal(12,2)	NN

Event_Registrations		
Registration_ID	int	
Event_ID	int	NN
Participant_ID	int	NN
Registration_Date	datetime	NN

Event_Management		
Management_ID	int	
Event_ID	int	NN
Team_ID	int	NN

Events		
Event_ID	int	
Event_Name	varchar(100)	NN
Event_Type	enum(Cultural,Performance)	NN
Prize_Money	decimal(10,2)	
Max_Participants	int	
DayID	int	
VenueID	int	
Performer_ID	int	

Tickets		
Ticket_ID	int	
Event_ID	int	NN
Participant_ID	int	NN
Quantity	int	NN
Purchase_Date	datetime	NN

Event_Sponsorships		
Sponsorship_ID	int	
Event_ID	int	NN
Sponsor_ID	int	NN
Contribution_Amount	decimal(12,2)	

NORMALIZATION

First Normal Form (1NF)

Requirement: All attributes must be atomic. Each cell must contain a single, indivisible value, and there must be no repeating groups.

Analysis: All tables in the schema comply with 1NF.

- There are no multi-valued attributes (e.g., a list of phone numbers in a single cell).
- Data types like varchar , int , decimal , date , and enum are inherently atomic.
- M:N relationships (like multiple sponsors for one event) are not stored in a list but are properly resolved using junction tables (Event_Sponsorships).

All tables are in 1NF

NORMALIZATION

Third Normal Form (3NF)

Requirement: The table must be in 1NF, and all non-key attributes must be fully functionally dependent on the entire primary key. There should be no partial dependency in the relation. There must be no transitive dependencies. A transitive dependency exists when a non-key attribute is functionally dependent on another non-key attribute (e.g., A -> B -> C where A is the PK, and B and C are non-key attributes).

Analysis:

● Example (Events & Venues): Consider the Events table. It contains VenueID but not VenueName or Location . The functional dependency is Event_ID -> VenueID . The details of the venue (VenueName , Capacity , Location) are dependent on VenueID , not Event_ID . If Location were stored in the Events table, it would create a transitive dependency (Event_ID -> VenueID -> Location). By moving venue details to a separate Venues table, this dependency is removed and 3NF is achieved.

NORMALIZATION

● Example (Students & Teams): The Student_Members table has Team_ID , but not Team_Name . Student_ID -> Team_ID . The Team_Name is dependent on Team_ID , not Student_ID . This design correctly adheres to 3NF.

Conclusion: All tables are in 3NF.

NORMALIZATION

Boyce-Codd Normal Form (BCNF)

Requirement: A stricter form of 3NF. For every non-trivial functional dependency X -> Y , X must be a superkey (i.e., a candidate key).

Analysis:

BCNF is primarily concerned with tables that have multiple, overlapping candidate keys.

● Example (Participants): The Participants table has Participant_ID (Primary Key) and Email (marked as unique). Both are candidate keys. ○ FD 1: Participant_ID -> (Name, Email, Phone, College) . The determinant, Participant_ID , is a superkey. This is BCNF-compliant. ○ FD 2: Email -> (Participant_ID, Name, Phone, College) . The determinant, Email , is also a superkey (a candidate key). This is BCNF-compliant.

● Example (Venues, Teams, Sponsors): These tables follow the same pattern, with a surrogate ID as the PK and a Name field as a unique candidate key. All functional dependencies in these tables originate from a superkey. Conclusion: All tables in the schema are in BCNF.

FUTURE ENHANCEMENT

While the current design is robust, one area could be revisited if requirements change:

● Performers and Events: The current design uses a nullable foreign key (Performer_ID) in the Events table. This implies a one-to-one (or one-to-zero) relationship : an event can have at most
one performer.

○ Future Requirement: If a single event (e.g., "Battle of the Bands") could feature multiple performers, or if one performer could play at multiple events, this 1:N relationship would become M:N.

○ Solution: In that scenario, the Performer_ID column would be removed from the Events table. A new junction table, Event_Performers (Event_Performer_ID , Event_ID , Performer_ID), would be created. This would be a more flexible design if the "one performer per event" rule is not strict.

CONCLUSION

The proposed database schema for the IIT Jammu Cultural Fest Management System is robust, efficient, and highly normalized, adhering to the Boyce-Codd Normal Form (BCNF). It effectively eliminates data redundancy and protects data integrity through the use of surrogate keys, foreign key constraints, and well-resolved many-to-many relationships. The design is production-ready for the current requirements and provides clear pathways for future scalability.

THANK YOU