Event and Party Management System

School of Mathematical and Computational Sciences
Daniel Troya A.
May 30, 2025

Contents

1 Objectives

- Design and implement a database to manage information related to large-scale events and parties.
- Facilitate data management for attendees, suppliers, staff, locations, and event activities.
- Apply best practices in database design to ensure scalability and performance.

2 Description

Managing large-scale events requires careful planning and resource allocation to ensure smooth operations. The lack of an efficient system can lead to data loss, mismanagement of staff and resources, and logistical failures, especially in backstage coordination. This project aims to develop a database system that optimizes event resource management by organizing attendees, ticketing, staff, suppliers, and venue logistics efficiently. By implementing this system, event organizers can track attendees, assign staff effectively, and manage supplier services, reducing operational risks and improving event execution.

3 Requirements

3.1 Functional Requirements

- The system must allow users to register attendees and assign them unique tickets for specific events.
- Tickets must be validated at the entry points of each venue during the event to verify their authenticity and status (Valid, Expired, or Canceled). Tickets are classified into three types:
  - **General:** Grants access to general venues.
  - **VIP:** Grants access to general and VIP venues.
  - **Premium:** Grants access to general, VIP, and Premium venues.

  NOTE: Each ticket class offers different services depending on the event's schedule.
- Staff members must be assigned to specific venues and events, with clearly defined tasks and roles.
- Suppliers must be linked to the services they provide, including associated staff and scheduled event dates (which may be unique).
- Event organizers should be able to generate detailed reports on attendee participation, staff assignments per venue, and supplier performance across events.

3.2 Non-Functional Requirements

- The system must ensure data integrity by enforcing primary and foreign key constraints, and prevent data duplication.
- The database must support concurrent access and modifications by multiple authorized users.
- The system should be scalable to manage events with thousands of attendees, staff, and suppliers.
- Database operations should be optimized for fast query execution and support real-time updates where applicable.

4 Project Scope

- The database will manage information related to event planning, execution, and analysis.
- Queries will be implemented to retrieve necessary data, such as attendance lists, supplier inventories and activity schedules.
- The database structure will be implemented in a database management system (DBMS) such as MySQL and using Docker for containerization.
- Database operations will be performed through an API developed in Python with SQLAlchemy.
- The databases will feature a graphical user interface (GUI) to facilitate user interaction and management. This GUI will be implemented using Tkinter, a Python library for designing user interfaces.
- The project will not include a financial analysis of the event at this stage, its implementation is planned for future development.

5 Conceptual Model

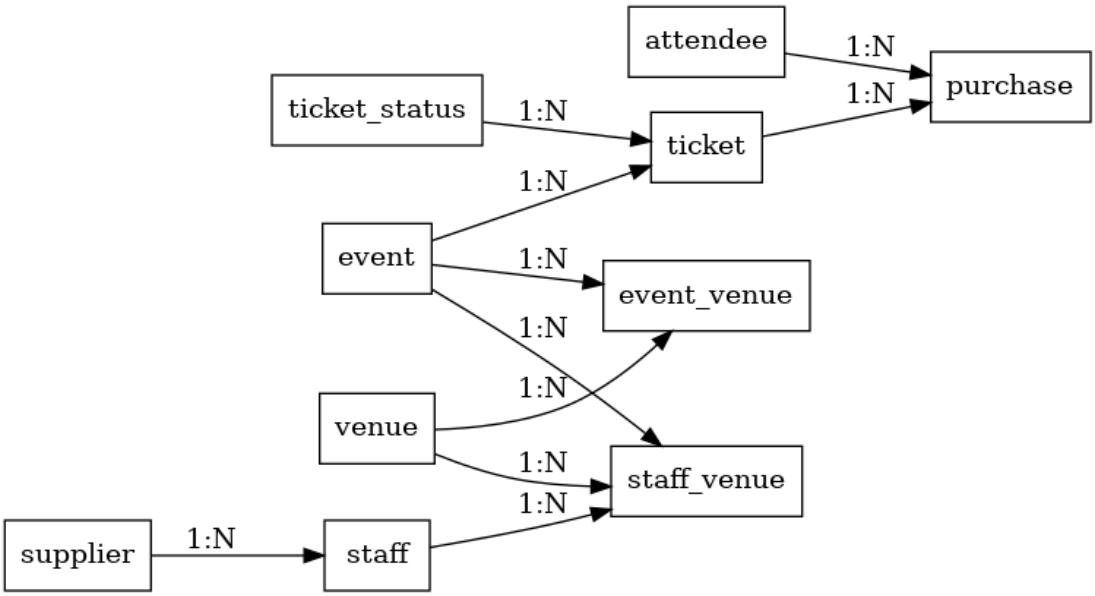

Figure 1: Conceptual Model Diagram

5.1 Relationships and Cardinality

- ***R1:*** An attendee can purchase one or more tickets, and each ticket can only be purchased by one attendee.
  Relationship: Many-to-Many (implemented on Table 10).

- ***R2:*** An event can take place in multiple venues, and a venue can host multiple events. This relationship is valid because each event is uniquely identified by its date.
  Relationship: Many-to-Many (implemented on Table 8).

- ***R3:*** Staff can be assigned to multiple venues across different events, and each event venue can have multiple staff.
  Relationship: Many-to-Many (implemented on Table 9).

- ***R4:*** A ticket is always associated with a single event.
  Relationship: One-to-Many (event to ticket).

- ***R5:*** A staff member is employed by a single supplier, but each supplier can have many staff.
  Relationship: One-to-Many (supplier to staff).

- ***R6:*** A ticket has one status, but each status may be used by multiple tickets.
  Relationship: One-to-Many (ticket status to ticket).

6 Logical Model

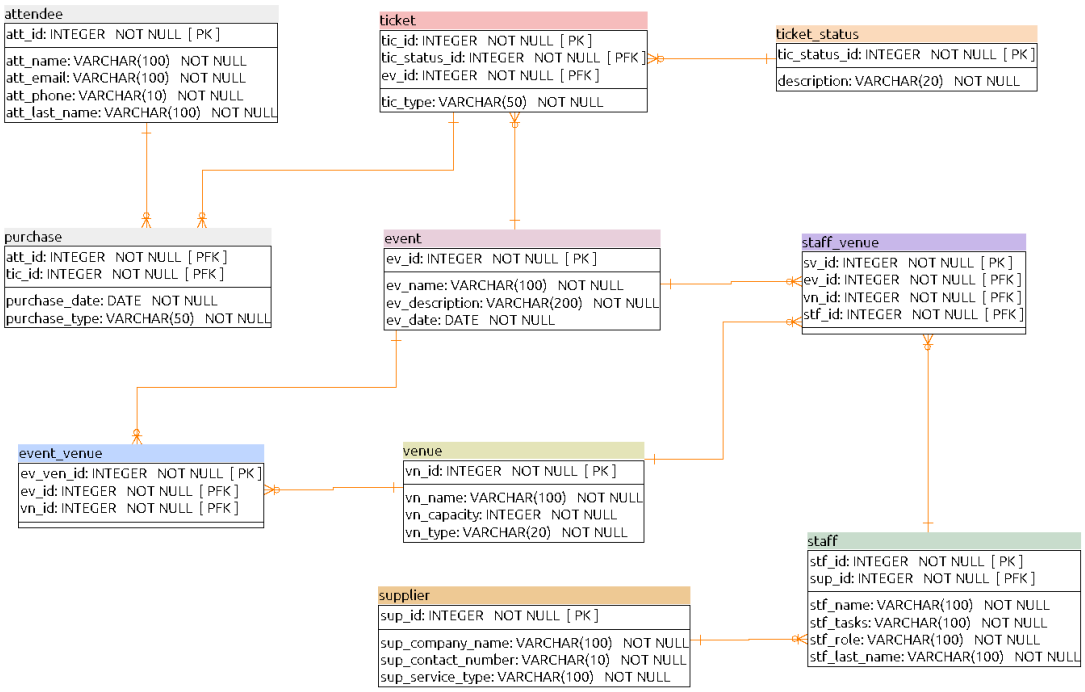

Figure 2: Logical Model Diagram

6.1 Entities and Attributes

- **Attendee:** Individuals attending the event.

Attribute	Attribute on DBS
Attendee ID	*att_id*
First Name	*att_name*
Last Name	*att_last_name*
Email Address	*att_email*
Phone Number	*att_phone*

Table 1: Attributes of the `Attendee`

- **Ticket:** Entries that grant access to an event.

Attribute	**Attribute on DBS**
Ticket ID	*tic_id*
Ticket Type	*tic_type*
Ticket Status ID (foreign key)	*tic_status_id*
Event ID (foreign key)	*ev_id*

Table 2: Attributes of the `Ticket`

- **Ticket Status:** Defines the status of a ticket (valid = 1, expired = 2, canceled = 3).

Attribute	**Attribute on DBS**
Ticket Status ID	*tic_status_id*
Status Description	*description*

Table 3: Attributes of the `Ticket_Status`

- **Staff:** Personnel working at the event.

Attribute	**Attribute on DBS**
Staff ID	*stf_id*
First Name	*stf_name*
Last Name	*stf_last_name*
Assigned Tasks	*stf_tasks*
Role	*stf_role*
Supplier ID (foreign key)	*sup_id*

Table 4: Attributes of the `Staff`

- **Supplier:** Companies providing services.

Attribute	**Attribute on DBS**
Supplier ID	*sup_id*
Company Name	*sup_company_name*
Contact Number	*sup_contact_number*
Service Type	*sup_service_type*

Table 5: Attributes of the `Supplier`

- **Venue:** Physical locations within an event.

Attribute	Attribute on DBS
Venue ID	*vn_id*
Venue Name	*vn_name*
Venue Type	*vn_type*
Capacity	*vn_capacity*

Table 6: Attributes of the `Venue`

- **Event:** Organized occasions with description activities.

Attribute	Attribute on DBS
Event ID	*ev_id*
Event Name	*ev_name*
Description	*ev_description*
Event Date	*ev_date*

Table 7: Attributes of the `Event`

Now, these tables were created to establish many-to-many relationships between the main entities:

- **Event-Venue Assignment:** Links events with venues.

Attribute	Attribute on DBS
Event-Venue Assignment ID	*ev_ven_id*
Event ID (foreign key)	*ev_id*
Venue ID (foreign key)	*vn_id*

Table 8: Attributes of the `Event_Venue_Assignment`

- **Staff-Venue Assignment:** Links staff to a specific event and venue.

Attribute	Attribute on DBS
Staff-Venue Assignment ID	*sv_id*
Event ID (foreign key)	*ev_id*
Staff ID (foreign key)	*stf_id*
Venue ID (foreign key)	*vn_id*

Table 9: Attributes of the `Staff_Venue_Assignment`

- **Purchase:** Represents the ticket buying action by an attendee.

Attribute	Attribute on DBS
Purchase Date	*purchase_date*
Purchase Type	*purchase_type*
Attendee ID (foreign key)	*att_id*
Ticket ID (foreign key)	*tic_id*

Table 10: Attributes of the `Purchase`

7 Physical Model

To generate the physical model, we used the tool `SQL Power Architect`. However, this tool only generates a basic physical structure. To enhance data control and adapt the model to our specific needs within the MySQL container, we manually modified parts of the SQL code.

The following adjustments were made to improve data validation, performance, and consistency:

- We added `ENUM` types to restrict column values to specific options (ticket types, purchase methods), improving data integrity.
- We implemented `CONSTRAINT` and `CHECK` clauses to enforce value rules (e.g., email format, phone number format, positive capacities), reducing input errors.
- We applied the `UNIQUE` constraint on certain fields such as event dates and attendee emails to prevent duplicate entries and maintain consistency.
- We specified `ENGINE=InnoDB` to support transactions and foreign key relationships, essential for maintaining referential integrity.

```
CREATE TABLE ticket_status (
    tic_status_id INT NOT NULL,
    description VARCHAR(20) NOT NULL,
    PRIMARY KEY (tic_status_id)
) ENGINE=InnoDB;

CREATE TABLE event (
    ev_id INT NOT NULL AUTO_INCREMENT,
    ev_name VARCHAR(100) NOT NULL,
    ev_description VARCHAR(200) NOT NULL,
    ev_date DATE NOT NULL UNIQUE,
    PRIMARY KEY (ev_id),
    INDEX idx_ev_date (ev_date)
) ENGINE=InnoDB;

CREATE TABLE venue (
    vn_id INT NOT NULL AUTO_INCREMENT,
```

```
    vn_name VARCHAR(100) NOT NULL,
    vn_type ENUM('VIP', 'General', 'Premium') NOT NULL,
    vn_capacity INT NOT NULL,
    PRIMARY KEY (vn_id),
    CONSTRAINT chk_capacity CHECK (vn_capacity > 0)
) ENGINE=InnoDB;

CREATE TABLE event_venue (
    ev_ven_id INT NOT NULL AUTO_INCREMENT,
    ev_id INT NOT NULL,
    vn_id INT NOT NULL,
    PRIMARY KEY (ev_ven_id),
    FOREIGN KEY (ev_id) REFERENCES event(ev_id) ON DELETE
        CASCADE,
    FOREIGN KEY (vn_id) REFERENCES venue(vn_id) ON DELETE
        CASCADE
) ENGINE=InnoDB;

CREATE TABLE supplier (
    sup_id INT NOT NULL AUTO_INCREMENT,
    sup_company_name VARCHAR(100) NOT NULL,
    sup_contact_number VARCHAR(10) NOT NULL,
    sup_service_type VARCHAR(100) NOT NULL,
    PRIMARY KEY (sup_id),
    INDEX idx_service_type (sup_service_type),
    CONSTRAINT chk_contact_number CHECK (sup_contact_number
        REGEXP '^09[0-9]{8}$')
) ENGINE=InnoDB;

CREATE TABLE staff (
    stf_id INT NOT NULL AUTO_INCREMENT,
    stf_name VARCHAR(100) NOT NULL,
    stf_last_name VARCHAR(100) NOT NULL,
    stf_tasks VARCHAR(100) NOT NULL,
    stf_role VARCHAR(100) NOT NULL,
    sup_id INT NOT NULL,
    PRIMARY KEY (stf_id),
    FOREIGN KEY (sup_id) REFERENCES supplier(sup_id) ON
        DELETE CASCADE
) ENGINE=InnoDB;

CREATE TABLE staff_venue (
    sv_id INT NOT NULL AUTO_INCREMENT,
    ev_id INT NOT NULL,
    stf_id INT NOT NULL,
    vn_id INT NOT NULL,
    PRIMARY KEY (sv_id),
    FOREIGN KEY (stf_id) REFERENCES staff(stf_id) ON DELETE
        CASCADE,
    FOREIGN KEY (vn_id) REFERENCES venue(vn_id) ON DELETE
```

```
        CASCADE,
    FOREIGN KEY (ev_id) REFERENCES event(ev_id) ON DELETE
        CASCADE,
    UNIQUE KEY unique_assignment (ev_id, stf_id, vn_id)
) ENGINE=InnoDB;

CREATE TABLE attendee (
    att_id INT NOT NULL AUTO_INCREMENT,
    att_name VARCHAR(100) NOT NULL,
    att_last_name VARCHAR(100) NOT NULL,
    att_email VARCHAR(100) NOT NULL,
    att_phone VARCHAR(10) NOT NULL,
    PRIMARY KEY (att_id),
    UNIQUE INDEX idx_email (att_email),
    CONSTRAINT chk_correo_valido
        CHECK (att_email REGEXP '^[A-Za-z0-9._%-]+@[A-Za-z0
            -9.-]+\\.[A-Za-z]{2,}$'),
    CONSTRAINT chk_phone
        CHECK (att_phone REGEXP '^09[0-9]{8}$')
) ENGINE=InnoDB;

CREATE TABLE ticket (
    tic_id INT NOT NULL AUTO_INCREMENT,
    tic_type ENUM('VIP', 'General', 'Premium') NOT NULL,
    tic_status_id INT NOT NULL,
    ev_id INT NOT NULL,
    PRIMARY KEY (tic_id),
    FOREIGN KEY (tic_status_id) REFERENCES ticket_status(
        tic_status_id) ON UPDATE CASCADE,
    FOREIGN KEY (ev_id) REFERENCES event(ev_id) ON DELETE
        CASCADE,
    INDEX idx_ticket_status (tic_status_id)
) ENGINE=InnoDB;

CREATE TABLE purchase (
    purchase_date DATE NOT NULL,
    purchase_type ENUM('Online', 'Mobile App', 'Box Office')
         NOT NULL,
    att_id INT NOT NULL,
    tic_id INT NOT NULL,
    PRIMARY KEY (att_id, tic_id),
    FOREIGN KEY (att_id) REFERENCES attendee(att_id) ON
        DELETE CASCADE,
    FOREIGN KEY (tic_id) REFERENCES ticket(tic_id) ON DELETE
         CASCADE
) ENGINE=InnoDB;
```

Listing 1: Physical Model

8 Database Container

Once finished the code for creating the database was developed, we created a **MySQL** container using Docker, where the database is initialized and deployed. Inside the `Container` folder, we include the necessary files and documentation to build and initialize the containerized database.

- **Dockerfile:** Used to build the Docker container with MySQL.
- **init.sql:** Contains the complete SQL schema of the database, including table definitions and sample `INSERT` statements.

9 API Design

9.1 Flask and Flask-SQLAlchemy

We used the **Flask** framework alongside **Flask-SQLAlchemy**, a Python library that facilitates the creation of CRUD (Create, Read, Update, Delete) operations in a clean and efficient manner. To complement these tools, we also used **Marshmallow** for data serialization/validation and **SQLAlchemy** as the ORM (Object Relational Mapping) to communicate with the MySQL container.

9.2 Marshmallow

The API layer allows communication between the user interface and the MySQL database. Using Marshmallow, we defined schemas that help with automatic data validation and transformation between JSON and Python objects.

To maintain a modular and scalable structure, each table in the database has its own Python script implementing its respective CRUD operations. These scripts are organized using Flask `Blueprints`, which allow us to separate logic by module and import them all into a single entry point:

- `use_db.py:` This script centralizes the SQLAlchemy `db` instance, avoiding redundancy in each module.
- `main.py:` The main application file where all Blueprints are registered. It also defines the host and port `3306` for connecting to the MySQL container.

For more information, the folder `flask_api` contains:

- **flaskenv:** Environment configuration file containing all required libraries.
- **Python scripts (.py):** All modular CRUD APIs developed for each database table.

10 Graphical User Interface

In this section, we explain how the graphical user interface (GUI) was created. To explore the code, open the folder: `GUI`. This folder contains all the Python scripts related to the GUI, which are:

- event_gui.py
- event_venue_gui.py
- main_GUI.py
- staff_gui.py
- supplier_gui.py
- ticket_gui.py
- URL.py
- venue_gui.py

Each of these files is similar in structure, as they are modeled based on the features of each corresponding database table. However, two important scripts must be highlighted:

- **main_GUI.py**: This script integrates all the individual GUI modules corresponding to the database tables. It also manages the connection between the GUI and the containerized API using Flask.
- **URL.py**: This script defines the URL and port used to connect to the backend. Remember that if you change the port number here, you must also update it in the Docker Compose YAML file. Otherwise, the GUI will not connect correctly to the container, resulting in a connection error.

We designed different windows (tabs) within a single screen, allowing the user to interact with various parts of the system from one interface. Below is the final implemented GUI:

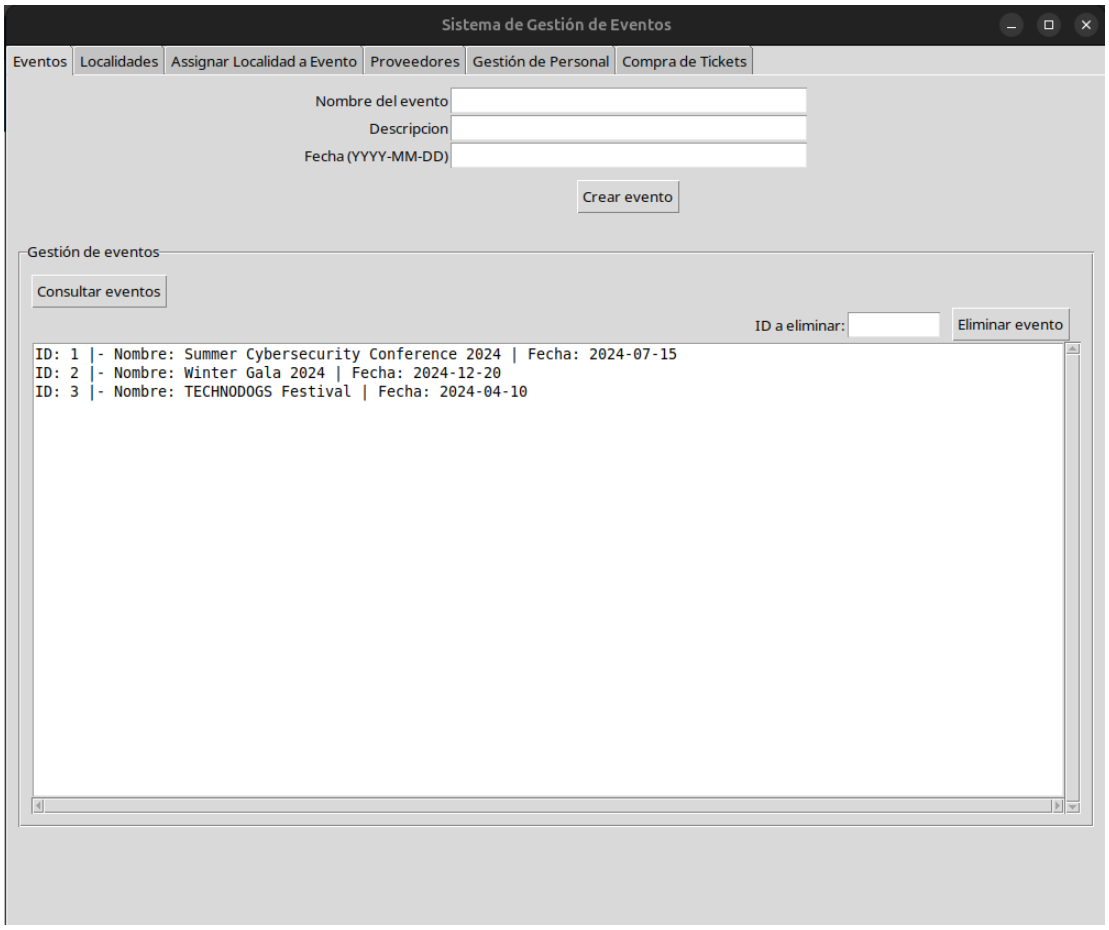

Figure 3: GUI design with Tkinter