Event Ticket Booking System

Pixel Pioneers

Raphaelle Smyth

Yurii Maisuradze

Askhat Bissembay

SRE Foundations c402 | mThree Academy

1. Project Overview

System Name: Event Ticket Booking System
Description: A web application designed to allows users to browse, search, and book tickets for a variety of entertainment events such as concerts, sports games, movies, or theater shows. The system should also include features for event management, ticket availability tracking, seat selection, and payment processing.

2. Objectives

- Scalable: Capable of handling multiple events or users.
- Reliability: Ensures high availability and minimal downtime.
- Security: Implements user authentication, data encryption, and role-based access control.

3. User Characteristics

1. **Admin Users**:
    a. **Role and Responsibilities**: Admin users manage the platform, including creating and managing events, monitoring ticket sales, handling payments, and issuing refunds.
    b. **Access Level**: Full access to all system features and data.
    c. **Skills and Technical Expertise**: Familiar with the event management platform, comfortable navigating a web interface, and capable of generating reports and performing administrative tasks.
2. **General Users**:
    a. **Role and Responsibilities**: Regular users who browse events, book tickets, and manage their bookings and payment details.
    b. **Access Level**: Access to user-specific features such as event browsing, ticket purchasing, payment management, and profile management.
    c. **Skills and Technical Expertise**: Basic technical proficiency to navigate the platform, register accounts, and perform transactions securely.

4. Technology Stack

Back-End Technologies

- **Python**: Used for back-end development.
- **Flask**: A micro web framework for building web applications.
- **MySQL**: Relational database management system for storing data.
- **SQLAlchemy**: Object-Relational Mapping (ORM) for database interactions.

Front-End Technologies

- **HTML/CSS**: For structuring and styling web pages.
- **Bootstrap**: For building responsive and mobile-first web interfaces.
- **Jinja2**: Templating engine for rendering dynamic HTML content.
- **JavaScript**: Enhancing interactivity and functionality on the client side.

DevOps and Infrastructure

- **Docker** (optional): Often used for containerization and deployment.

Other Tools and Platforms

- **GitHub**: For version control and collaborative development.

5. System Requirements

5.1 Functional Requirements

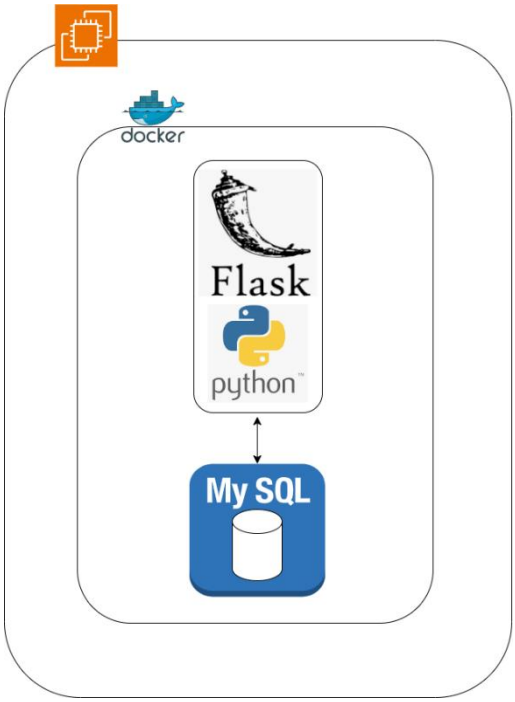
- Event Management: Admins can create, update, and delete events with details such as title, description, date, venue, and available tickets.
- Ticket Booking: Users can select events, choose seats (if applicable), and book tickets for one or multiple events.
- Seat Availability Map: A real-time visual representation of seat availability during the booking process to help users pick the best seats.
- Payment Gateway Integration: Secure payment processing for ticket purchases with multiple payment methods available (e.g., credit card, debit card, UPI).
- Booking Management: Users can view, modify, or cancel their bookings, and receive email or SMS confirmations.
- Notifications and Reminders: Automated email/SMS notifications to remind users of upcoming events, event updates, or changes.
- Sales Analytics for Admins: Reports on ticket sales, event performance, and revenue, enabling admins to monitor event success.
- Refund Management: Admins can manage refunds for canceled or rescheduled events, and notify users about the status of their refunds.
- User Profiles: Allow users to save payment details for quicker bookings in the future and view their booking history.

5.2 Non-functional Requirements

- Scalability: Horizontal scaling with Docker containers.
- Availability: High availability through container orchestration.
- Security: Secure access

6. Architecture Components

6.1 Overview Diagram



6.2 Component Details

Event Ticket Management System (Backend)

- **Models**: Defines database schemas for events, users, bookings, payments, tickets, and related entities, ensuring a structured and relational database system.
- **Views**: Implements business logic to handle user requests and responses, supporting core functionalities like ticket booking, payment processing, and event management.
- **Templates**: Uses frontend HTML templates to dynamically render the user interface for various user interactions.
- **Authentication & Authorization**: Manages user authentication and enforces role-based access control, assigning specific permissions for roles such as admin and user.

MySQL Database

- **Relational Database**: Organizes and maintains structured data across tables like Users, Events, Bookings, Payments, and Ticket Tiers to enable efficient data retrieval and management.
- **Data Backup**: Conducts regular data backups with volume mounts in Docker, ensuring data persistence and protection against data loss.

Docker

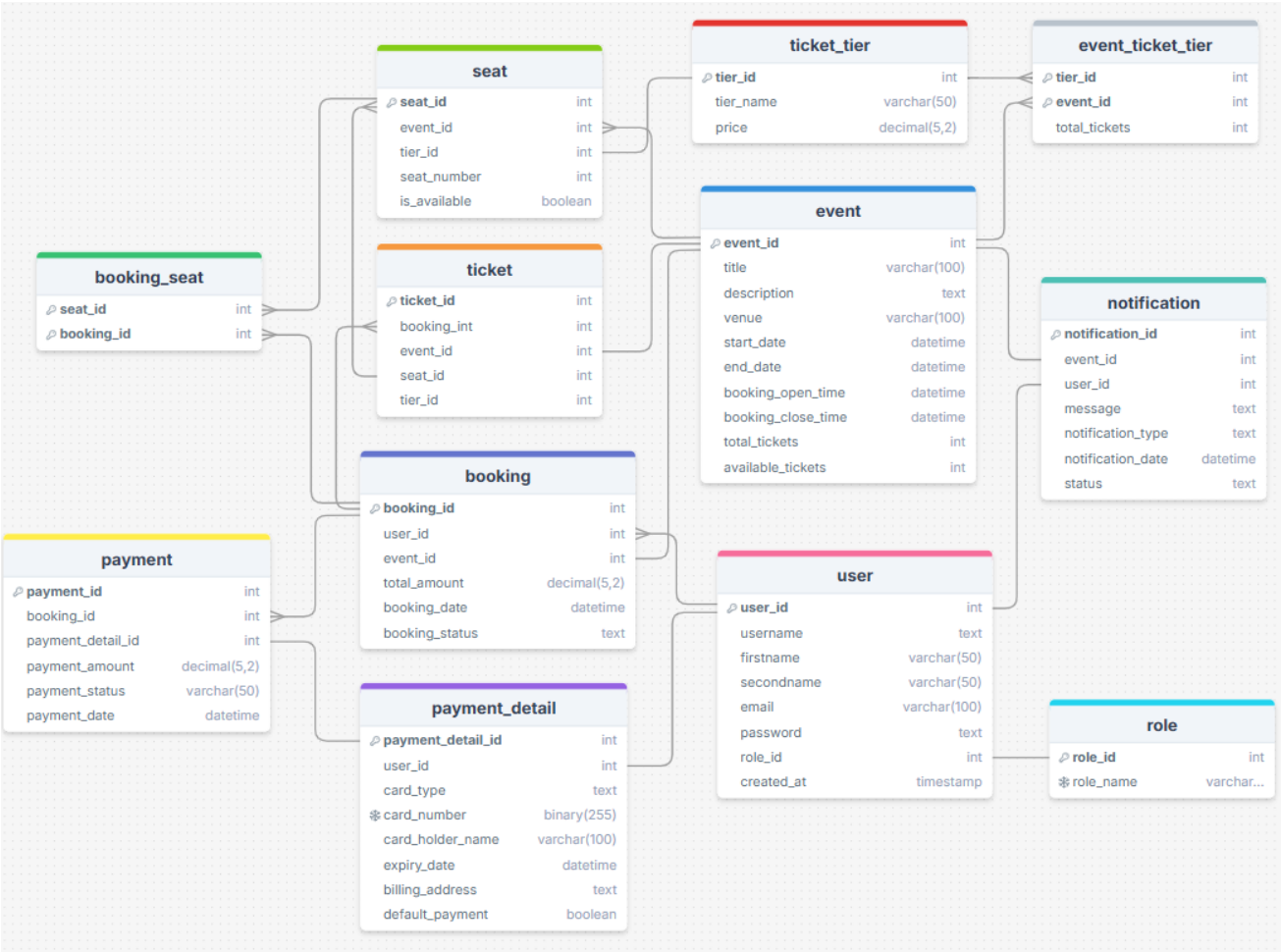
- **Containerization**: Deploys Flask and MySQL in separate Docker containers, providing process isolation and improved scalability.
- **Networking**: Utilizes Docker Compose to manage communication networks between containers, streamlining interactions between components.
- **Persistent Storage**: Ensures MySQL data persists across container restarts through Docker volumes, maintaining consistent data storage.

Frontend (HTML/CSS/JavaScript)

- **Flask Templates**: Generates dynamic HTML pages for seamless user experiences, including event browsing, ticket booking, and payment confirmation processes.
- **Bootstrap**: Enhances the user interface with a responsive and modern design framework, improving accessibility and aesthetic appeal.
- **JavaScript**: Adds interactive elements and enables AJAX for real-time content updates without the need for full page reloads, enhancing user engagement.

7. Database Design

7.1 ER Diagram



- **role**: Stores user role information.
- **user**: Stores user details and their associated roles.
- **event**: Stores event details.
- **booking**: Represents bookings made by users for events.
- **ticket_tier**: Defines different ticket tiers.
- **event_ticket_tier**: Links ticket tiers to events and specifies total tickets per tier.
- **seat**: Defines seat details for events.

- **booking_seat**: Junction table for the many-to-many relationship between bookings and seats.
- **ticket**: Represents tickets linked to bookings.
- **notification**: Manages user notifications related to events.
- **payment_detail**: Stores user payment information.
- **payment**: Tracks payments related to bookings.

7.2 Sample Schema

Data Integrity Measures:

- **ON DELETE CASCADE:**
- **NOT NULL**
- **COMPOSITE UNIQUES**
- **TRIGGERS**
- **Data Encryption – card number varbinary(255)**

```
CREATE DATABASE event_bookings;
USE event_bookings;

CREATE TABLE role (
   role_id INT AUTO_INCREMENT PRIMARY KEY,
   role_name VARCHAR(50) NOT NULL
);

CREATE TABLE `user` (
   user_id INT AUTO_INCREMENT PRIMARY KEY,
   username VARCHAR(50) NOT NULL UNIQUE,
   firstname VARCHAR(50) NOT NULL,
   secondname VARCHAR(50) NOT NULL,
   email VARCHAR(100) UNIQUE NOT NULL,
   password_hash VARCHAR(255) NOT NULL,
   role_id INT NOT NULL,
   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   FOREIGN KEY (role_id) REFERENCES role(role_id) ON DELETE CASCADE
```

```
);

CREATE TABLE event (
    event_id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    description TEXT,
    venue VARCHAR(255) NOT NULL,
    start_date DATETIME NOT NULL,
    end_date DATETIME NOT NULL,
    total_tickets INT NOT NULL,
    available_tickets INT NOT NULL,
    booking_open_time DATETIME,
    booking_close_time DATETIME
);

CREATE TABLE booking (
    booking_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    event_id INT NOT NULL,
    total_amount DECIMAL(10, 2) NOT NULL,
    booking_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    booking_status ENUM('confirmed', 'cancelled') DEFAULT 'confirmed',
    FOREIGN KEY (user_id) REFERENCES `user`(user_id) ON DELETE CASCADE,
    FOREIGN KEY (event_id) REFERENCES event(event_id) ON DELETE CASCADE
);

CREATE TABLE ticket_tier (
    tier_id INT AUTO_INCREMENT PRIMARY KEY,
    tier_name VARCHAR(50) NOT NULL,
    price DECIMAL(10, 2) NOT NULL
);

CREATE TABLE event_ticket_tier (
    event_id INT NOT NULL,
    tier_id INT NOT NULL,
    total_tickets INT NOT NULL,
```

```
    PRIMARY KEY (event_id, tier_id),
    FOREIGN KEY (event_id) REFERENCES event(event_id) ON DELETE CASCADE,
    FOREIGN KEY (tier_id) REFERENCES ticket_tier(tier_id) ON DELETE CASCADE
);

CREATE TABLE seat (
    seat_id INT AUTO_INCREMENT PRIMARY KEY,
    event_id INT NOT NULL,
    tier_id INT,
    seat_number VARCHAR(10) NOT NULL,
    is_available BOOLEAN DEFAULT TRUE,
    FOREIGN KEY (event_id) REFERENCES event(event_id) ON DELETE CASCADE,
    FOREIGN KEY (tier_id) REFERENCES ticket_tier(tier_id) ON DELETE SET NULL,
    UNIQUE(event_id, seat_number)
);

CREATE TABLE booking_seat (
    seat_id INT NOT NULL,
    booking_id INT NOT NULL,
    PRIMARY KEY (seat_id, booking_id),
    FOREIGN KEY (seat_id) REFERENCES seat(seat_id) ON DELETE CASCADE,
    FOREIGN KEY (booking_id) REFERENCES booking(booking_id) ON DELETE CASCADE
);

CREATE TABLE ticket (
    ticket_id INT AUTO_INCREMENT PRIMARY KEY,
    booking_id INT NOT NULL,
    event_id INT NOT NULL,
    seat_id INT NOT NULL,
    tier_id INT NOT NULL,
    FOREIGN KEY (booking_id) REFERENCES booking(booking_id) ON DELETE CASCADE,
    FOREIGN KEY (event_id) REFERENCES event(event_id) ON DELETE CASCADE,
    FOREIGN KEY (seat_id) REFERENCES seat(seat_id) ON DELETE CASCADE,
    FOREIGN KEY (tier_id) REFERENCES ticket_tier(tier_id) ON DELETE CASCADE
);
```

```
CREATE TABLE notification (
    notification_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    event_id INT NOT NULL,
    message TEXT,
    notification_type TEXT,
    notification_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    status ENUM('sent', 'pending') DEFAULT 'pending',
    FOREIGN KEY (user_id) REFERENCES `user`(user_id) ON DELETE CASCADE,
    FOREIGN KEY (event_id) REFERENCES event(event_id) ON DELETE CASCADE
);

CREATE TABLE payment_detail (
    payment_detail_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    card_type ENUM('Visa', 'MasterCard', 'AmEx', 'Discover') NOT NULL,
    card_number VARBINARY(255) NOT NULL,
    cardholder_name VARCHAR(100) NOT NULL,
    expiration_date DATE NOT NULL,
    billing_address TEXT NOT NULL,
    default_payment BOOLEAN DEFAULT FALSE,
    FOREIGN KEY (user_id) REFERENCES `user`(user_id) ON DELETE CASCADE
);

CREATE TABLE payment (
    payment_id INT AUTO_INCREMENT PRIMARY KEY,
    booking_id INT NOT NULL,
    payment_detail_id INT,
    payment_amount DECIMAL(10, 2) NOT NULL,
    payment_status ENUM('pending', 'paid', 'failed', 'refunded') DEFAULT 'pending',
    payment_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (booking_id) REFERENCES booking(booking_id) ON DELETE CASCADE,
    FOREIGN KEY (payment_detail_id) REFERENCES payment_detail(payment_detail_id) ON
DELETE SET NULL
);
```

8. Deployment Architecture

8.1 Docker Configuration

Dockerfile

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt /app/
RUN pip install --no-cache-dir -r requirements.txt
COPY . /app
ENV FLASK_APP=run.py
ENV ENCRYPTION_KEY="$(cat /app/secrets/encryption_key)"
EXPOSE 5000
CMD ["flask", "run", "--host=0.0.0.0"]
```

Docker Compose
Manages both Flask and MySQL services.

```
 version: '3.8'

services:
  flask-app:
    build: .
    container_name: flask-app
    ports:
      - "5000:5000"
    depends_on:
      - mysql-db
    environment:
      - FLASK_APP=run.py
      - ENCRYPTION_KEY=CCbTLCkE3XcX-dUUoV1RcXNJiBchflFe1ROvnELcVJ8=
      - DB_USER=root
      - DB_PASSWORD=root
      - DB_HOST=mysql-db  # Use the service name for Docker networking
```

```
    - DB_NAME=event_bookings
  volumes:
    - .:/app
  networks:
    - app-network

mysql-db:
  image: mysql:8.0
  container_name: mysql-db
  environment:
    MYSQL_ROOT_PASSWORD: root
    MYSQL_DATABASE: event_bookings
  volumes:
    - "./database_setup/sql_setup:/docker-entrypoint-initdb.d"
  ports:
    - "3306:3306"
  networks:
    - app-network

networks:
  app-network:
    driver: bridge
```

8.2 Deployment Steps

Local Development

1. Clone the repository:
    - Run the command: git clone https://github.com/rsmythrepo/Event_ticket_booking_system.git
    - Navigate to the project directory: cd Event_ticket_booking_system
2. Install dependencies:
    - Run the command: pip install -r requirements.txt
3. Set up environment variables:
    - Run the command: setx ENCRYPTION_KEY "crypto-key"
4. Run the application:
    - Run the command: python run.py

5. Access the app:
    - Open the URL: http://localhost:5000 in your web browser.

Local Docker Deployment

1. Build and run the containers:
    - Run the command: docker-compose up --build
2. Access the app:
    - Visit the URL: http://localhost:5000

EC2 Docker Deployment

1. Connect to Your EC2 Instance:
    - Run the command: ssh -i /path/to/your-key.pem ec2-user@your-ec2-public-ip
2. Update the system packages:
    - Run the command: sudo yum update -y
3. Install Docker:
    - Run the command: sudo yum install -y docker
    - Start Docker service: sudo service docker start
    - Enable Docker to start on boot: sudo systemctl enable docker
4. Install Docker Compose:
    - Run the command: sudo curl -L "https://github.com/docker/compose/releases/download/$(curl -s https://api.github.com/repos/docker/compose/releases/latest | grep -Po '"tag_name": "\K[0-9.]+')" /docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
    - Make Docker Compose executable: sudo chmod +x /usr/local/bin/docker-compose
5. Build and run the containers:
    - Run the command: sudo docker-compose up --build -d
6. Show running containers:
    - Run the command: sudo docker ps
7. Configure Security Group:
    - Create an inbound rule under the EC2 security groups:
        - Type: Custom TCP
        - Protocol: TCP
        - Port Range: 5000
        - Source: Anywhere (0.0.0.0/0)
8. Access the app from EC2:
    - Open the URL: http://<EC2_PUBLIC_IP>:5000 in your web browser.

8. Security Considerations

Data Protection: Ensure MySQL is secured with a strong root password, and store sensitive credentials, such as encryption keys and email credentials, in environment variables.

Session Security: Configure Flask sessions securely to prevent unauthorized access, ensuring data is stored in secure locations.

Role-Based Access Control: Use the `User` and `Role` models to enforce role-based access, granting admin-only access to critical management features.

Network Security: Restrict inbound traffic to only necessary ports (5000 for Flask, 3306 for MySQL) in the EC2 security group settings.

SSL: Configure SSL termination at the load balancer or web server level for secure data transmission in a production environment.

9. Monitoring and Logging

Application Logging: Enable Flask logging to track application errors, user activities, and access patterns for audit purposes.

Database Logging: Enable MySQL logging to monitor database queries and detect any performance or security issues.

Error Notifications: Configure email notifications for critical application errors to alert the development team for timely resolution.

10. Future Considerations

Scalability: Use Docker Compose in swarm mode or consider Kubernetes for container orchestration if user traffic increases.

Decoupling Services: Evaluate the possibility of breaking down the monolithic Flask app into smaller services for modularity and scalability as the project grows.

Automated Backups: Set up automated backups for the MySQL database to prevent data loss in case of failures.

CI/CD Pipeline: Integrate a CI/CD pipeline to automate testing, building, and deployment of updates for quicker release cycles.

File Storage: Consider using cloud-based storage for static files (e.g., user uploads or event images) for better scalability and faster access times.