ITE 18 – Application Development and Emerging Technologies

Web/Mobile Application - Final Project Submission

Student Name(s): James Kirby A. Orias

Project Title: Event Ticketing System

Date of Submission: Dec 22-26,2025

Course/Section: ITE 18 -EJ1

---

1. Project Overview

1.1 Project Description

This project is a **Philippine-based** web-**based Event Ticketing System designed** to cater to the local events industry. It allows users to browse events, purchase tickets, and manage their bookings online through a centralized digital platform, replacing manual ticketing processes and fragmented social media-based sales.

By tailoring the system to the Philippine market, it addresses local needs for more organized and transparent transactions. The platform also provides an **admin dashboard** where event organizers can:

- Create and manage events, venues, and categories.
- Monitor real-time payment and sales analytics.
- Streamline local event management to make ticket selling faster and more reliable for both Filipino event-goers and administrators.

1.2 Target Users

- General Users / Attendees
    - Students, young professionals, and the general public who want to discover events, check details (date, time, venue, price), and buy tickets online.
    - They need an easy, fast, and secure way to browse events and manage their purchased tickets.
- Admin Users / Event Organizers
    - Event organizers or admin staff who manage concerts, seminars, school events, etc.
    - They need tools to create events, define venues and categories, track ticket sales, and monitor payments and revenue.

1.3 Key Features

List the main features and functionalities of your application.

- Feature 1: User Authentication
    - User and admin login, registration, and logout with token-based authentication.
- Feature 2: Event Browsing and Search
    - Users can view a list of upcoming events, filter by category, and see event details (date, time, venue, price, description, image).
- Feature 3: Ticket Purchase and Payments
    - Users can purchase tickets for selected events and create payments (simulated or via backend payment endpoint).
- Feature 4: User Dashboard
    - Users can view their purchased tickets, see ticket details, and manage their bookings.
- Feature 5: Admin Dashboard
    - Admins can view key statistics such as total events, total tickets, total users, and total revenue.
- Feature 6: Admin Event Management
    - Admins can create, update, and delete events, including uploading event images and setting prices, capacities, and statuses.
- Feature 7: Admin Venue Management
    - Admins can manage venues (name, address, city, capacity, description).
- Feature 8: Admin Category Management
    - Admins can manage event categories (name, description, icon).
- Feature 9: Admin Orders / Payments Management
    - Admins can view payments, filter by status or method, see customer and event details, and calculate revenue statistics.

1.4 Technology Stack

- **Frontend:**
    - Next.js (React + TypeScript)
    - Tailwind CSS for styling
    - Shadcn UI components / custom UI components
- **Backend:**
    - Laravel (PHP)
    - Laravel Sanctum for API authentication
- **Database:**
    - MySQL (via Laravel's Eloquent ORM)
- **Other Tools:**
    - Postman for API testing
    - Figma for UI/UX design (based on provided mockups)
    - XAMPP (Apache, PHP, MySQL) for local backend environment
    - npm / Node.js for frontend package management and build tools

---

2. App Plan

2.1 Project Scope

In Scope:

- **User Authentication**
    - User and admin login, registration, and logout using API tokens.
- **Event Browsing and Details**
    - View list of events, search/filter by category, and view event details.
- **Ticket Purchase and Payment Creation**
    - Create tickets and payments for selected events.
- **User Dashboard**
    - View list of user's purchased tickets and ticket details.

- **Admin Dashboard**
    - View aggregated stats: number of events, users, tickets, revenue, and basic analytics.
- **Admin Management Modules**
    - Events: Create, read, update, delete events.
    - Venues: CRUD operations for venues.
    - Categories: CRUD operations for categories.
    - Admin Payments/Orders: View and filter payments/orders and simple statistics.

Out of Scope:

- **Advanced seat selection** (specific seat maps, seat-level selection).
- **Multiple organizers with separate branded portals** (multi-tenant architecture).
- **Complex payment gateway integration with real banks** (e.g., PayPal/Stripe fully implemented with webhooks).
- **Native mobile apps** (Android/iOS); only web UI is covered.

2.2 Objectives & Goals

1. Provide a complete, working event ticketing flow from browsing events to purchasing tickets and recording payments.
2. Enable admins to manage events, venues, categories, and view basic sales and payment analytics through an admin dashboard.
3. Implement secure authentication and authorization, ensuring that only admins can access admin features and only logged-in users can purchase tickets.
4. Deliver a clean, modern, and responsive UI that follows the given Figma design while being fully integrated with the Laravel backend API.

2.3 User Stories & Use Cases

User Story 1: Browse and View Events (User)

- As a user, I want to browse a list of upcoming events, so that I can find interesting events to attend.

Acceptance Criteria:

- I can see a list of events with key information (title, date, venue, price).
- I can filter or search events by category or keyword.
- I can click an event to view detailed information (description, full date/time, venue, image).

User Story 2: Purchase Tickets (User)

- As a user, I want to purchase tickets for an event, so that I can reserve my spot and pay online.

Acceptance Criteria:

- I must be logged in to purchase tickets.
- I can select an event and proceed to a purchase flow (quantity, confirmation).
- After purchase, a ticket record is created and linked to my account.
- A payment record is created with the correct amount and status.
- I can see my new ticket in my user dashboard.

User Story 3: Manage Events (Admin)

- As an admin, I want to create and manage events, so that I can publish new events and keep event information up to date.

Acceptance Criteria:

- Only admins can access the admin dashboard and event management pages.
- I can create a new event with title, description, date, time, venue, category, total tickets, price, and image.
- I can edit an existing event (e.g., change date, price, or status).
- I can delete an event if needed.
- Event changes are reflected on the public event listing and details.

User Story 4: View Sales and Payments (Admin)

- As an admin, I want to view orders/payments and basic stats, so that I can understand revenue and ticket performance.

Acceptance Criteria:

- I can see a list of payments/orders with customer, event, amount, status, and date.
- I can filter payments by status or payment method.
- I can see summarized stats: total orders, total revenue, number of paid
- Dashboard stats (e.g., total events, tickets, users, revenue) are visible in the admin dashboard.

2.4 System Architecture

The system utilizes a **Three-Tier Architecture**, ensuring a clean separation between the user interface, business logic, and data storage.

1. Presentation Layer (Frontend)

- **Technology:** Next.js (React + TypeScript).
- **Role:** Handles the user interface and client-side logic. It uses a **Service Layer (api.ts)** to communicate with the backend and an **AuthProvider** to manage user sessions and role-based access (User vs. Admin) via local storage tokens.

2. Application Layer (Backend)

- **Technology:** Laravel (RESTful API).
- **Role:** Processes business logic, validates data, and secures endpoints.
- **Security:** Uses **Laravel Sanctum** for token-based authentication. It ensures that only authorized users or administrators can access specific resources (e.g., event creation or payment reports).

3. Data Layer (Database)

- **Technology:** MySQL.
- **Role:** Stores all persistent data, including user accounts, event details, venue information, and transaction records. Laravel's **Eloquent ORM** is used to manage database interactions efficiently.

High-Level Interaction Flow

1. **Request:** The user interacts with the **Next.js** UI (e.g., booking a ticket).
2. **API Call:** The frontend sends an HTTP request with a **Bearer Token** to the **Laravel API**.
3. **Processing:** Laravel validates the request, checks permissions, and interacts with the **MySQL** database via models.
4. **Response:** The backend sends a **JSON response** back to the frontend.

5. **Update:** Next.js dynamically updates the UI to reflect the changes (e.g., showing a "Booking Successful" message).

2.5 Development Timeline

- **Phase 1 – Design & Planning (Week 1–2)**
    - Gather requirements (user roles, flows, features).
    - Study reference systems and define scope (what is in/out).
    - Draft initial wireframes and information architecture.
    - Plan database schema (tables, relationships, keys).
- **Phase 2 – Backend & Database Setup (Week 2–4)**
    - Set up Laravel project and configure .env (database, CORS).
    - Create migrations and models for users, admins, events, venues, categories, tickets, payments.
    - Implement core API endpoints for authentication, events, venues, categories.
    - Seed the database with sample data for testing.
- **Phase 3 – Frontend Structure & UI Layout (Week 3–5)**
    - Initialize Next.js + Tailwind project.
    - Implement global layout (root layout, navigation, AuthProvider).
    - Build core pages: landing, login/register, events listing, basic dashboard layout.
    - Integrate Figma design into reusable components (buttons, cards, forms, dialogs).
- **Phase 4 – Feature Implementation (Frontend + Backend Integration) (Week 5–6)**
    - Connect UI to backend APIs via api.ts service layer.
    - Implement user flows: login, event browsing, ticket purchase, payments.
    - Build admin flows: events CRUD, venues CRUD, categories CRUD.
    - Implement dashboards: user dashboard and admin dashboard (stats & charts).
- **Phase 5 – Integration, Testing & Refinement (Week 6–8)**
    - End-to-end testing of all main flows (user and admin).
    - Fix bugs in routing, authentication, and form validation.
    - Improve UX (loading states, error messages, empty states).
    - Prepare project documentation, deployment notes, and final presentation.

---

3. UI/UX Design

3.1 Design Philosophy

- **Modern and Clean:**
    - Card-based layouts, generous spacing, and clear typography make data-heavy screens (like admin orders) easy to scan.
- **Role-Based Clarity:**
    - Users see a simplified experience focused on browsing events and managing tickets.
    - Admins see a more data-driven interface with statistics, tables, and management tools.
- **Consistency:**
    - Shared components (buttons, inputs, cards, navigation) are reused across pages to keep the UI consistent.
    - Colors and spacing follow a single design system defined in CSS variables and Tailwind.
- **Accessibility & Readability:**
    - Sufficient color contrast between text and background (e.g., dark text on light backgrounds, white text on blue primary).
    - Clear hierarchy using typography (larger, bolder headings; smaller body text).
- **Responsive Layout:**
    - Layouts adapt to different screen sizes (sidebar collapses into a mobile menu, cards stack vertically on small screens).
- **Focus on Feedback:**
    - Loading spinners, toast notifications (success/error), and dialog confirmations help users understand what is happening.

3.2 Color Scheme

- **Primary Color:**
    - Hex: #0ea5e9 (a bright sky blue)
    - Used for: primary buttons, links, highlights, call-to-action elements, charts, accent outlines.
- **Secondary Color:**
    - Light mode: #f1f5f9
    - Dark mode: #1e1e2e
    - Used for: secondary buttons, subtle backgrounds (panels, sections, muted cards).
- **Accent Color:**
    - Hex: #0ea5e9 (same as primary to keep the palette simple)
    - Used for: hover states, icons, accent borders, selected states.
- **Background Color:**
    - Light mode: #ffffff
    - Dark mode: #0a0a0f
    - Used for: main page background; dark mode gives a “dashboard” feeling that fits analytics.
- **Text Color (Foreground):**
    - Light mode: #0a0a0f (very dark navy/black)
    - Dark mode: #f8fafc (very light gray/white)
- **Muted / Subtext Colors:**
    - Light mode muted foreground: #64748b (gray-blue)
    - Dark mode muted foreground: #94a3b8
    - Used for: secondary text, descriptions, help text.
- **Destructive Color:**
    - Light mode: #ef4444
    - Dark mode: #ef4444
    - Used for: delete buttons, error states, destructive actions.

3.3 Typography

- **Font Family:**
  - **Inter** (Google Font), applied globally in the root layout.
- **Base Font Size:**
  - 16px (set via --font-size: 16px on the html / :root).
- **Heading Sizes (approximate, from your CSS):**
  - H1: 1.875rem ≈ 30px, bold, for main page titles.
  - H2: 1.5rem ≈ 24px, for section headings.
  - H3: 1.25rem ≈ 20px, for card titles and sub-sections.
  - H4: 1.125rem ≈ 18px, for smaller headings or labels.
- **Body Text:**
  - Font size: 1rem (16px).
  - Line height: about 1.5 for good readability.
- **Weight:**
  - Headings: 500–700 (medium to bold).
  - Body: 400 (normal).

3.4 UI Components

- **Button**
  - Style: Rounded corners (border-radius: 0.75rem base), solid primary background (#0ea5e9) with white text for primary actions; secondary/ghost variants for less important actions.
  - Usage: Submit forms (login, register, create event), navigation actions (e.g., “Create Event”, “Book Now”), and destructive actions (delete) with the destructive color.
- **Input Fields**
  - Style: Subtle borders (var(--border)), filled backgrounds in light mode (#e2e8f0) and darker in dark mode.
  - Include labels and placeholders, with inline or toast-based error messages when validation fails.
  - Validation:
    - Required fields (e.g., email/password, event name, venue name).

        - Basic format validation (e.g., email format, number fields for capacity/price).

- **Navigation Bar / Sidebar**
    - For dashboards (LayoutDashboard):
        - Left sidebar with app logo at top, navigation items with icons (Dashboard, Events, Orders, Venues, Categories, Settings).
        - Active item highlighted with primary color background.
        - On mobile, collapses into a menu (hamburger icon).
    - For landing/public pages:
        - Top navigation with links to main sections (e.g., “Browse Events”, “Categories”, “Login/Register”).
- **Cards**
    - Used for:
        - Event cards (image, title, date/time, price, “Book Now” button).
        - Statistic cards (total revenue, number of tickets sold, total events).
    - Style: White/dark card background, subtle shadow, rounded corners, clear separation between header and content.
- **Modal/Dialog**
    - Used in admin pages for actions like creating/editing venues and categories, or viewing order details.
    - Style: Centered dialog with card look, backdrop overlay, primary buttons (Save/Confirm) and secondary buttons (Cancel).
- **Tables / Lists**
    - Orders and payments are shown in table-like layouts with rows, columns, and badges for status (e.g., “Paid”, “Pending”, “Failed”).
    - Filters and search inputs at the top of the table for admin convenience.
- **Badges / Status Chips**
    - Small rounded labels with color-coded states (primary, muted, destructive) to quickly show status (e.g., ticket/payment status).

3.5 Wireframes & Mockups

Event Ticketing System Wireframe

https://www.figma.com/design/fPlX8TWBIFeoPdbCeduW3S/EVENT-TICKETING-SYSTEM?node-id=0-1&t=7UPC0VIgHi4tLWJj-1

3.6 User Flows

- **Flow 1: User Registration & Login**
    1. User visits the landing page.
    2. Clicks "Register" and is navigated to /register.
    3. Fills in registration form (username, email, password, confirmation) and submits.
    4. Frontend sends POST /api/v1/auth/register to Laravel.
    5. On success, backend returns a token; frontend stores it in localStorage and updates AuthContext.
    6. User is redirected to /events (user role) to start browsing events.
    7. For returning users, they go to /login, submit credentials, call POST /api/v1/auth/login or /admin-auth/login (for admin), then redirect to /events or /admin/dashboard based on role.
- **Flow 2: Making a Ticket Purchase (User)**
    1. Logged-in user visits /events.
    2. Selects an event card and goes to the event details page (e.g., /events/[id]).
    3. Clicks "Purchase"/"Book Now" to start the purchase flow (e.g., quantity selection).
    4. Frontend sends POST /api/v1/tickets to create a ticket record for the user + event.
    5. Then sends POST /api/v1/payments with the amount and ticket ID to create a payment record.
    6. On success, a confirmation message is shown and the user is redirected to /dashboard to see their tickets.

- **Flow 3: Admin Managing Events**
    1. Admin logs in via /login using admin credentials.
    2. After auth, admin is redirected to /admin/dashboard.

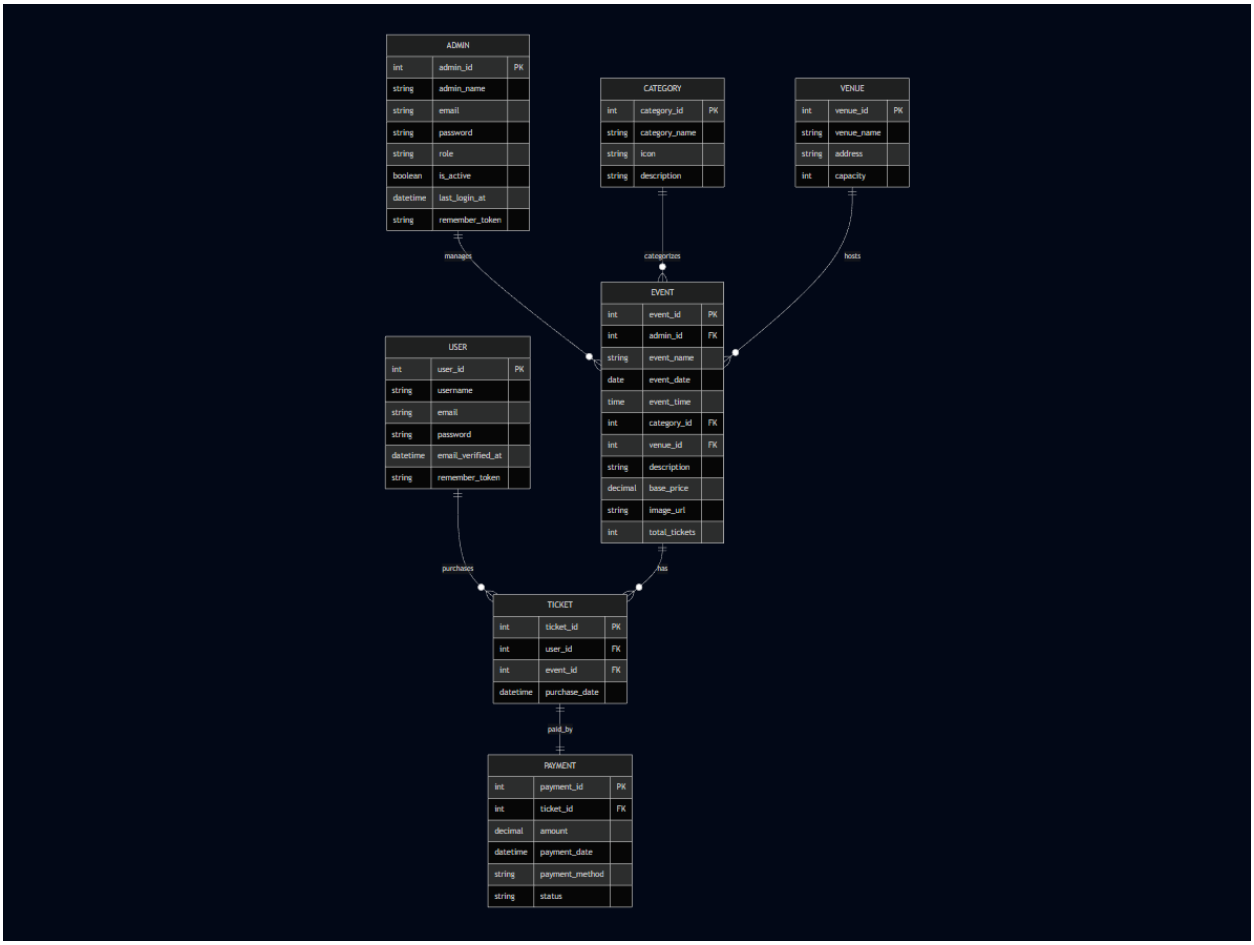
3. From the sidebar, admin clicks “Events” or “Create Event” (routes to /admin/events/create).
4. Fills in event form (name, date, time, venue, category, tickets, price, image) and submits.
5. Frontend sends POST /api/v1/admin/events with form data (JSON or FormData for image).
6. Backend validates, stores the event, and returns data; UI shows success toast and may redirect back to events list or dashboard.
7. For updates, admin uses the edit option; frontend sends PUT /api/v1/admin/events/{id}.

- **Flow 4: Admin Viewing Orders / Payments**
  1. Admin goes to /admin/orders.
  2. Page loads and calls GET /api/v1/admin/payments with optional filters (status, method, search).
  3. Backend returns payments with related ticket, event, and user data.
  4. UI displays statistics (total orders, revenue, paid vs pending) and the list of orders.
  5. Admin can open an order’s details dialog to see full information.

---

4. Database Architecture (ERD)

4.1 Entity Relationship Diagram



4.2 Entity Descriptions

Provide detailed descriptions of each entity in your database.

Entity: Users

Field	Data Type	Constraints	Description
user_id	BIGINT	PRIMARY KEY, AUTO_INCREMENT	Unique identifier for each user.
username	VARCHAR(255)	NOT NULL, UNIQUE	Public username used for login/display.
email	VARCHAR(255)	NOT NULL, UNIQUE	User’s email address used for

Field	Data Type	Constraints	Description
			login and notifications.
password	VARCHAR(255)	NOT NULL	Hashed password (stored securely by Laravel).
created_at	TIMESTAMP	NULLABLE	Timestamp when the user was created.
updated_at	TIMESTAMP	NULLABLE	Timestamp when the user was last updated.

Entity: Admins

Field	Data Type	Constraints	Description
admin_id	BIGINT	PRIMARY KEY, AUTO_INCREMENT	Unique identifier for each admin.
admin_name	VARCHAR(255)	NOT NULL	Admin's display name.
email	VARCHAR(255)	NOT NULL, UNIQUE	Admin login email.
password	VARCHAR(255)	NOT NULL	Hashed admin password.
role	VARCHAR(50)	NOT NULL	Admin role (e.g., "super_admin", "manager").
is_active	BOOLEAN	NOT NULL, DEFAULT TRUE	Indicates if the admin account is active.

Field	Data Type	Constraints	Description
last_login_at	TIMESTAMP	NULLABLE	Last login timestamp for auditing.
created_at	TIMESTAMP	NULLABLE	When the admin account was created.
updated_at	TIMESTAMP	NULLABLE	When the admin account was last updated.

Entity: Venues

Field	Data Type	Constraints	Description
venue_id	BIGINT	PRIMARY KEY, AUTO_INCREMENT	Unique identifier for each venue.
venue_name	VARCHAR(255)	NOT NULL	Name of the venue (e.g., “Main Hall”).
address	VARCHAR(255)	NOT NULL	Full address/location of the venue.
capacity	INT	NOT NULL	Maximum number of attendees the venue can hold.
created_at	TIMESTAMP	NULLABLE	When the venue record was created.
updated_at	TIMESTAMP	NULLABLE	When the venue record was last updated.

Entity: Categories

Field	Data Type	Constraints	Description
category_id	BIGINT	PRIMARY KEY, AUTO_INCREMENT	Unique identifier for each category.
category_name	VARCHAR(255)	NOT NULL	Name of the category (e.g., “Concert”, “Seminar”).
created_at	TIMESTAMP	NULLABLE	When the category was created.
updated_at	TIMESTAMP	NULLABLE	When the category was last updated.

Entity: Events

Field	Data Type	Constraints	Description
event_id	BIGINT	PRIMARY KEY, AUTO_INCREMENT	Unique identifier for each event.
admin_id	BIGINT	NOT NULL, FOREIGN KEY → admins(admin_id)	Admin who created/owns the event.
event_name	VARCHAR(255)	NOT NULL	Title/name of the event.
event_date	DATE	NOT NULL	Date when the event will take place.
category_id	BIGINT	NOT NULL, FOREIGN KEY → categories(category_id)	Category of the event.

Field	Data Type	Constraints	Description
venue_id	BIGINT	NOT NULL, FOREIGN KEY → venues(venue_id)	Venue where the event is held.
created_at	TIMESTAMP	NULLABLE	When the event was created.
updated_at	TIMESTAMP	NULLABLE	When the event was last updated.

Entity: Tickets

Field	Data Type	Constraints	Description
ticket_id	BIGINT	PRIMARY KEY, AUTO_INCREMENT	Unique identifier for each ticket.
user_id	BIGINT	NOT NULL, FOREIGN KEY → users(user_id)	User who purchased the ticket.
event_id	BIGINT	NOT NULL, FOREIGN KEY → events(event_id)	Event this ticket belongs to.
purchase_date	TIMESTAMP	NOT NULL	Date and time when the ticket was purchased.
created_at	TIMESTAMP	NULLABLE	When the ticket record was created.
updated_at	TIMESTAMP	NULLABLE	When the ticket record was last updated.

Entity: Payments

Field	Data Type	Constraints	Description
payment_id	BIGINT	PRIMARY KEY, AUTO_INCREMENT	Unique identifier for each payment.
ticket_id	BIGINT	NOT NULL, FOREIGN KEY → tickets(ticket_id)	Ticket associated with this payment.
amount	DECIMAL(10,2)	NOT NULL	Amount paid for the ticket.
payment_date	TIMESTAMP	NOT NULL	Date and time when payment was made.
payment_method	VARCHAR(100)	NOT NULL	Method used (e.g., “GCash”, “Credit Card”, etc.).
status	VARCHAR(50)	NOT NULL	Payment status (e.g., “completed”, “pending”, “failed”).
created_at	TIMESTAMP	NULLABLE	When the payment was recorded.
updated_at	TIMESTAMP	NULLABLE	When the payment record was last updated.

4.3 Relationships

- **Relationship 1: Users – Tickets (One-to-Many)**
    - One user can have many tickets, but each ticket belongs to a single user. This represents all the events that a user has bought tickets for.
- **Relationship 2: Events – Tickets (One-to-Many)**
    - One event can have many tickets, but each ticket is linked to one specific event. This captures all attendees for that event.
- **Relationship 3: Events – Venues (Many-to-One)**
    - Many events can be held at the same venue, but each event takes place at one venue. This avoids repeating venue details for every event.
- **Relationship 4: Events – Categories (Many-to-One)**
    - Many events can share the same category (e.g., Concert, Seminar), but each event belongs to one main category. This allows grouping and filtering events by type.
- **Relationship 5: Admins – Events (One-to-Many)**
    - One admin can manage many events, but each event is managed by a single admin. This shows who is responsible for creating and maintaining each event.
- **Relationship 6: Tickets – Payments (One-to-One)**
    - Each ticket can have one payment record that represents how it was paid. The payment describes the amount, method, and status for that ticket purchase.

4.4 Database Normalization

Our event ticketing database is designed to follow the main normalization rules: First Normal Form (1NF), Second Normal Form (2NF), and Third Normal Form (3NF).

First Normal Form (1NF)
The database satisfies 1NF because each table:

- Has a primary key that uniquely identifies each row (e.g., user_id, event_id, ticket_id).
- Stores only atomic (indivisible) values in each column. For example:
    - The events table stores a single event date per row, not a list of dates.
    - The venues table stores one capacity value per venue.
    - The tickets table stores one user_id and one event_id for each ticket. There are no repeating groups or multi-valued columns.

Second Normal Form (2NF)
The database satisfies 2NF because:

- All tables use a single-column primary key (like event_id, venue_id), so there are no composite keys.
- Every non-key attribute in a table depends on the whole primary key. For example, in the events table, attributes such as event_name, event_date, venue_id, and category_id all describe that specific event identified by event_id. There are no attributes that depend on only part of a key.

Third Normal Form (3NF)
The database satisfies 3NF by avoiding transitive dependencies (non-key attributes depending on other non-key attributes):

- Venue details (like address and capacity) are stored only in the venues table, not duplicated in events or tickets.
- Category information (such as category_name) is stored only in the categories table, not directly inside events.
- Payment information (amount, payment_method, status) is stored in the payments table and linked to a ticket, instead of being repeated on tickets or users.

By separating data into these related tables, the design reduces redundancy, keeps data consistent, and makes updates safer (for example, changing a venue's address is done in one place and automatically reflected for all events that use that venue).

---

5. Application Features & Functionality

5.1 Feature 1: User & Admin Authentication

Description: Handles login, registration (users), and admin login, and manages session state for protected pages.

Functionality:

- Users can register with username, email, and password.
- Users and admins can log in to receive an API token.
- The frontend stores the token and keeps the user logged in.
- Admins are redirected to the admin dashboard; normal users to the events page.

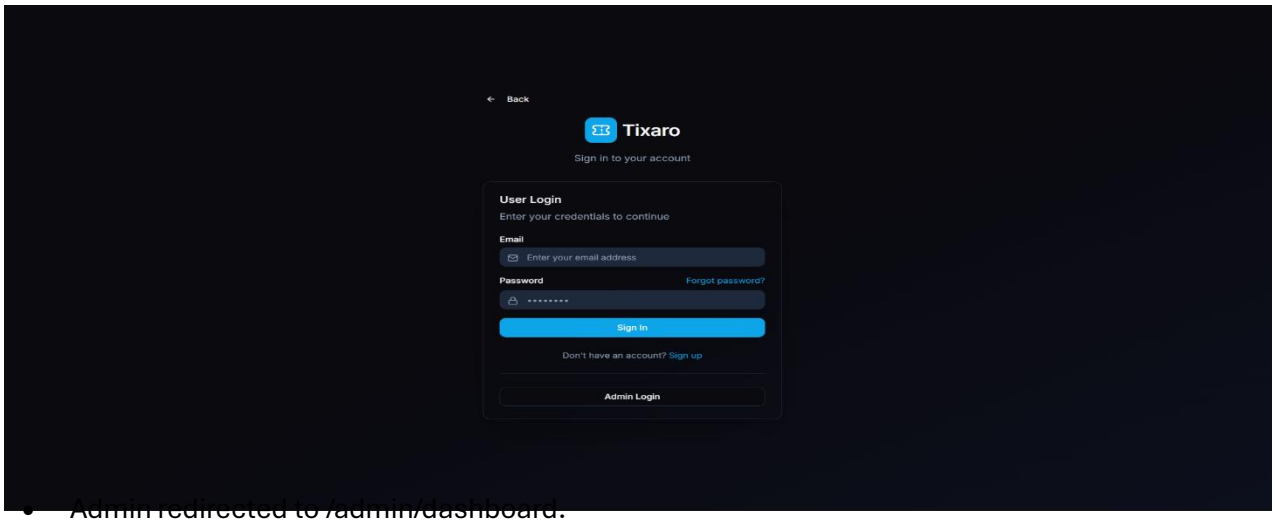
Implementation Details:

- Backend: Laravel routes /api/v1/auth/register, /api/v1/auth/login, /api/v1/admin-auth/login, /api/v1/users/logout, /api/v1/admin/logout, /api/v1/users/me, /api/v1/admin/profile.
- Tokens are handled via Laravel Sanctum.

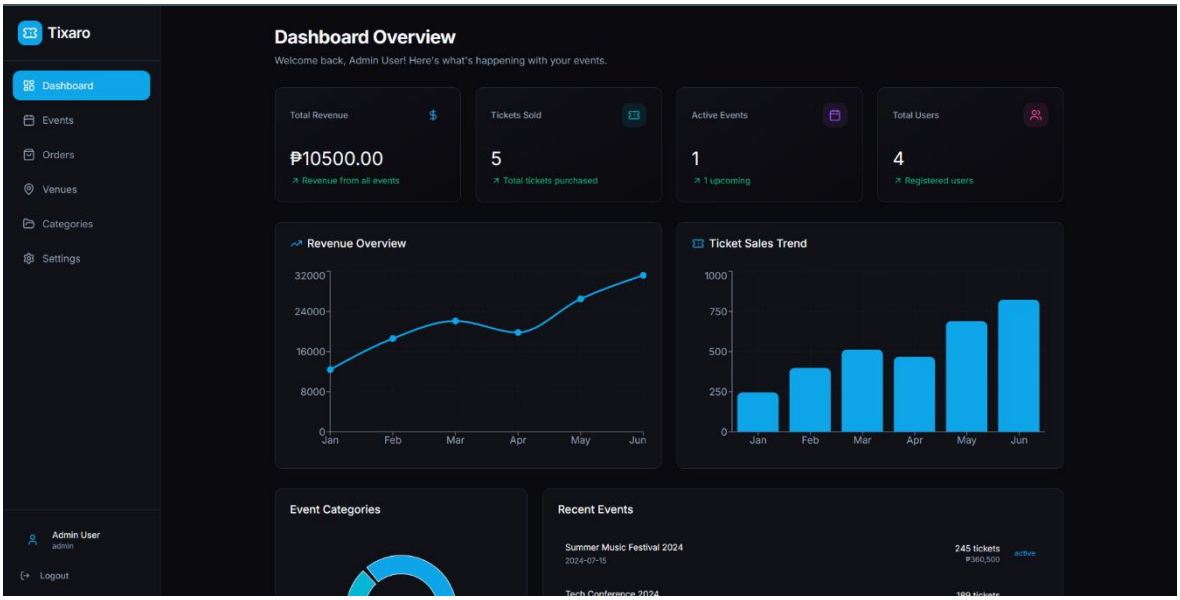
- Frontend: AuthContext in src/contexts/AuthContext.tsx stores user and token, uses localStorage and provides login, logout, and getCurrentUser.
- Protected pages (e.g., /dashboard, /admin/dashboard, /admin/venues) check the user and user.role and redirect to /login if unauthorized.

Screenshots:

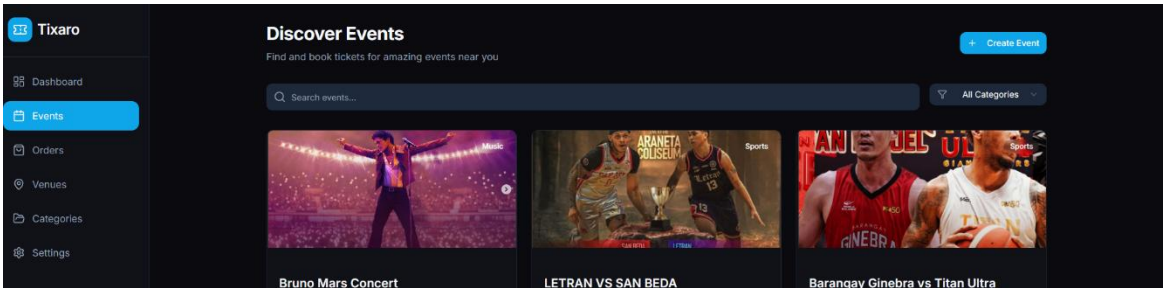
- Login page (user/admin toggle).



- Admin redirected to /admin/dashboard.



- User redirected to /events.



5.2 Feature 2: Event Browsing & Search

Allows users to view a list of events, filter/search, and see event details.

Functionality:

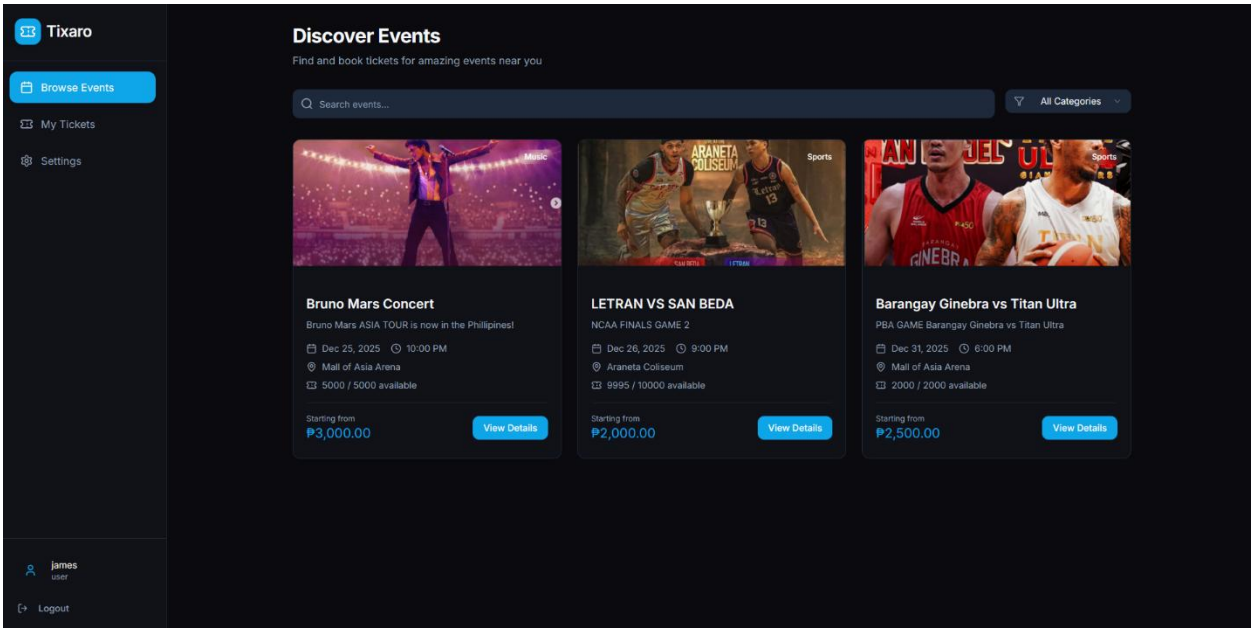
- Display list of events with title, date, venue, price, and image.
- Filter events by category or search keyword.
- Open a detailed view for a single event.

Implementation Details:

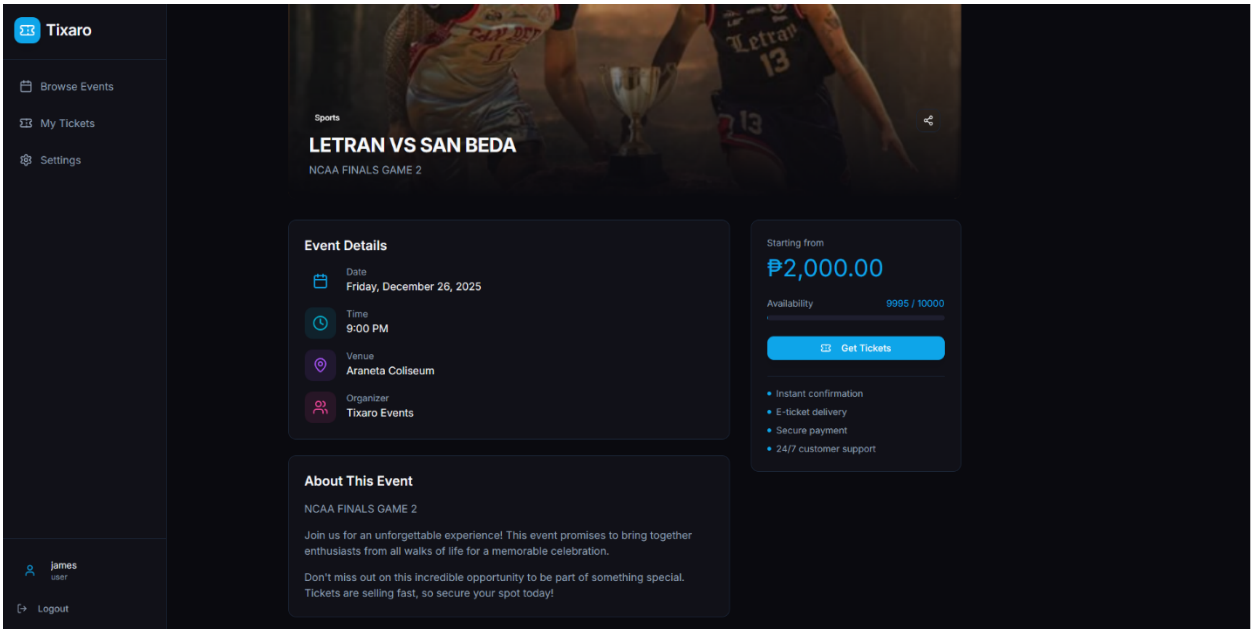
- Backend: GET /api/v1/events (with query params like search and category) and GET /api/v1/events/{id}.
- Frontend: EventsPage in EventsPage.tsx calls eventsApi.getEvents() from api.ts, then renders cards.
- Event details page uses dynamic routes (/events/[id]) to show full information based on the selected event.

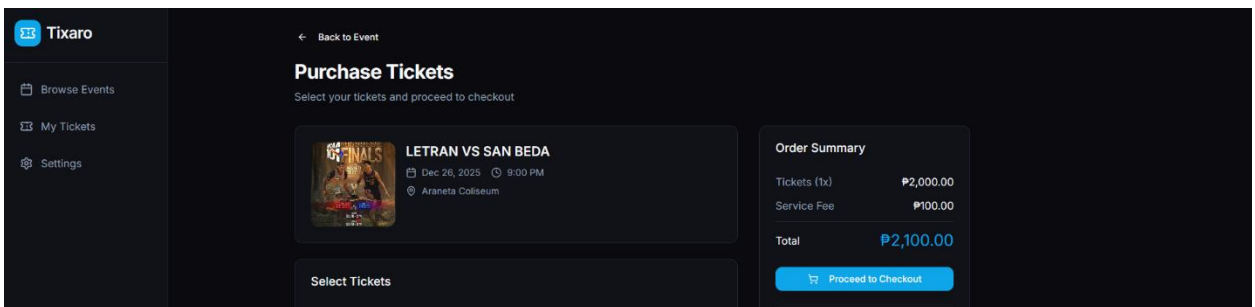
Screenshots

- **Events list page showing multiple event cards.**



- **Event detail page with full description and “Get Tickets” button.**





---

6. Security & Error Handling

6.1 Security Measures Implemented

- **Authentication:**
    - Username/email + password login.
    - Laravel Sanctum tokens for API authentication.
    - Tokens are stored in localStorage on the frontend and sent via Authorization: Bearer <token> in api.ts.
- **Authorization:**
    - Laravel middleware for auth:sanctum and admin restricts admin routes (/api/v1/admin/...) to admins only.
    - Frontend checks user.role in AuthContext to block admin pages for non-admin users and redirect to /login.
- **Input Validation:**
    - Backend: Laravel validation in controllers for fields like event_name, event_date, capacity, amount, etc.

    - Frontend: Basic validation in forms (required fields, number formats) before calling APIs.

- **Data Protection:**
    - Passwords are hashed using Laravel's built-in hashing (bcrypt/argon).
    - Tokens are random and not guessable.
    - No plain-text passwords or tokens stored in the database.
- **Protection Against Vulnerabilities:**
    - SQL Injection: Prevented by using Eloquent ORM and query builder with bound parameters instead of raw SQL.
    - XSS: React/Next.js escapes output by default; no unsafe innerHTML is used for user content.
    - CSRF: For normal web routes, Laravel CSRF protection is enabled; for API routes, Sanctum + token-based auth reduces CSRF impact.

6.2 Error Handling

- **Backend Error Handling:**
    - Laravel controllers return JSON responses with success, message, and errors fields.
    - Appropriate HTTP status codes (e.g., 400, 401, 403, 404, 422, 500) are used to indicate error types.
- **Frontend Error Handling:**
    - api.ts wraps fetch calls in try/catch and throws errors with meaningful messages if response.ok is false.
    - Components catch errors and display them using toast notifications (e.g., toast.error('Failed to load events')).
    - Forms show validation errors, and pages show loading states and fallbacks (e.g., mock data in OrdersPage when API fails).
- **User Experience:**
    - Errors are shown in a user-friendly way (toasts, messages) instead of raw stack traces.

- Protected pages redirect unauthorized users to /login instead of showing broken content.

---

7. Installation & Setup Instructions

7.1 Prerequisites

List all software and tools required to run your application.

- PHP 8.1+
- Composer
- Node.js 18+ and npm
- MySQL 8+ (or MariaDB equivalent)
- XAMPP or similar (Apache + PHP + MySQL)
- Git

7.2 Installation Steps

Provide step-by-step instructions to set up and run your application.

1. Clone the repository: git clone [your-repo-url]
2. Navigate to the project directory: cd [project-name]
3. Install dependencies: [command for your technology stack]
4. Configure environment variables: [instructions for setup]
5. Set up the database: [database setup instructions]
6. Run the application: [command to start the app]

7.3 Running the Application

Explain how to access and use the application once it's running.

[Provide URL/port information and initial login credentials if applicable]

---

8. Testing

Test Case	Steps	Expected Result	Actual Result	Status
Login as User	**1) Open /login 2) Enter valid user email/password 3) Click Login**	**User is authenticated**	**Works as expected;**	**Pass**

Test Case	Steps	Expected Result	Actual Result	Status
		and redirected to /events; user token stored; events list loads.	user redirected to events page and token saved.	
Login as Admin	1) Open /login 2) Switch to admin login (if applicable) 3) Enter valid admin credentials 4) Click Login	Admin is authenticated and redirected to /admin/dashboard; admin stats load.	Works as expected; dashboard stats and cards appear.	Pass
Browse Events & View Details	1) Login as user 2) Go to /events 3) Use search/filter 4) Click an event card	Events list loads from API; filters change results; clicking an event opens details page with correct info.	Events load; filters work; details page shows correct event.	Pass
Purchase Ticket & Payment	1) Login as user 2) Open an event 3) Start purchase flow 4) Confirm purchase/payment	A ticket record is created and a payment is recorded; user sees success message and can see ticket in /dashboard.	Ticket and payment are created; dashboard shows new ticket.	Pass
Admin Create Event	1) Login as admin 2) Go to /admin/events/create 3) Fill in form 4) Submit	New event appears in events list and is visible to users on /events.	New event is created and visible on user events page.	Pass

Test Case	Steps	Expected Result	Actual Result	Status
Admin Manage Venues/Categories	1) Login as admin 2) Go to /admin/venues or /admin/categories 3) Create/edit/delete entries	Changes are saved; updated venues/categories appear in event forms and filters.	CRUD operations work; dropdowns show updated data.	Pass
Access Control (Unauthorized)	1) Try to open /admin/dashboard as logged-out user 2) Try as normal user	Both are redirected to /login (or user dashboard) and cannot see admin content.	Redirects work; admin pages not visible to non-admins.	Pass
API Error Handling	1) Stop backend or break connection 2) Try to load events/venues from UI	UI shows error toast and/or fallback content; app does not crash.	Toasts appear; mock/fallback data used where implemented.	Pass

8.2 Known Issues & Limitations

- No Real Payment Gateway
    - Payments are recorded in the database but there is no full integration with real payment providers (e.g., PayPal/Stripe).
    - Future improvement: integrate an actual payment gateway and handle webhooks for confirmation.
- Limited Role System
    - Roles are essentially “user” and “admin”. There is no fine-grained permission system (e.g., event manager vs finance admin).
    - Future improvement: add more roles/permissions and a UI to manage them.
- Basic Reporting

    - Admin dashboard shows counts and simple stats, but reporting and analytics (e.g., charts over time, per-category revenue) are limited.
    - Future improvement: add advanced filters, export to CSV/PDF, and more visual charts.
- No Email Notifications
    - System does not currently send emails for ticket purchase confirmations, password resets, or event updates.
    - Future improvement: integrate email notifications using Laravel mail and queue.
- Mobile Optimization
    - Layout is responsive, but some complex admin tables may still require horizontal scrolling on very small screens.
    - Future improvement: further optimize mobile layouts, especially for admin views.

---

9. Code Quality & Documentation

9.1 Code Structure

```
event_ticketing/
│
├── UI/
│   ├── app/
│   │   ├── layout.tsx
│   │   ├── page.tsx
│   │   ├── auth/
│   │   │   ├── login/page.tsx
│   │   │   └── register/page.tsx
│   │   ├── dashboard/page.tsx
│   │   ├── events/
│   │   │   ├── page.tsx
│   │   │   └── [id]/page.tsx
│   │   ├── tickets/page.tsx
│   │   └── admin/
│   │       ├── events/page.tsx
│   │       ├── users/page.tsx
│   │       └── reports/page.tsx
```

```
│ │
│ ├── components/
│ │ ├── Navbar.tsx
│ │ ├── Footer.tsx
│ │ ├── EventCard.tsx
│ │ ├── TicketCard.tsx
│ │ └── Modal.tsx
│ │
│ ├── lib/
│ │ ├── api.ts
│ │ ├── auth.ts
│ │ └── utils.ts
│ │
│ ├── styles/globals.css
│ └── public/images/
│
├── API/
│ ├── auth/
│ │ ├── login.php
│ │ ├── register.php
│ │ └── logout.php
│ │
│ ├── events/
│ │ ├── create.php
│ │ ├── read.php
│ │ ├── update.php
│ │ └── delete.php
│ │
│ ├── tickets/
│ │ ├── purchase.php
│ │ ├── verify.php
│ │ └── userTickets.php
│ │
│ ├── users/
```

```
│   │   ├── profile.php
│   │   └── update.php
│   │
│   ├── admin/
│   │   ├── dashboard.php
│   │   ├── reports.php
│   │   └── manageUsers.php
│   │
│   ├── middleware/authMiddleware.php
│   └── config/database.php
│
├── database/
│   ├── schema.sql
│   ├── tables.sql
│   ├── indexes.sql
│   ├── views.sql
│   ├── functions.sql
│   ├── procedures.sql
│   └── triggers.sql
│
├── docs/
│   ├── API_DOCUMENTATION.md
├── .env
├── .env.example
├── .gitignore
├── package.json
├── next.config.js
└── README.md
```

9.2 Code Standards

- **Naming Conventions**
    - **Backend (PHP/Laravel):**
        - Classes: PascalCase (e.g., EventController, Ticket, PaymentController).
        - Methods & variables: camelCase (e.g., getDashboardStats, $eventDate).

        - Database columns: snake_case with suffix _id for keys (e.g., user_id, event_id).
    - **Frontend (TypeScript/React):**
        - Components: PascalCase (e.g., LoginPage, EventsPage, LayoutDashboard).
        - Variables & functions: camelCase (e.g., loadEvents, handleLogin).
        - Types & interfaces: PascalCase (e.g., User, Event, Payment).
- **Code Style**
    - **Frontend:**
        - Next.js + TypeScript with ESLint rules (Next defaults) and Prettier-style formatting (consistent quotes, semicolons, indentation).
        - Functional React components and hooks (useState, useEffect, custom hooks).
    - **Backend:**
        - Laravel conventions (controllers in Controllers, models in Models, route definitions in api.php).
        - Use of Eloquent for database access instead of raw SQL.
- **Comments & Documentation**
    - Inline comments mainly where logic is non-obvious (e.g., data transformation, special cases).
    - High-level documentation in project markdown files: API_DOCUMENTATION.md, INTEGRATION_GUIDE.md, INTEGRATION_SUMMARY.md, UI_BACKEND_MAPPING.md.

9.3 API Endpoints (if applicable)

Document all API endpoints if your application has a backend API.

Below is a summary of main API groups (not every single field), matching your system:

- **Authentication**
    - **POST /api/v1/auth/register – Register a new user.**
    - **POST /api/v1/auth/login – Login as normal user and get token.**
    - **POST /api/v1/admin-auth/login – Login as admin and get token.**
    - **POST /api/v1/users/logout – Logout user (invalidate token).**
    - **POST /api/v1/admin/logout – Logout admin.**

    - **GET /api/v1/users/me – Get current logged-in user info.**
    - **GET /api/v1/admin/profile – Get current admin profile.**
- **Events**
    - **GET /api/v1/events – List public events (with search/filter).**
    - **GET /api/v1/events/{id} – Get event details.**
    - **GET /api/v1/admin/events – List all events for admin.**
    - **POST /api/v1/admin/events – Create a new event.**
    - **PUT /api/v1/admin/events/{id} – Update event.**
    - **DELETE /api/v1/admin/events/{id} – Delete event.**
- **Categories**
    - **GET /api/v1/categories – List categories (public).**
    - **GET /api/v1/admin/categories – List categories for admin.**
    - **POST /api/v1/admin/categories – Create category.**
    - **PUT /api/v1/admin/categories/{id} – Update category.**
    - **DELETE /api/v1/admin/categories/{id} – Delete category.**
- **Venues**
    - **GET /api/v1/venues – List venues (public).**
    - **GET /api/v1/admin/venues – List venues for admin.**
    - **POST /api/v1/admin/venues – Create venue.**
    - **PUT /api/v1/admin/venues/{id} – Update venue.**
    - **DELETE /api/v1/admin/venues/{id} – Delete venue.**
- **Tickets**
    - **GET /api/v1/tickets – List tickets for logged-in user.**
    - **POST /api/v1/tickets – Purchase ticket for an event.**
    - **GET /api/v1/tickets/{id} – Get ticket details.**
    - **DELETE /api/v1/tickets/{id} – Cancel ticket.**
    - **(Admin equivalents under /api/v1/admin/tickets for admin view/management.)**
- **Payments**
    - **GET /api/v1/payments – List payments for logged-in user.**

- **POST /api/v1/payments – Create payment for a ticket.**
- **GET /api/v1/payments/{id} – Get payment details.**
- **GET /api/v1/admin/payments – List all payments (admin, with filters).**
- **PUT /api/v1/admin/payments/{id} – Update payment status.**

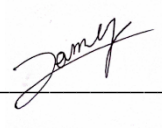
- **Dashboard**
  - **GET /api/v1/admin/dashboard/stats – Summary statistics for admin dashboard.**
  - **GET /api/v1/admin/dashboard/reports – Reports data.**
  - **GET /api/v1/admin/dashboard/analytics – Analytics data (if implemented).**

---

10. References & Resources

List any resources, tutorials, or documentation you used during development.

- Laravel Documentation – https://laravel.com/docs
- Next.js Documentation – https://nextjs.org/docs
- MySQL Documentation – https://dev.mysql.com/doc/
- Tailwind CSS – https://tailwindcss.com/docs
- Mermaid ERD Docs – https://mermaid.js.org/syntax/entityRelationshipDiagram.html
- Postman – https://www.postman.com/ (for testing the API)
- Any Figma design file or UI kit provided by your instructor/team (for layout and styling inspiration).

Student/Team Signature:  **Date:** 22/12/25