Event Manager Design Document

With 4+1 Architectural view

Scenarios (Use Case View)

Actors

- Event Manager (Admin User): Can create, update, and manage events.

- Normal User: Can browse events, book tickets, and make payments.

Use Cases

1. User Registration
   - Admin User Registration: Provides user details and company details.
   - Normal User Registration: Provides user details.
2. Event Management (Admin User)
   - Create Event
   - Update Event
   - View Event Details
3. Event Browsing (Normal User)
   - Search Events
   - View Event Details
4. Ticket Booking (Normal User)
   - Select Event
   - Choose Quantity
   - Book Tickets
5. Payment Processing (Normal User)
   - Make Payment
   - Receive Payment Confirmation

Logical View (Design and Structure)

1. Models (Data Layer)
   - User
   - Company
   - Event
   - Booking
   - Payment

2. Data Access Layer (DAL)
   - UserDAL
   - CompanyDAL
   - EventDAL
   - BookingDAL
   - PaymentDAL

3. REST API Endpoints
   - User Endpoints
   - Event Endpoints
   - Booking Endpoints
   - Payment Endpoints

Relationships

- User can have many Bookings.
- Booking is associated with one Event and one User.
- Event can have many Bookings.
- Payment is associated with one Booking.
- User may belong to one Company (if Admin).

Architectural Patterns

- Model-View-Controller (MVC) Pattern
  - Models: Represent the data and business logic (ORM models using SQLAlchemy).
  - Views: In a web API context, this can be the API responses.
  - Controllers: FastAPI endpoints that handle HTTP requests.
- Repository Pattern (Implemented in DAL)
  - Each DAL class acts as a repository for its respective model, encapsulating data access logic.

Design Patterns

- Singleton Pattern
  - Used for dal layer instance

- Factory Pattern
  - Used for paymet gateway

- Classes:
  - User
    - Attributes: name, email, country_code, phone_number, role, company_id, keycloak_id, username
    - Relationships: belongs to Company, has many Bookings
  - Company
    - Attributes: name, address, email, country_code, phone_number, registration_number
    - Relationships: has many Users
  - Event
    - Attributes: name, event_date, event_time, venue, location_lat, location_long, available_tickets, base_price, surge_price, surge_threshold, version

    - Relationships: has many Bookings

  - Booking

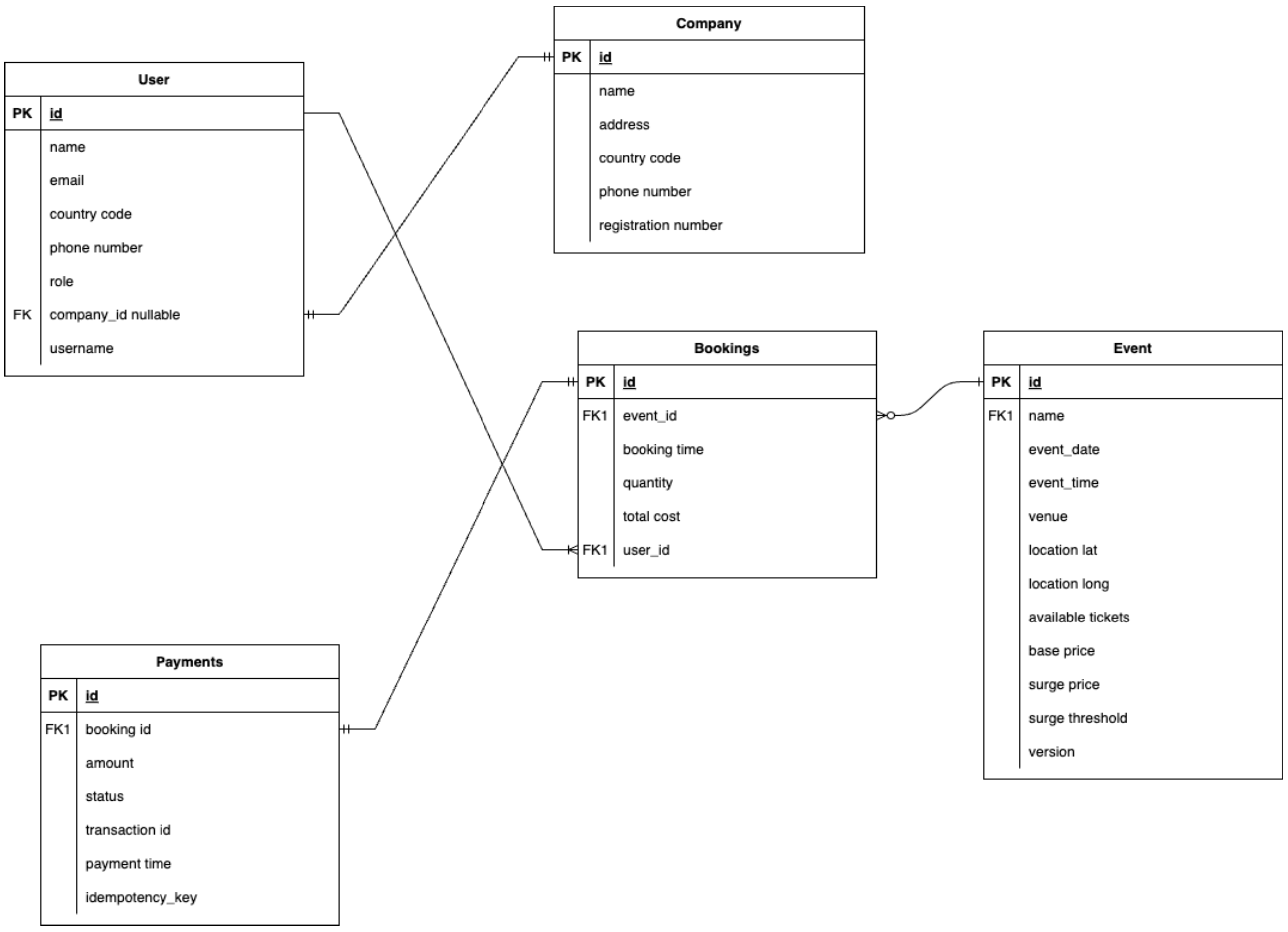
    - Attributes: event_id, user_id, booking_time, quantity, total_cost

    - Relationships: belongs to User, belongs to Event, has many Payments

  - Payment

    - Attributes: booking_id, amount, status, transaction_id, payment_time, idempotency_key

    - Relationships: belongs to Booking



Process View

- Asynchronous Framework: Using FastAPI with async capabilities.

- Asynchronous Database Operations: SQLAlchemy's async ORM is used for non-blocking database access.

- Payment Processing:

  - Payment gateway interactions via webhooks ensures that the application remains responsive during payment transactions.

Inter-Process Communication

- RESTful APIs: Communication between client and server over HTTP/HTTPS.

- External Services:

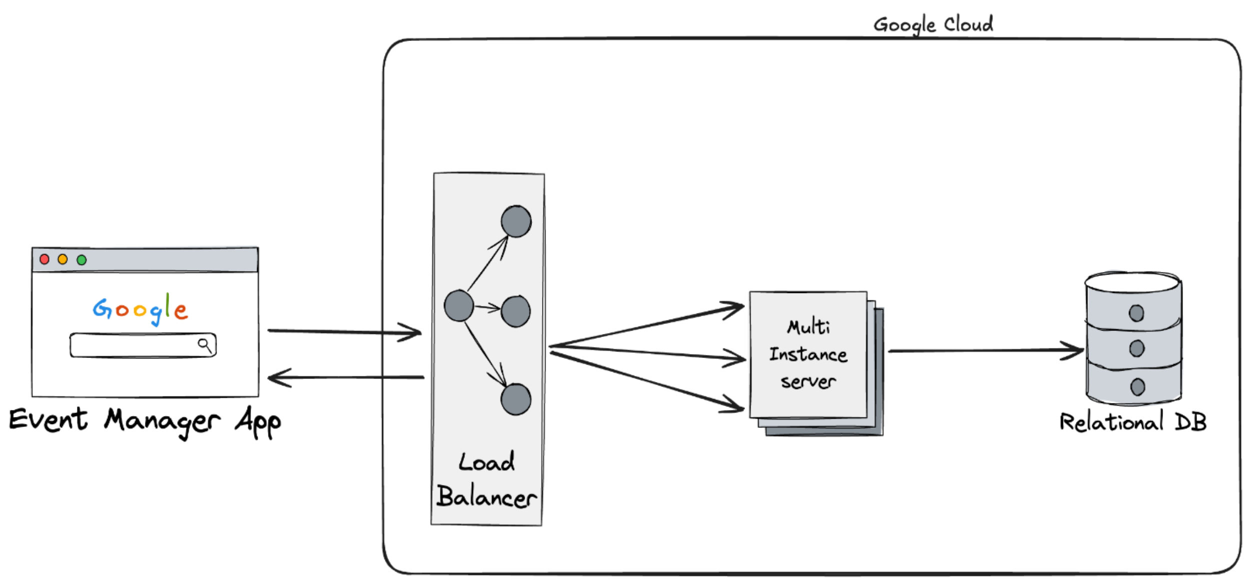
  - Payment Gateway: Communicates with external payment service (i.e., Stripe) over HTTP/HTTPS.

Load Balancing and Scalability

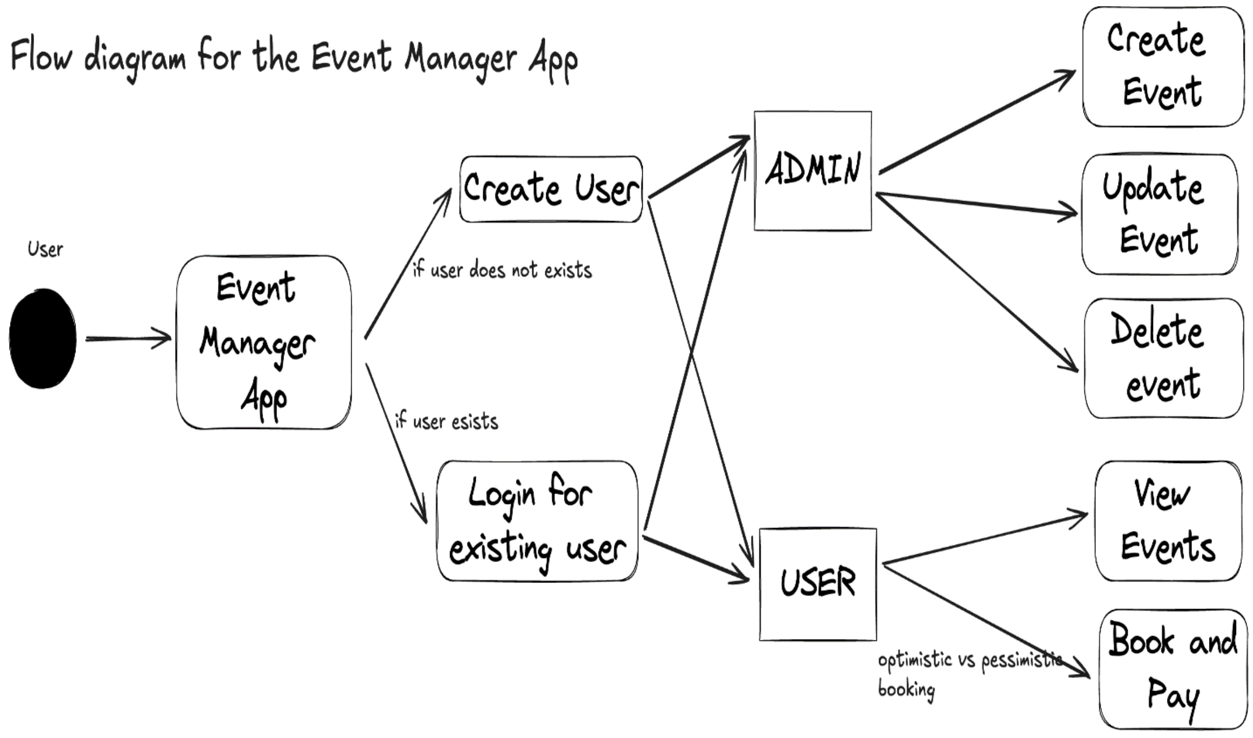
- Load Balancers: Distribute incoming traffic across multiple instances of the application.

- Horizontal Scaling: Multiple instances of the application can run to handle increased load.

Architecture



Flow diagram for the Event Manager App



Development View (Implementation)

Code Organization

- Packages/Folders:
  - event_manager
    - api: Contains FastAPI endpoint definitions.
    - models: SQLAlchemy ORM models.
    - schemas: Pydantic models for data validation.
    - dal: Data Access Layer and business logic.
    - core: Configuration and utility functions.
    - main.py: Application entry point.
    - config.py: Configuration settings.

Technologies and Frameworks

- Programming Language: Python 3.10
- Web Framework: FastAPI
- ORM: SQLAlchemy (with async support)
- Database: PostgreSQL
- Asynchronous Programming: `asyncio`
- IDE: Visual Studio Code

Development Tools

- Package Management: Poetry
- Version Control: Git
- Testing: Pytest, pytest-asyncio
- Linters and Formatters: flake8, black
- Virtual Environments: Managed by Poetry

Design Patterns

- Repository Pattern: Encapsulates data access logic.
- Dependency Injection: Using FastAPI's `Depends` for injecting dependencies.

- Components:
  - API Layer: FastAPI endpoints.
  - Service Layer: Business logic.
  - DAL Layer: Data access repositories.
  - Models: ORM models.
  - Schemas: Data validation models.
  - Database: PostgreSQL.

Physical View (Deployment)

- Platform: Google Cloud Platform (GCP)
- Deployment Methods:
  - Containers: Using Docker for containerization.
  - Virtual Machines: Deployed on GCP Compute Engine VMs.

Infrastructure Components

1. Load Balancer
   - Distributes traffic to multiple instances of the application.
   - Ensures high availability and scalability.

2. Application Instances
   - Docker Containers: Running the FastAPI application.
   - Deployed across multiple VMs or container instances.

3. Database

   - PostgreSQL: Hosted on a managed service or VM.
   - Accessible by application instances within the same network.

4. Payment Gateway

   - External service (i.e., Stripe) for processing payments.
   - Communicates over HTTPS.

- Nodes:
  - Load Balancer
    - Connected to the Internet.
    - Forwards requests to Application Instances.
  - Application Server(s)
    - Docker containers running the FastAPI app.
    - Connected to the Database and Payment Gateway.
  - Database Server
    - PostgreSQL database instance.
  - Payment Gateway
    - External service accessible over the Internet.

6. Cross-Cutting Concerns

- Encryption
  - Communication over HTTPS.
  - Sensitive data encrypted at rest.

Error Handling and Logging

- Exception Handling
  - Custom exceptions for business logic errors.
  - Global exception handlers in FastAPI.
- Logging
  - Standard logging using Python's `logging` module.

Configuration Management

- Environment Variables
  - Sensitive configurations (e.g., database credentials) managed via environment variables.
- Configuration Files
  - Separate configuration files for different environments (development, staging, production).

Architectural Patterns

- Model-View-Controller (MVC)
  - Separates concerns, making the application easier to maintain and scale.
  - Models: Data representations and business logic.
  - Views: API responses.
  - Controllers: API endpoints handling requests.

- Repository Pattern
  - DAL Layer abstracts data access.
  - Promotes loose coupling between the business logic and data storage.

Design Patterns

- Dependency Injection
  - Utilized via FastAPI's `Depends` mechanism.
  - Facilitates testing and modularity.

- Singleton Pattern
  - Applied for configurations or shared resources like database connections.

- Asynchronous Patterns
  - Leveraging async/await for non-blocking I/O operations.
  - Improves performance and scalability under concurrent load.

Enterprise Integration Patterns

- Gateway Pattern
  - The application acts as a gateway to external services like the Payment Gateway.
  - Handles communication and error handling with external APIs.