# Documentation for Ticket Management

## Background

The system aims to create a comprehensive backend solution for managing bus tickets and includes a simple frontend UI to enable users to view available buses, purchase tickets, and interact with the system seamlessly. A frontend will provide a user-friendly interface while leveraging the robust backend for secure and efficient operations.Administrators can manage buses and tickets, while regular users focus on ticket browsing and purchasing. The application will adhere to the **PHP Laravel framework** and utilize **MySQL** for database management, following an MVC architectural pattern to ensure scalability and maintainability.

The core focus is on ensuring secure role-based access, robust ticketing functionalities, and a modular system design that supports future enhancements.

---

## Requirements

The Ticket Management System must fulfill the following requirements, organized using the MoSCoW prioritization method:

*Must-Have:*

- **User Authentication**:
  - User registration, login, logout.
  - Password hashing and JWT-based authentication.
  - Role-based authorization (Admin, User).
- **Admin Functionalities**:
  - **Bus/Vehicle Management**:
    - Add, update, and delete buses/vehicles.
    - Manage vehicle types (e.g., Royal Coach, Deluxe, Super Deluxe).
  - **Ticket and Schedule Management**:
    - Add, update, and delete tickets with specific prices and time slots.
    - Define trip schedules, including pick-up and drop-off points.
    - Create and manage trips based on schedules, vehicles, and ticket prices.
- **User Functionalities**:
  - View available buses and tickets.
  - Purchase tickets for specific buses and time slots.
- **API Development**:
  - Authentication APIs (register, login, logout).
  - Admin APIs (manage buses, schedules, trips, tickets).
  - User APIs (browse buses and tickets, purchase tickets).
- **Database Design**:

    - Clear entity-relationship structure, including schemas for `User`, `Bus`, `VehicleType`, `TripSchedule`, `Trip`, and `Ticket`.
- **Validation and Error Handling**:
    - Input validation and error responses for all API endpoints.
- **Scalability**:
    - Modular pattern to ensure scalability and maintainability.

*Should-Have:*

- Testing:
    - Sample test cases for critical API flows.
    - Ensure APIs are fully functional before deployment.
- Documentation:
    - Postman API documentation with detailed request/response formats.

*Could-Have:*

- Implementation with TypeScript for enhanced type safety (if feasible within PHP).

*Won't-Have (Out of Scope):*

- Frontend UI for ticket management or user operations.
- Real-time ticket booking concurrency handling (beyond basic validations).

---

## Method

*System Architecture*

The Ticket Management System will follow a **Modular MVC (Model-View-Controller)** architecture using PHP and Laravel. The key components are:

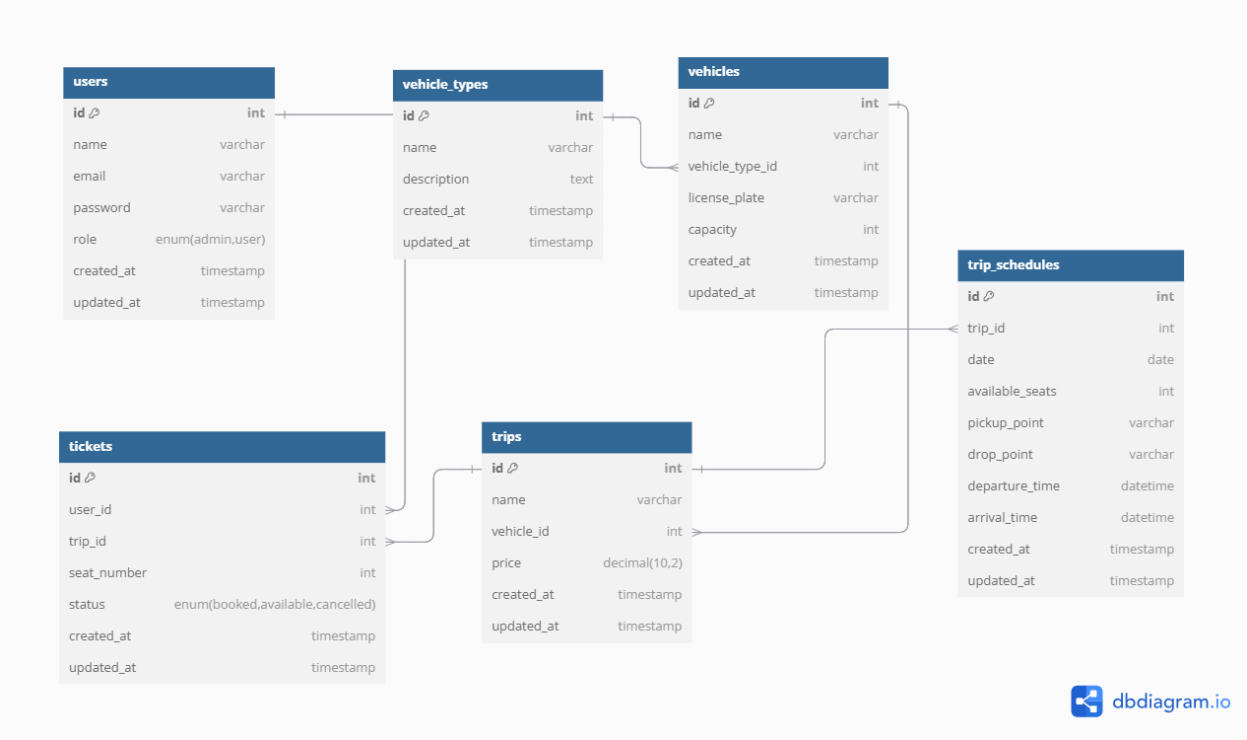
1. **Controllers**:
    - Handle API requests and route them to the appropriate service layer.
    - Separate controllers for better modularity:
        - `AuthController`: Handles authentication-related actions.
        - `BusController`: Manages buses (add, update, delete).
        - `VehicleTypeController`: Manages vehicle types.
        - `TripController`: Manages trips and schedules.
        - `TicketController`: Handles ticket-related actions.
        - `UserController`: Handles user-specific functionalities like profile and booking history.
2. **Models**:
    - Represent data entities and implement business logic.
    - Examples: `User`, `Bus`, `VehicleType`, `TripSchedule`, `Trip`, `Ticket`.
3. **Services**:

- Contain reusable business logic (e.g., trip scheduling, ticket validation).

4. **Database**:
   - Relational database (MySQL) with normalized tables.
   - Includes relations such as:
     - A `Bus` belongs to a `VehicleType`.
     - A `Trip` references a `Bus`, a `TripSchedule`, and associated tickets.
5. **Middleware**:
   - JWT-based authentication.
   - Role-based authorization to restrict admin-only actions.
   - Input validation and logging for sensitive endpoints.
6. **APIs**:
   - RESTful endpoints grouped by controllers to separate concerns.
   - Examples:
     - `/auth`: User authentication.
     - `/admin`: Admin operations.
     - `/user`: User operations.

---

*Database Schema and ER Diagram*




Here is the refined database schema:

```
plaintext
Copy code
Users
```

```
-----
id (PK)
name
email (unique)
password (hashed)
role (ENUM: 'admin', 'user')
created_at
updated_at

VehicleTypes
------------
id (PK)
type_name (e.g., Royal Coach, Deluxe)
created_at
updated_at

Buses
-----
id (PK)
name
vehicle_type_id (FK -> VehicleTypes.id)
capacity
created_at
updated_at

TripSchedules
-------------
id (PK)
departure_time
arrival_time
pickup_point
drop_point
created_at
updated_at

Trips
-----
id (PK)
bus_id (FK -> Buses.id)
trip_schedule_id (FK -> TripSchedules.id)
ticket_price
created_at
updated_at

Tickets
-------
id (PK)
trip_id (FK -> Trips.id)
user_id (FK -> Users.id, NULLABLE for unregistered users)
status (ENUM: 'available', 'booked')
created_at
updated_at
```

---

### *Security Measures*

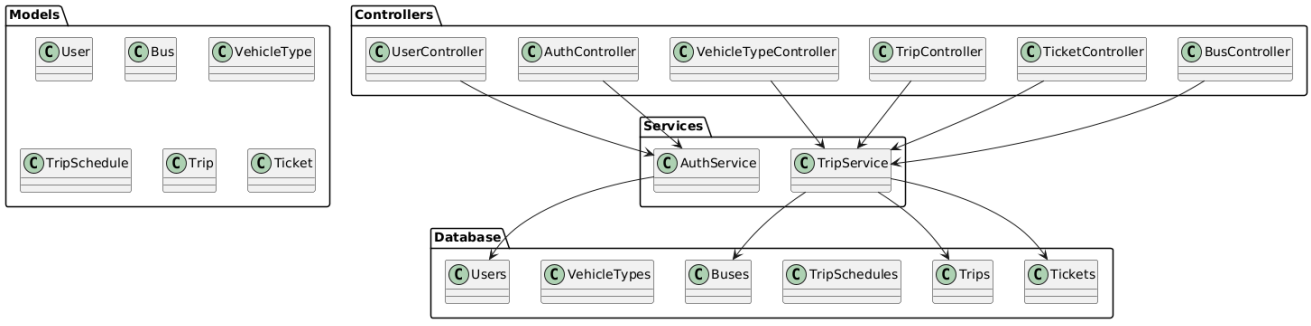
The following measures will secure sensitive operations:

1. **JWT Authentication**:
   - All API endpoints require a valid JWT token.
   - Tokens include role information to enforce access restrictions.
2. **Role-Based Authorization**:
   - Admin APIs (e.g., add bus, manage trips) restricted to users with `admin` role.
3. **Input Validation**:
   - Validate incoming data to prevent injection attacks.
4. **Rate Limiting**:
   - Apply limits to prevent abuse of sensitive APIs.
5. **Audit Logging**:
   - Log critical admin actions (e.g., adding or deleting trips).
6. **HTTPS**:
   - Secure all communication with HTTPS.

---

### *System Diagram*

Here is the high-level component diagram:




```
plantuml
Copy code
@startuml
package "Controllers" {
  class AuthController
  class BusController
  class VehicleTypeController
  class TripController
  class TicketController
  class UserController
}

package "Models" {
  class User
  class Bus
  class VehicleType
```

```
  class TripSchedule
  class Trip
  class Ticket
}

package "Services" {
  class AuthService
  class TripService
}

package "Database" {
  class Users
  class VehicleTypes
  class Buses
  class TripSchedules
  class Trips
  class Tickets
}

AuthController --> AuthService
BusController --> TripService
VehicleTypeController --> TripService
TripController --> TripService
TicketController --> TripService
UserController --> AuthService
AuthService --> Users
TripService --> Buses
TripService --> Trips
TripService --> Tickets
@enduml
```

---

### *API Structure*

Here's how the refined API structure looks:

```
plaintext
Copy code
/api
|-- /auth
|    |-- POST /register (AuthController)
|    |-- POST /login (AuthController)
|    |-- POST /logout (AuthController)
|
|-- /admin
|    |-- /buses
|    |    |-- POST / (BusController) - Add bus
|    |    |-- PUT /:id (BusController) - Update bus
|    |    |-- DELETE /:id (BusController) - Delete bus
|    |-- /vehicle-types
|    |    |-- POST / (VehicleTypeController) - Add vehicle type
|    |    |-- DELETE /:id (VehicleTypeController) - Delete vehicle type
|    |-- /trips
|         |-- POST / (TripController) - Add trip
|         |-- DELETE /:id (TripController) - Delete trip
|
```

```
|-- /user
|    |-- GET /buses (UserController) - View buses
|    |-- GET /tickets (TicketController) - View tickets
|    |-- POST /tickets/purchase (TicketController) - Purchase ticket
```

---

## Implementation

### *Step 1: Setup Development Environment*

1. Install **PHP**, **Laravel**, and a local MySQL database server.
2. Configure the Laravel project:
    - Set up `.env` for database connection (MySQL).
    - Install required packages (e.g., `laravel/passport` for JWT authentication).

### *Step 2: Database Initialization*

1. Define the database schemas:
    - Use Laravel migrations for creating tables (`Users`, `VehicleTypes`, `Buses`, etc.).
    - Set up relationships using Eloquent ORM.
2. Populate reference data:
    - Insert predefined vehicle types (e.g., Royal Coach, Deluxe).

### *Step 3: Build Authentication Module*

1. Create `AuthController`:
    - Implement registration, login, and logout APIs.
    - Use Laravel Passport for JWT token generation and validation.
2. Middleware:
    - Add JWT validation and role-based access middleware.

### *Step 4: Develop Admin Modules*

1. Create `BusController`:
    - Implement APIs for adding, updating, and deleting buses.
2. Create `VehicleTypeController`:
    - Add APIs for managing vehicle types.
3. Create `TripController`:
    - Implement APIs to create and manage trips and schedules.
    - Include ticket price management.

### *Step 5: Develop User Modules*

1. Create `UserController`:
    - Add functionalities for users to view their profile and booking history.
2. Create `TicketController`:

- Implement APIs for viewing available tickets and purchasing tickets.
- Ensure status updates from `available` to `booked` after purchase.

### *Step 6: Implement Validation and Error Handling*

1. Add request validation rules in controllers.
2. Customize error responses to include meaningful messages.

### *Step 7: Secure Sensitive APIs*

1. Apply middleware to secure all APIs:
   - Ensure JWT tokens are validated for every request.
   - Enforce admin-only access for critical APIs.
2. Test security measures:
   - Perform SQL injection, authentication, and rate-limiting tests.

### *Step 8: Testing*

1. Unit Testing:
   - Use Laravel's built-in testing tools to validate models and controllers.
2. API Testing:
   - Use Postman to test all endpoints.
   - Validate authentication, role-based access, and data consistency.
3. Integration Testing:
   - Test end-to-end workflows, such as ticket purchasing.

### *Step 9: Documentation*

1. Generate Postman documentation with example payloads and responses.
2. Write a comprehensive `README.md`:
   - Include instructions for setting up and running the project.
   - Document the database structure and endpoints.

### *Step 10: Deployment*

1. Set up a production environment:
   - Configure a server with PHP, Laravel, and MySQL.
   - Use tools like Docker for containerized deployment (optional).
2. Deploy the application:
   - Push the code to a GitHub repository.
   - Use CI/CD tools (e.g., GitHub Actions) to automate deployment.
3. Secure the production setup:
   - Enable HTTPS and configure firewalls.

---

## Milestones

### *Milestone 1: Project Setup*

- **Timeline**: 1 Day
- **Tasks**:
  - Initialize the Laravel project.
  - Configure the `.env` file for database connection.
  - Install necessary packages (e.g., Laravel Passport).
  - Set up GitHub repository.

### *Milestone 2: Database Design and Implementation*

- **Timeline**: 3 Days
- **Tasks**:
  - Create database tables using Laravel migrations.
  - Define relationships using Eloquent ORM.
  - Seed the database with initial data (e.g., vehicle types).

### *Milestone 3: Authentication Module*

- **Timeline**: 3 Days
- **Tasks**:
  - Implement registration, login, and logout APIs.
  - Configure Laravel Passport for JWT authentication.
  - Add middleware for role-based access control.

### *Milestone 4: Admin Module*

- **Timeline**: 5 Days
- **Tasks**:
  - Develop APIs for managing buses, vehicle types, trips, and schedules.
  - Add validation and error handling for admin endpoints.
  - Test and secure admin APIs.
  - Develop `BusController` for managing buses
  - Implement `VehicleTypeController` for vehicle types.
  - Create `TripController` to handle trip and schedule management.
  - Add validation and middleware for admin-specific APIs

### *Milestone 5: User Module*

- **Timeline**: 4 Days
- **Tasks**:
  - Develop APIs for viewing buses and tickets.
  - Implement ticket purchase workflow with status updates.
  - Test and secure user APIs.
  - Develop `UserController` for user profile and booking history.
  - Implement `TicketController` for ticket browsing and purchasing.
  - Ensure consistent behavior for ticket availability and booking workflows.

### *Milestone 6: Security and Validation (1 Week)*

- Apply validation rules for all endpoints.
- Secure sensitive APIs with rate limiting and input sanitization.
- Test the system for vulnerabilities (e.g., SQL injection).

### *Milestone 6: Testing and Documentation*

- **Timeline**: 3 Days
- **Tasks**:
    - Perform unit and integration testing for all APIs.
    - Create Postman API documentation with detailed examples (requests, responses, sample payloads).
    - Write a comprehensive `README.md` with deployment and usage instructions

### *Milestone 7: Deployment*

- **Timeline**: 2 Days
- **Tasks**:
    - Set up the production environment (server, PHP, MySQL).
    - Deploy the application and secure it with HTTPS.
    - Perform final end-to-end testing on production.

---

## Total Estimated Timeline: 21 Days

---

## Gathering Results

### *Evaluation of Requirements Fulfillment*

To determine if the system meets all defined requirements, the following checklist will be used:

1. **User Authentication**:
    - Confirm that user registration, login, and logout functionality is fully operational.
    - Verify JWT tokens are issued and validated correctly.
    - Test role-based access control for admin and user roles.
2. **Admin Functionalities**:
    - Ensure admins can add, update, and delete buses and vehicle types.
    - Validate that trip schedules and ticket prices are managed correctly.
    - Confirm proper error handling and validation for admin actions.
3. **User Functionalities**:
    - Verify that users can view buses and available tickets.

    - Test ticket purchase workflow, ensuring the status updates from `available` to `booked`.
    - Check that users can view their booking history.
4. **API Functionality**:
    - Test all endpoints for expected responses using Postman.
    - Validate error handling for invalid inputs and unauthorized access.
5. **Database Design**:
    - Confirm the database schema matches the design specifications.
    - Verify data relationships (e.g., trips associated with buses and schedules).
6. **Documentation**:
    - Ensure Postman API documentation is comprehensive.
    - Check that the `README.md` provides clear setup and usage instructions.

---

### *Performance Metrics*

Evaluate system performance using the following criteria:

1. **API Response Times**:
    - Ensure average API response times are under 500ms for common requests.
2. **Scalability**:
    - Test system behavior under simulated load (e.g., concurrent ticket purchases).
3. **Security**:
    - Validate that JWT tokens are secure and cannot be forged.
    - Check that sensitive endpoints are protected by role-based access and rate limiting.
4. **Error Resilience**:
    - Confirm proper error handling for invalid requests and database failures.

---

### *Post-Deployment Validation*

1. Conduct a **User Acceptance Test (UAT)**:
    - Collaborate with stakeholders to validate the system against real-world scenarios.
2. Collect Feedback:
    - Gather feedback from both admin and user testers on functionality and usability.
3. Continuous Monitoring:
    - Set up monitoring for API performance, error rates, and security incidents post-deployment.

---