**Key Functionalities for a Cab Booking System:**

Here’s a quick outline of what your system might include. Let me know if you want to customize or prioritize certain features:

1. **User Management**:
   * Sign-up/Login for customers and drivers.
   * Role-based authentication (e.g., Admin, Customer, Driver).
2. **Cab Management**:
   * Add, remove, and update cab details.
   * View available cabs with real-time locations.
3. **Booking**:
   * Book a ride by entering pickup and drop-off locations.
   * Calculate fare dynamically based on distance and type of cab.
   * Show estimated time of arrival (ETA).
4. **Driver Matching**:
   * Assign the nearest available driver to a booking.
   * Notify the driver and customer of the assignment.
5. **Ride Management**:
   * Track ride status: Pending, In Progress, Completed, or Cancelled.
   * Provide real-time location tracking during a ride.
6. **Payment System**:
   * Integrate payment gateways (e.g., Stripe, PayPal).
   * Allow cash and online payment options.
7. **Admin Panel**:
   * Manage users, cabs, and bookings.
   * Generate reports (e.g., total earnings, active drivers).
8. **Ratings and Feedback**:
   * Allow customers to rate drivers and leave feedback.

**Project Structure:**

1. **Backend Technology**: Spring Boot
   * **Modules**: Spring Web, Spring Data JPA, Spring Security, Spring Validation, Spring Boot Actuator.
   * **Database**: MySQL/PostgreSQL.
   * **ORM**: Hibernate.
2. **Project Layers**:
   * **Controller**: Expose REST APIs for different functionalities.
   * **Service**: Contain the business logic.
   * **Repository**: Handle database operations.
   * **Entities**: Represent the database tables.
3. **APIs**:
   * /users/register (POST) - Register new users.
   * /users/login (POST) - Authenticate users.
   * /cabs (GET) - Get available cabs.
   * /bookings/create (POST) - Book a cab.
   * /bookings/{id}/status (PATCH) - Update booking status.
4. **Tools**:
   * **Build Tool**: Maven or Gradle.
   * **Testing**: JUnit and Mockito for unit and integration tests.
   * **Documentation**: Swagger UI for API documentation.

cab-booking/

├── src/

│ ├── main/

│ │ ├── java/

│ │ │ └── com/

│ │ │ └── cabbooking/

│ │ │ ├── controller/

│ │ │ │ ├── UserController.java

│ │ │ │ ├── CabController.java

│ │ │ │ ├── BookingController.java

│ │ │ │ └── AdminController.java

│ │ │ ├── dto/

│ │ │ │ ├── UserDTO.java

│ │ │ │ ├── BookingRequestDTO.java

│ │ │ │ └── PaymentDTO.java

│ │ │ ├── entity/

│ │ │ │ ├── User.java

│ │ │ │ ├── Cab.java

│ │ │ │ ├── Booking.java

│ │ │ │ └── Payment.java

│ │ │ ├── exception/

│ │ │ │ ├── GlobalExceptionHandler.java

│ │ │ │ ├── UserNotFoundException.java

│ │ │ │ └── CabNotAvailableException.java

│ │ │ ├── repository/

│ │ │ │ ├── UserRepository.java

│ │ │ │ ├── CabRepository.java

│ │ │ │ ├── BookingRepository.java

│ │ │ │ └── PaymentRepository.java

│ │ │ ├── service/

│ │ │ │ ├── UserService.java

│ │ │ │ ├── CabService.java

│ │ │ │ ├── BookingService.java

│ │ │ │ └── PaymentService.java

│ │ │ ├── util/

│ │ │ │ ├── DistanceCalculator.java

│ │ │ │ ├── FareCalculator.java

│ │ │ │ └── NotificationService.java

│ │ │ └── CabBookingApplication.java

│ │ └── resources/

│ │ ├── application.properties

│ │ ├── data.sql

│ │ ├── schema.sql

│ │ └── logback-spring.xml

│ └── test/

│ └── java/

│ └── com/

│ └── cabbooking/

│ ├── UserServiceTest.java

│ ├── CabServiceTest.java

│ └── BookingControllerTest.java

├── .gitignore

├── pom.xml

└── README.md

**Explanation of Key Components**

1. **controller/**:
   * Contains REST controllers that expose APIs for users, cabs, bookings, and admin functionalities.
   * Example: UserController.java manages user-related APIs like login and registration.
2. **dto/**:
   * Data Transfer Objects (DTOs) for transferring data between layers.
   * Example: BookingRequestDTO includes user ID, cab type, pickup, and drop-off locations.
3. **entity/**:
   * JPA entity classes representing database tables.
   * Example: User.java maps to a users table with fields like id, name, and role.
4. **exception/**:
   * Handles application-wide exceptions.
   * Example: GlobalExceptionHandler maps exceptions to meaningful HTTP responses.
5. **repository/**:
   * Spring Data JPA repositories for CRUD operations on the database.
   * Example: UserRepository has methods like findByEmail().
6. **service/**:
   * Contains business logic for user management, cab operations, booking, and payments.
   * Example: BookingService handles cab allocation, fare calculation, and booking status updates.
7. **util/**:
   * Utility classes for reusable components like fare and distance calculation or sending notifications.
8. **resources/**:
   * application.properties: Configuration for database, server, and logging.
   * schema.sql: SQL scripts for schema initialization.
   * data.sql: Initial data for testing purposes.
9. **test/**:
   * Unit and integration tests for services and controllers.

The best approach is to follow a **bottom-up development strategy**, starting with the foundational modules and gradually building on them. Here's the recommended order:

**1. Database Design and entity/ Module**

* **Reason**: All business logic and APIs rely on a well-designed database schema and corresponding entity classes.
* **Action Items**:
  1. Define the tables (e.g., users, cabs, bookings, payments).
  2. Create JPA entity classes like User.java, Cab.java, Booking.java, and Payment.java.
  3. Add relationships between entities (e.g., OneToMany, ManyToOne).

**2. repository/ Module**

* **Reason**: These interfaces allow seamless database operations for CRUD functionality.
* **Action Items**:
  1. Create repositories for all entities (UserRepository, CabRepository, etc.).
  2. Add custom query methods if needed (e.g., findByEmail in UserRepository).

**3. service/ Module**

* **Reason**: The business logic sits here, ensuring the core functionality is implemented before exposing APIs.
* **Action Items**:
  1. Start with the **UserService** (registration, login).
  2. Implement **CabService** (list available cabs, manage cabs).
  3. Implement **BookingService** (allocate cabs, calculate fares).
  4. Add helper methods in the util/ package if needed (e.g., distance or fare calculations).

**4. Controller Layer (controller/)**

* **Reason**: Once the core logic is ready, you can expose it through REST APIs.
* **Action Items**:
  1. Create basic endpoints like user registration (/users/register) and login.
  2. Gradually expand to cab operations (/cabs) and booking APIs (/bookings).

**5. Authentication & Security**

* **Reason**: Adding authentication and role-based access ensures your app is secure.
* **Action Items**:
  1. Use Spring Security to implement JWT-based authentication.
  2. Secure APIs for different roles (e.g., Customer, Driver, Admin).

**6. Testing**

* **Reason**: Ensure your modules work independently and as a system.
* **Action Items**:
  1. Write unit tests for service/ and repository/.
  2. Add integration tests for controller/.

**Database Tables**

1. **users**:
   * Stores information about customers, drivers, and admins.

| **Column Name** | **Data Type** | **Constraints** |
| --- | --- | --- |
| user\_id | INT | Primary Key, Auto Increment |
| name | VARCHAR(100) | NOT NULL |
| email | VARCHAR(100) | UNIQUE, NOT NULL |
| password | VARCHAR(255) | NOT NULL |
| role | ENUM('CUSTOMER', 'DRIVER', 'ADMIN') | NOT NULL |
| phone\_number | VARCHAR(15) | UNIQUE, NOT NULL |
| created\_at | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP |

1. **cabs**:
   * Stores details about cabs in the system.

| **Column Name** | **Data Type** | **Constraints** |
| --- | --- | --- |
| cab\_id | INT | Primary Key, Auto Increment |
| driver\_id | INT | Foreign Key (user\_id) |
| cab\_number | VARCHAR(20) | UNIQUE, NOT NULL |
| cab\_type | ENUM('STANDARD', 'PREMIUM') | NOT NULL |
| is\_available | BOOLEAN | DEFAULT TRUE |
| created\_at | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP |

1. **bookings**:
   * Stores booking information between customers and drivers.

| **Column Name** | **Data Type** | **Constraints** |
| --- | --- | --- |
| booking\_id | INT | Primary Key, Auto Increment |
| customer\_id | INT | Foreign Key (user\_id) |
| cab\_id | INT | Foreign Key (cab\_id) |
| pickup\_location | VARCHAR(255) | NOT NULL |
| dropoff\_location | VARCHAR(255) | NOT NULL |
| fare | DECIMAL(10,2) | NOT NULL |
| status | ENUM('PENDING', 'IN\_PROGRESS', 'COMPLETED', 'CANCELLED') | DEFAULT 'PENDING' |
| booking\_time | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP |

1. **payments**:
   * Stores payment details for bookings.

| **Column Name** | **Data Type** | **Constraints** |
| --- | --- | --- |
| payment\_id | INT | Primary Key, Auto Increment |
| booking\_id | INT | Foreign Key (booking\_id) |
| payment\_method | ENUM('CASH', 'CARD', 'ONLINE') | NOT NULL |
| amount | DECIMAL(10,2) | NOT NULL |
| payment\_time | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP |

1. **ride\_locations** (Optional, for tracking live ride locations):
   * Tracks real-time ride location updates.

| **Column Name** | **Data Type** | **Constraints** |
| --- | --- | --- |
| location\_id | INT | Primary Key, Auto Increment |
| booking\_id | INT | Foreign Key (booking\_id) |
| latitude | DECIMAL(9,6) | NOT NULL |
| longitude | DECIMAL(9,6) | NOT NULL |
| timestamp | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP |

**Relationships:**

1. **users ↔ cabs**:
   * One driver (user) is assigned to one cab.  
     **Relationship**: @OneToOne
2. **users ↔ bookings**:
   * One customer can have multiple bookings.  
     **Relationship**: @OneToMany
3. **cabs ↔ bookings**:
   * One cab can serve multiple bookings.  
     **Relationship**: @OneToMany
4. **bookings ↔ payments**:
   * One booking can have one payment.  
     **Relationship**: @OneToOne

**Entity Relationship Diagram (ERD)**

The database schema can be visualized as follows:

[users] <--> [cabs] <--> [bookings] <--> [payments]

**Core User APIs**

These APIs handle essential operations for users in the system.

| **API** | **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- | --- |
| **Register User** | POST | /users/register | Registers a new user (customer, driver, or admin). |
| **Get User by ID** | GET | /users/{id} | Retrieves details of a user by their unique ID. |
| **Get All Drivers** | GET | /users/drivers | Retrieves a list of all users with the role DRIVER. |
| **Update User** | PUT | /users/{id} | Updates user information (e.g., name, email, password). |
| **Delete User** | DELETE | /users/{id} | Deletes a user account by ID (soft delete or permanent delete, based on system requirements). |

**Authentication & Authorization APIs**

These APIs handle user login, logout, and role-based security.

| **API** | **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- | --- |
| **Login User** | POST | /users/login | Authenticates a user and returns a JWT token. |
| **Logout User** | POST | /users/logout | Invalidates the user session or JWT token. |
| **Change Password** | PUT | /users/{id}/password | Allows the user to update their password. |

**Role-Specific APIs**

APIs for admin, drivers, or customers to perform role-specific actions.

| **API** | **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- | --- |
| **Get All Customers** | GET | /users/customers | Retrieves a list of all users with the role CUSTOMER. |
| **Assign Role** (Admin Only) | PUT | /users/{id}/role | Updates the role of a user (e.g., promote a customer to driver or admin). |

**Other APIs**

Additional APIs for enhancing user functionality.

| **API** | **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- | --- |
| **Search Users** | GET | /users/search | Search users by name, email, or phone number. |
| **Deactivate User** | PUT | /users/{id}/deactivate | Marks a user as inactive without permanently deleting their account. |

**Approach:**

1. **Phase 1**: Implement **Register**, **Get User by ID**, and **Get All Drivers** APIs.
2. **Phase 2**: Add **Login**, **Logout**, and **Change Password** for authentication.
3. **Phase 3**: Enhance the module with admin-only APIs like **Assign Role** and **Delete User**.

**Cab Module:**

The **Cab Module** will handle the booking and management of cabs, along with related functionalities such as:

1. **Cab Entity** (for storing cab details).
2. **Cab Booking** (to create and manage bookings).
3. **Cab Availability** (check if a cab is available).
4. **Cab Driver Assignment** (assign a driver to a booking).
5. the **Cab Module** API details in the requested format:

### ****Cab Entity APIs****

| **API** | **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- | --- |
| Add New Cab | POST | /cabs | Adds a new cab to the system. |
| Update Cab Details | PUT | /cabs/{id} | Updates details of an existing cab. |
| Get Cab by ID | GET | /cabs/{id} | Retrieves details of a specific cab. |
| Get All Cabs | GET | /cabs | Retrieves a list of all available cabs. |

### ****Cab Booking APIs****

| **API** | **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- | --- |
| Book a Cab | POST | /bookings | Allows a customer to book a cab. |
| Get Booking by ID | GET | /bookings/{id} | Retrieves details of a specific booking. |
| Get All Bookings for User | GET | /users/{userId}/bookings | Retrieves all bookings made by a specific user. |
| Update Booking Status | PATCH | /bookings/{id}/status | Updates the status of a booking (e.g., PENDING to COMPLETED). |

### ****Cab Availability APIs****

| **API** | **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- | --- |
| Check Cab Availability | GET | /cabs/{id}/availability | Checks if a specific cab is available for booking. |
| Set Cab Availability | PATCH | /cabs/{id}/availability | Updates the availability status of a cab. |

### ****Driver Assignment APIs****

| **API** | **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- | --- |
| Assign Driver to Cab | POST | /cabs/{cabId}/assign-driver | Assigns a driver to a specific cab. |

1. Let me know if you'd like to proceed with any specific API implementation or if you'd like to add any further details! 🚗

# **Booking Module - API Documentation**

## **Overview**

The Booking Module handles the process of booking a cab, updating its status, and retrieving booking details. It ensures that only available cabs are booked and manages booking status updates.

## **Entity Schema**

### ****Booking Table****

| **Column Name** | **Data Type** | **Constraints** |
| --- | --- | --- |
| booking\_id | INT | Primary Key, Auto Increment |
| customer\_id | INT | Foreign Key (users.user\_id), NOT NULL |
| cab\_id | INT | Foreign Key (cabs.cab\_id), NOT NULL |
| pickup\_location | VARCHAR(255) | NOT NULL |
| dropoff\_location | VARCHAR(255) | NOT NULL |
| fare | DECIMAL(10,2) | NOT NULL |
| status | ENUM | ['PENDING', 'CONFIRMED', 'CANCELLED', 'COMPLETED'], Default: 'PENDING' |
| booking\_time | TIMESTAMP | Default: CURRENT\_TIMESTAMP, NOT NULL |

## **API Endpoints**

| **API** | **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- | --- |
| **Create Booking** | POST | /bookings | Allows a customer to book a cab. |
| **Get Booking by ID** | GET | /bookings/{id} | Fetches booking details using the booking ID. |
| **Update Booking Status** | PATCH | /bookings/{id}/status | Updates the status of a booking (PENDING, CONFIRMED, CANCELLED, COMPLETED). |
| **Get Customer Bookings** | GET | /bookings/customer/{customerId} | Fetches all bookings made by a specific customer. |

## **Request & Response Details**

### ****1. Create Booking****

* **Endpoint:** POST /bookings
* **Request Body (JSON):**

json

CopyEdit

{

"customerId": 1,

"cabId": 10,

"pickupLocation": "Airport",

"dropoffLocation": "City Center",

"fare": 25.50

}

* **Response Body (JSON):**

json

CopyEdit

{

"bookingId": 101,

"customerId": 1,

"cabId": 10,

"pickupLocation": "Airport",

"dropoffLocation": "City Center",

"fare": 25.50,

"status": "PENDING",

"bookingTime": "2025-02-02T10:15:30"

}

### ****2. Get Booking by ID****

* **Endpoint:** GET /bookings/{id}
* **Response Body (JSON):**

json

CopyEdit

{

"bookingId": 101,

"customerId": 1,

"cabId": 10,

"pickupLocation": "Airport",

"dropoffLocation": "City Center",

"fare": 25.50,

"status": "CONFIRMED",

"bookingTime": "2025-02-02T10:15:30"

}

### ****3. Update Booking Status****

* **Endpoint:** PATCH /bookings/{id}/status
* **Request Body (JSON):**

json

CopyEdit

{

"status": "COMPLETED"

}

* **Response Body (JSON):**

json

CopyEdit

{

"bookingId": 101,

"customerId": 1,

"cabId": 10,

"pickupLocation": "Airport",

"dropoffLocation": "City Center",

"fare": 25.50,

"status": "COMPLETED",

"bookingTime": "2025-02-02T10:15:30"

}

### ****4. Get Customer Bookings****

* **Endpoint:** GET /bookings/customer/{customerId}
* **Response Body (JSON):**

json

CopyEdit

[

{

"bookingId": 101,

"customerId": 1,

"cabId": 10,

"pickupLocation": "Airport",

"dropoffLocation": "City Center",

"fare": 25.50,

"status": "COMPLETED",

"bookingTime": "2025-02-02T10:15:30"

},

{

"bookingId": 102,

"customerId": 1,

"cabId": 12,

"pickupLocation": "Mall",

"dropoffLocation": "Hotel",

"fare": 30.75,

"status": "PENDING",

"bookingTime": "2025-02-02T12:00:00"

}

]

## **Business Logic**

1. **Booking Creation**
   * The system checks if the **customer** exists.
   * The system checks if the **cab is available**.
   * If valid, a new **booking entry** is created.
   * The **cab’s availability** is set to **false** (unavailable).
2. **Booking Status Update**
   * Only valid statuses (PENDING, CONFIRMED, CANCELLED, COMPLETED) are accepted.
   * If a booking is **CANCELLED** or **COMPLETED**, the **cab is marked as available** again.
3. **Fetching Bookings**
   * Customers can view their **past and ongoing bookings**.
   * Admins can fetch **all bookings** for monitoring.

## **Validation Rules**

| **Field** | **Validation** |
| --- | --- |
| customerId | Must be a valid customer ID, cannot be null. |
| cabId | Must be a valid cab ID, cannot be null. |
| pickupLocation | Must be between 3 to 255 characters. |
| dropoffLocation | Must be between 3 to 255 characters. |
| fare | Must be a positive value (> 1). |
| status | Must be one of PENDING, CONFIRMED, CANCELLED, COMPLETED. |

## **Error Handling**

| **Scenario** | **Response Code** | **Message** |
| --- | --- | --- |
| Booking created successfully | 201 CREATED | Returns booking details. |
| Booking not found | 404 NOT FOUND | "Booking not found." |
| Cab not available | 400 BAD REQUEST | "Cab is not available for booking." |
| Invalid status update | 400 BAD REQUEST | "Invalid booking status." |

# **Payment Module - API Documentation**

## **Overview**

The Payment Module is responsible for handling fare payments, transaction history, and invoice generation. It ensures that customers can pay securely and track their payments.

## **Entity Schema**

### ****Payment Table****

| **Column Name** | **Data Type** | **Constraints** |
| --- | --- | --- |
| payment\_id | INT | Primary Key, Auto Increment |
| booking\_id | INT | Foreign Key (bookings.booking\_id), NOT NULL |
| customer\_id | INT | Foreign Key (users.user\_id), NOT NULL |
| amount | DECIMAL(10,2) | NOT NULL |
| payment\_method | ENUM | ['CASH', 'CARD', 'WALLET', 'UPI'], NOT NULL |
| payment\_status | ENUM | ['PENDING', 'COMPLETED', 'FAILED'], Default: 'PENDING' |
| transaction\_id | VARCHAR(50) | UNIQUE, Nullable |
| payment\_time | TIMESTAMP | Default: CURRENT\_TIMESTAMP |

## **API Endpoints**

| **API** | **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- | --- |
| **Initiate Payment** | POST | /payments | Starts a payment process for a completed booking. |
| **Get Payment Details** | GET | /payments/{id} | Fetches payment details by payment ID. |
| **Get Payment History for Customer** | GET | /payments/customer/{customerId} | Retrieves all payment transactions for a specific customer. |
| **Update Payment Status** | PATCH | /payments/{id}/status | Updates the status of a payment (PENDING, COMPLETED, FAILED). |

## **Request & Response Details**

### ****1. Initiate Payment****

* **Endpoint:** POST /payments
* **Request Body (JSON):**

json

CopyEdit

{

"bookingId": 101,

"customerId": 1,

"amount": 25.50,

"paymentMethod": "CARD"

}

* **Response Body (JSON):**

json

CopyEdit

{

"paymentId": 201,

"bookingId": 101,

"customerId": 1,

"amount": 25.50,

"paymentMethod": "CARD",

"paymentStatus": "PENDING",

"transactionId": "TRX12345",

"paymentTime": "2025-02-02T14:30:00"

}

### ****2. Get Payment Details****

* **Endpoint:** GET /payments/{id}
* **Response Body (JSON):**

json

CopyEdit

{

"paymentId": 201,

"bookingId": 101,

"customerId": 1,

"amount": 25.50,

"paymentMethod": "CARD",

"paymentStatus": "COMPLETED",

"transactionId": "TRX12345",

"paymentTime": "2025-02-02T14:30:00"

}

### ****3. Get Payment History for Customer****

* **Endpoint:** GET /payments/customer/{customerId}
* **Response Body (JSON):**

json

CopyEdit

[

{

"paymentId": 201,

"bookingId": 101,

"customerId": 1,

"amount": 25.50,

"paymentMethod": "CARD",

"paymentStatus": "COMPLETED",

"transactionId": "TRX12345",

"paymentTime": "2025-02-02T14:30:00"

},

{

"paymentId": 202,

"bookingId": 103,

"customerId": 1,

"amount": 30.75,

"paymentMethod": "UPI",

"paymentStatus": "PENDING",

"transactionId": "TRX12346",

"paymentTime": "2025-02-02T16:15:00"

}

]

### ****4. Update Payment Status****

* **Endpoint:** PATCH /payments/{id}/status
* **Request Body (JSON):**

json

CopyEdit

{

"paymentStatus": "COMPLETED"

}

* **Response Body (JSON):**

json

CopyEdit

{

"paymentId": 201,

"bookingId": 101,

"customerId": 1,

"amount": 25.50,

"paymentMethod": "CARD",

"paymentStatus": "COMPLETED",

"transactionId": "TRX12345",

"paymentTime": "2025-02-02T14:30:00"

}

## **Business Logic**

1. **Payment Initiation**
   * A payment request is created only if the **booking is completed**.
   * The **amount is fetched from the booking fare**.
   * A **unique transaction ID** is generated for tracking.
2. **Payment Status Update**
   * Status can be updated to **COMPLETED** or **FAILED**.
   * If **FAILED**, the customer needs to retry.
3. **Fetching Payment Details**
   * Customers can view **their past payments**.
   * Admins can track **all payments** for auditing.

## **Validation Rules**

| **Field** | **Validation** |
| --- | --- |
| bookingId | Must be a valid booking ID, cannot be null. |
| customerId | Must be a valid customer ID, cannot be null. |
| amount | Must be a positive value (> 1). |
| paymentMethod | Must be one of CASH, CARD, WALLET, UPI. |
| paymentStatus | Must be one of PENDING, COMPLETED, FAILED. |

## **Error Handling**

| **Scenario** | **Response Code** | **Message** |
| --- | --- | --- |
| Payment initiated successfully | 201 CREATED | Returns payment details. |
| Payment not found | 404 NOT FOUND | "Payment not found." |
| Booking not completed | 400 BAD REQUEST | "Cannot initiate payment before completing the ride." |
| Invalid payment status update | 400 BAD REQUEST | "Invalid payment status." |