Booking Management Service Design Specification

Enterprise Architecture Team

September 10, 2025

Contents

1 Introduction

1.1 System Context

The Booking Management Service (BMS) is a critical component of our on-demand logistics platform, responsible for managing the entire lifecycle of transportation bookings. Positioned at the core of our microservices architecture, the BMS orchestrates the complex interactions between service providers, users, and supporting systems.

1.2 Service Overview

1.2.1 BookingLifecycleManager

- Manages the complete lifecycle of a booking, from creation to completion.
- Handles state transitions (e.g., `Created`, `Confirmed`, `Started`, `Completed`, `Cancelled`).
- Publishes relevant Kafka events (e.g., `Booking.V1.Created`, `Booking.V1.Completed`).
- Coordinates with external systems (e.g., Provider Matching, Transport Management).

1.2.2 PolicyEnforcer

- Validates that booking requests comply with defined business policies (e.g., user eligibility, location restrictions, cargo limitations).
- Enforces SLA compliance and service-level agreements during the booking lifecycle.
- Integrates with the SLAManager for SLA monitoring and reporting.

1.2.3 SLAManager

- Tracks service-level agreement (SLA) metrics for bookings.
- Provides real-time monitoring of SLA compliance (e.g., response times, delivery timeframes).
- Issues alerts for SLA violations and publishes metrics for downstream analysis.

1.2.4 PaymentCoordinator

- Handles all payment-related workflows, including pre-booking authorizations, payment processing, and refunds.
- Publishes and consumes payment-related Kafka events (e.g., `Payment.V1.Successful`, `Payment.V1.Failed`).
- Ensures idempotency in payment processing to prevent duplicate charges.

1.2.5 Partial Fulfillment

- Supports splitting large requests across multiple providers if capacity is insufficient.

1.3 Key Design Principles

The design of the Booking Management Service is guided by the following architectural principles:

1. **Event-Driven Architecture:** Utilize event-driven patterns to ensure loose coupling and scalability.

2. **Eventual Consistency:** Embrace eventual consistency model for improved performance and resilience.

3. **Domain-Driven Design:** Align technical implementation closely with business domain logic.

4. **Transactional Boundary:** Maintain clear service boundaries with robust error handling.

2 API Contracts

This section outlines the API design for the Booking Management Service, focusing on endpoints, descriptions, functionalities, and associated JSON schemas. It includes integrations with the Transport Request Service for transport coordination and other services for payment and notifications.

3 API Contracts

This section outlines the updated API design for the Booking Management Service, focusing on its endpoints, functionalities, and associated JSON schemas. It emphasizes separation of concerns, particularly for interactions with the Transport Request Service (TRS).

3.1 API Endpoints

The following table summarizes the endpoints provided by the Booking Management Service:

3.2 Endpoint Details with JSON Schemas

3.2.1 Create a New Booking

This endpoint creates a new booking. It validates and persists the booking details and interacts with the TRS to generate transport requests.

Endpoint: `POST /api/v1/bookings`
Request Schema:

Table 1: Booking Management Service Endpoints

Method	Endpoint	Description	Auth Scope
POST	`/api/v1/bookings`	Create a new booking and interact with TRS for transport requests.	`booking:write`
GET	`/api/v1/bookings/{id}`	Retrieve booking details by ID.	`booking:read`
GET	`/api/v1/bookings`	List or search bookings with filters.	`booking:read`
PATCH	`/api/v1/bookings/{id}`	Update booking details by ID.	`booking:write`
DELETE	`/api/v1/bookings/{id}`	Cancel a booking and notify TRS.	`booking:write`
POST	`/api/v1/bookings/{id}/status`	Update the status of a booking (e.g., `Started`, `Completed`).	`booking:execute`
GET	`/api/v1/bookings/{id}/history`	Retrieve the state change history of a booking.	`booking:read`
GET	`/api/v1/bookings/health`	Check the health of the Booking Management Service.	`monitoring:read`

Listing 1: Create Booking Request

```
{
    "user_id": "string",
    "scheduled_time": "string (ISO 8601 timestamp)",
    "additional_requirements": {
        "vehicle_type": "string",
        "special_notes": "string"
    }
}
```

Response Schema:

Listing 2: Create Booking Response

```
{
    "booking_id": "string",
    "status": "string",
    "created_at": "string (ISO 8601 timestamp)"
}
```

Interaction with TRS: - Upon receiving this request, the Booking Management Service forwards transport-related details (e.g., ‘pickup_location‘, ‘dropoff_location‘, etc.) to the TRS via an API call. - The TRS handles geospatial and route planning responsibilities and returns the transport request ID, which is linked to the booking.

—

3.2.2 Retrieve Booking Details

This endpoint retrieves the details of a specific booking.

Endpoint: `GET /api/v1/bookings/{id}`
Response Schema:

Listing 3: Retrieve Booking Response

```
{
    "booking_id": "string",
    "user_id": "string",
    "status": "string",
    "scheduled_time": "string (ISO 8601 timestamp)",
    "created_at": "string (ISO 8601 timestamp)",
    "transport_request_id": "string"
}
```

Note: - The response includes a reference to the associated ‘transport_request_id‘, allowing clients to query the TRS for transport-specific details.

—

3.2.3 List or Search Bookings

This endpoint lists or searches bookings based on filters.

Endpoint: `GET /api/v1/bookings`
Request Schema:

Listing 4: List Bookings Request

```
{
    "status": "string (optional)",
    "start_date": "string (ISO 8601 timestamp, optional)",
    "end_date": "string (ISO 8601 timestamp, optional)",
    "user_id": "string (optional)"
}
```

Response Schema:

Listing 5: List Bookings Response

```
{
    "bookings": [
        {
            "booking_id": "string",
            "user_id": "string",
            "status": "string",
            "scheduled_time": "string (ISO 8601 timestamp)",
            "created_at": "string (ISO 8601 timestamp)"
        }
    ],
    "pagination": {
        "total": "integer",
        "page": "integer",
        "page_size": "integer"
    }
}
```

—

3.2.4 Cancel a Booking

This endpoint cancels an existing booking and notifies the TRS to cancel the associated transport request.

Endpoint: `DELETE /api/v1/bookings/{id}`
Response Schema:

Listing 6: Cancel Booking Response

```
{
    "booking_id": "string",
    "status": "Canceled",
    "updated_at": "string (ISO 8601 timestamp)"
}
```

Interaction with TRS: - The Booking Management Service sends a cancellation request to the TRS for the associated ‘transport_request_id‘.

—

3.2.5 Update Booking Status

This endpoint updates the status of a booking (e.g., `Started`, `Completed`).

Endpoint: `POST /api/v1/bookings/{id}/status`
Request Schema:

Listing 7: Update Booking Status Request

```
{
    "status": "string"
}
```

Response Schema:

Listing 8: Update Booking Status Response

```
{
    "booking_id": "string",
    "status": "string",
    "updated_at": "string (ISO 8601 timestamp)"
}
```

Note: - Status updates for bookings are propagated to the TRS if relevant.
—

3.2.6 Retrieve Booking History

This endpoint retrieves the history of status changes for a booking.

Endpoint: `GET /api/v1/bookings/{id}/history`
Response Schema:

Listing 9: Retrieve Booking History Response

```
{
    "history": [
        {
            "status": "string",
            "timestamp": "string (ISO 8601 timestamp)"
        }
    ]
}
```

—

3.2.7 Health Check

This endpoint checks the health of the Booking Management Service.

Endpoint: `GET /api/v1/bookings/health`
Response Schema:

Listing 10: Health Check Response

```
{
    "status": "Healthy",
    "timestamp": "string (ISO 8601 timestamp)"
}
```

—

3.3 Integration with TRS

The Booking Management Service interacts with the TRS for:

- Creating transport requests when a booking is created.
- Updating or canceling transport requests when booking details change.
- Querying transport request details for reporting purposes.

3.4 Interactions with Other Services

- **Transport Request Service**: The Booking Management Service delegates transport-related workflows, such as vehicle matching and route optimization, to the Transport Request Service. This is initiated through the '/finalize' endpoint and monitored via the 'transport_request_id'.

- **Notification Service**: After booking creation, status updates, or cancellations, the Notification Service is informed for user communication.

- **Payment Service**: If payment is required for booking confirmation, the Booking Management Service integrates with the Payment Service via events.

- **Kafka Integration**: Booking lifecycle events (e.g., created, updated, cancelled) are published to Kafka for other services to consume.

3.5 Error Handling

Standardized error codes and messages:

```
{
    "code": "BOOKING_NOT_FOUND",
    "message": "The requested booking does not exist."
}
```

Key error scenarios: - `BOOKING_NOT_FOUND`: Booking ID does not exist. - `INVALID_PARAMETERS`: Booking parameters fail validation. - `TRANSPORT_SERVICE_UNAVAILABLE`: Failure to communicate with the Transport Request Service.

4 Data Models

This section defines the data models for the Booking Management Service, including comprehensive schemas for goods transportation, enumerations, and validation rules. The models ensure consistency and validation across the API.

4.1 Core Data Models

Table 2: Core Data Models

Model Name	Description
Booking	Represents a booking entity, including transport and goods details.
Transport Details	Specifies the characteristics of the goods being transported.
Pickup and Delivery Details	Represents locations and contact information for pickup and delivery.
Metadata	Optional additional information for a booking, such as SLA or special notes.

4.2 Schema Definitions

4.2.1 Booking Model

The Booking model captures core booking details, including transport specifications and requirements.
Schema:

```
{
    "booking_id": "string",
    "user_id": "string",
    "status": "string (enum: ['Created', 'Confirmed', 'Cancelled
       ', 'Started', 'Completed'])",
    "transport_details": {
        "$ref": "#/definitions/TransportDetails"
    },
    "pickup_details": {
        "$ref": "#/definitions/PickupDetails"
    },
    "delivery_details": {
        "$ref": "#/definitions/DeliveryDetails"
    },
    "created_at": "string (ISO 8601 timestamp)",
    "updated_at": "string (ISO 8601 timestamp)"
}
```

4.2.2 Transport Details Model

Specifies the goods being transported and their attributes.
Schema:

```
{
    "goods_type": "string (enum: ['FURNITURE', 'ELECTRONICS', '
       GROCERIES', ...])",
    "cargo_category": "string (enum: ['SMALL_PARCEL', 'LARGE_ITEM
       ', ...])",
    "weight": {
        "value": "number (range: 0.1 - 5000)",
        "unit": "string (enum: ['kg', 'lbs'])"
    },
    "dimensions": {
        "length": "number (max: 6.0)",
        "width": "number (max: 2.5)",
        "height": "number (max: 2.5)",
        "unit": "string (enum: ['m', 'cm', 'inches'])"
    },
    "quantity": "number (minimum: 1)",
    "packaging_type": "string (enum: ['CARDBOARD_BOX', '
       PLASTIC_CONTAINER', ...])"
}
```

4.2.3 Pickup and Delivery Details Models

These models represent location, contact information, and scheduling for pickup and delivery.

Schema: Pickup Details

```
{
    "location": {
        "address": "string",
        "latitude": "number (range: -90 to 90)",
        "longitude": "number (range: -180 to 180)",
        "contact_name": "string",
        "contact_phone": "string (regex: '^\\+?[0-9]{7,15}$')",
        "instructions": "string (optional)"
    },
    "scheduled_pickup_time": "string (ISO 8601 timestamp)"
}
```

Schema: Delivery Details

```
{
    "location": {
        "address": "string",
        "latitude": "number (range: -90 to 90)",
        "longitude": "number (range: -180 to 180)",
        "contact_name": "string",
        "contact_phone": "string (regex: '^\\+?[0-9]{7,15}$')",
        "instructions": "string (optional)"
    },
    "scheduled_delivery_time": "string (ISO 8601 timestamp)"
}
```

4.2.4 Enumerations

Goods Type Enumeration Schema:

```
{
    "goods_type_options": [
        "FURNITURE",
        "ELECTRONICS",
        "GROCERIES",
        "CONSTRUCTION_MATERIALS",
        "MEDICAL_SUPPLIES",
        "PERISHABLES",
        "INDUSTRIAL_EQUIPMENT",
        "PERSONAL_ITEMS",
        "DOCUMENTS",
        "OTHER"
    ]
}
```

Cargo Category Enumeration Schema:

```
{
    "cargo_category_options": [
        "SMALL_PARCEL",
        "LARGE_ITEM",
        "BULK_SHIPMENT",
        "PALLETIZED",
        "CONTAINER",
        "REFRIGERATED"
    ]
}
```

Packaging Type Enumeration Schema:

```
{
    "packaging_type_options": [
        "CARDBOARD_BOX",
        "PLASTIC_CONTAINER",
        "WOODEN_CRATE",
        "PALLET",
        "VACUUM_SEALED",
        "REFRIGERATED_CONTAINER",
        "BUBBLE_WRAP",
        "CUSTOM_PACKAGING"
    ]
}
```

4.3 Validation Rules and Constraints

Weight Constraints

- Minimum weight: 0.1 kg
- Maximum weight: 5000 kg
- Supported units: `["kg", "lbs"]`

Dimensions Constraints

- Minimum dimension: 0.01 m
- Maximum dimensions:
  - Length: 6 m
  - Width: 2.5 m
  - Height: 2.5 m
- Supported units: `["m", "cm", "inches"]`

Special Handling Rules **Schema:**

```
{
    "special_handling_rules": {
        "temperature_controlled": {
            "required_for": ["PERISHABLES", "MEDICAL_SUPPLIES"],
            "temperature_range": {
                "min": -20,
                "max": 15,
                "unit": "celsius"
            }
        },
        "fragile_items": {
            "requires_extra_padding": true,
            "max_weight": 50,
            "additional_insurance_required": true
        },
        "hazardous_materials": {
            "requires_special_permit": true,
            "maximum_quantity": 100,
            "special_packaging_required": true
        }
    }
}
```

4.4 Examples of Payloads

Example: Furniture Transportation

```
{
    "goods_type": "FURNITURE",
    "cargo_category": "LARGE_ITEM",
    "weight": {
        "value": 150,
        "unit": "kg"
    },
    "dimensions": {
        "length": 2.2,
        "width": 1.0,
        "height": 1.0,
        "unit": "m"
    },
    "transportation_requirements": {
        "special_handling": true,
        "fragile": true,
        "insurance_required": true
    }
}
```

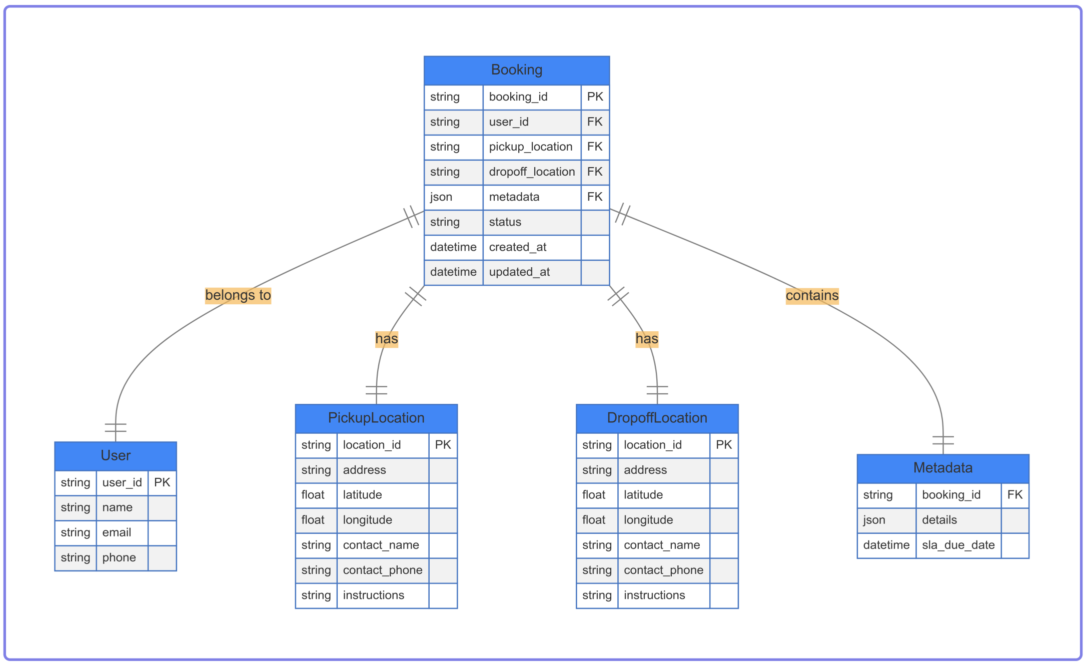
4.5 Relationships Between Data Models

The relationships between the data models in the Booking Management Service (BMS) are depicted in the diagram below. These models define the essential entities and their associations in the booking lifecycle process.

- **Booking**: Represents the central entity that encapsulates the details of a transportation request. It references associated models such as the user who created the booking, the pickup and dropoff locations, and additional metadata.
- **User**: Represents the customer who initiated the booking. A booking is linked to a user via the ‘user_id‘ attribute.

- **Pickup Location**: Defines the starting location of the transportation request. This is associated with a booking via the ‘pickup_location‘ attribute.
- **Dropoff Location**: Specifies the destination for the goods being transported. It is linked to a booking via the ‘dropoff_location‘ attribute.
- **Metadata**: Stores supplementary information about the booking, such as timestamps (e.g., creation time, update time), SLA details, and system-generated attributes.

The following figure illustrates these relationships:



Booking		
string	booking_id	PK
string	user_id	FK
string	pickup_location	FK
string	dropoff_location	FK
json	metadata	FK
string	status	
datetime	created_at	
datetime	updated_at	

User		
string	user_id	PK
string	name	
string	email	
string	phone	

PickupLocation		
string	location_id	PK
string	address	
float	latitude	
float	longitude	
string	contact_name	
string	contact_phone	
string	instructions	

DropoffLocation		
string	location_id	PK
string	address	
float	latitude	
float	longitude	
string	contact_name	
string	contact_phone	
string	instructions	

Metadata		
string	booking_id	FK
json	details	
datetime	sla_due_date	

Figure 1: Entity-Relationship Diagram for Booking Management Service

4.5.1 Detailed Explanations of Relationships

- **Booking → User**: The `user_id` field in the `Booking` model is a foreign key that connects a booking to the user who created it. This relationship enables querying all bookings associated with a particular user.

- **Booking → Pickup Location**: The `pickup_location` field in the `Booking` model references the details of the pickup point. This includes attributes such as the address, contact person, and instructions.
- **Booking → Dropoff Location**: Similarly, the `dropoff_location` field in the `Booking` model references the delivery point, enabling precise tracking of the destination.
- **Booking → Metadata**: The `metadata` field links to system-related information, such as timestamps, SLA policies, and additional optional details. This ensures that a booking's lifecycle is well-documented and traceable.

4.5.2 Use Case Example

Scenario: A user books a logistics service for transporting furniture.

- The `User` model identifies the customer who made the booking.
- The `Pickup Location` and `Dropoff Location` models store the starting and destination addresses for the transport.
- The `Metadata` model holds timestamps for when the booking was created, updated, and completed, as well as SLA-related metrics for compliance tracking.

This structure ensures a comprehensive representation of the relationships between models, enabling a robust and scalable data management approach in the Booking Management Service.

5 Bounded Context Definition

The Booking Management Service operates within a clearly defined bounded context that focuses on the following key areas:

- **Booking Creation:** This encompasses the process of initiating a new booking, capturing all necessary user and transport details.
- **Booking State Management:** It involves tracking and updating the various states of a booking, such as Created, Confirmed, Cancelled, Started, and Completed.
- **Service Allocation:** This pertains to the assignment of transportation services or providers to fulfill the booking requirements effectively.
- **Lifecycle Progression:** It includes managing the transition of bookings through their lifecycle stages, ensuring compliance with business rules and service-level agreements.

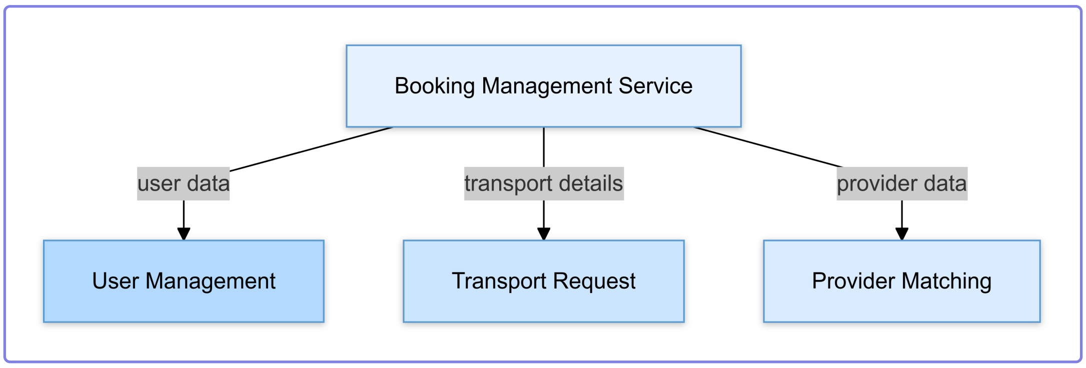


Figure 2: Flow chart for Booking Management Service

5.1 Bounded Context Diagram

5.1.1 Explanation of Bounded Context

The bounded context for the Booking Management Service is designed to encapsulate all functionalities related to booking operations. It clearly delineates responsibilities and interactions with other services, ensuring that the business logic related to booking management is consistently applied and maintained. This context serves as a boundary within which specific business rules, processes, and language are defined and used.

By structuring the Booking Management Service in this manner, we facilitate better maintainability, scalability, and clarity regarding the service's responsibilities in the overall logistics and transport system.

6 Error Handling

Effective error handling is essential for the robustness of the Booking Management Service (BMS). This section outlines standardized error codes, the structure of error responses, and handling scenarios like retries, idempotency, and common error conditions.

6.1 Standardized Error Codes

The service adopts a consistent set of HTTP status codes and domain-specific error codes to represent various error scenarios. The following table summarizes the error codes and their descriptions:

6.2 Error Response Format

All errors are returned in a standardized JSON format to ensure consistency across API responses. The structure includes the following fields:

- **error_code:** A domain-specific error code for programmatic handling.
- **message:** A human-readable description of the error.
- **details:** (Optional) Additional information about the error for debugging purposes.
- **timestamp:** The time at which the error occurred, in ISO 8601 format.

Example Error Response:

Table 3: Standardized Error Codes

HTTP Code	Error Code	Description
400	INVALID_REQUEST	The request payload or parameters are invalid.
401	UNAUTHORIZED	The request is missing authentication credentials or has invalid credentials.
403	FORBIDDEN	The user does not have permission to access the resource.
404	BOOKING_NOT_FOUND	The specified booking ID does not exist in the system.
409	CONFLICT	The booking request conflicts with an existing booking or resource allocation.
422	VALIDATION_ERROR	One or more fields failed validation (e.g., weight exceeds limit).
500	INTERNAL_SERVER_ERROR	An unexpected server error occurred.
503	SERVICE_UNAVAILABLE	The service is temporarily unavailable (e.g., due to maintenance).

```
{
    "error_code": "BOOKING_NOT_FOUND",
    "message": "The specified booking ID does not exist.",
    "details": {
        "booking_id": "12345"
    },
    "timestamp": "2024-11-29T14:35:00Z"
}
```

6.3 Retries and Idempotency

To ensure reliability in distributed systems, the following guidelines are implemented:

6.3.1 Retries

Retries are supported for transient errors, such as network failures or service unavailability (HTTP 503). The client must implement an exponential backoff strategy to avoid overwhelming the system.
Example Scenario: When receiving a `503 SERVICE_UNAVAILABLE` error, the client should retry after 1 second, then 2 seconds, then 4 seconds, etc., up to a maximum of 5 retries.

6.3.2 Idempotency

Certain operations (e.g., booking creation) must be idempotent to avoid duplicate bookings in case of retries. This is achieved using an `Idempotency-Key` header in API requests.
Example:

- A client sends a `POST /bookings` request with an `Idempotency-Key: 12345`.

- If the request times out, the client can resend the same request with the same `Idempotency-Key`.
- The server processes the request only once and returns the same response for subsequent requests with the same key.

6.4 Common Error Scenarios

6.4.1 Validation Failures

Validation errors occur when input data violates predefined constraints. Examples include exceeding weight limits or invalid date formats.
Example Response:

```
{
    "error_code": "VALIDATION_ERROR",
    "message": "The weight exceeds the maximum allowed limit of
       5000 kg.",
    "details": {
        "field": "weight",
        "current_value": 5200,
        "max_allowed": 5000,
        "unit": "kg"
    },
    "timestamp": "2024-11-29T14:35:00Z"
}
```

6.4.2 Conflict Errors

Conflict errors occur when a request violates business rules or overlaps with existing data (e.g., overlapping booking requests).
Example Response:

```
{
    "error_code": "CONFLICT",
    "message": "The booking request overlaps with an existing
       booking.",
    "details": {
        "conflicting_booking_id": "67890",
        "requested_timeslot": "2024-12-01T10:00:00Z to 2024-12-01
           T12:00:00Z"
    },
    "timestamp": "2024-11-29T14:35:00Z"
}
```

6.4.3 Payment Failures

Payment failures can occur due to insufficient funds, expired cards, or payment gateway issues.
Example Response:

```
{
    "error_code": "PAYMENT_FAILURE",
    "message": "The payment could not be processed due to
       insufficient funds.",
    "details": {
        "payment_method": "credit_card",
        "transaction_id": "abc123"
    },
    "timestamp": "2024-11-29T14:35:00Z"
}
```

6.5 Design Principles for Error Handling

- Use clear and descriptive error messages to help clients debug issues effectively.
- Ensure all error responses are consistent in format.
- Avoid exposing internal system details (e.g., stack traces) in production.
- Document all error codes and scenarios in the API specification to aid developers.

7 Event-Driven Interactions

The Booking Management Service (BMS) operates within an event-driven architecture, publishing and consuming events via Kafka topics to facilitate asynchronous communication between microservices. This section details the events published and consumed by BMS, their schemas, and examples. subsectionSummary of Events

Table 4: Summary of Published and Consumed Events

Event Name	**Type**	**Description**
Booking.V1.Created	Published	Emitted when a new booking is created successfully.
Booking.V1.Cancelled	Published	Emitted when a booking is cancelled.
Booking.V1.PaymentConfirmed	Published	Emitted when a payment is confirmed for a booking.
Payment.V1.Successful	Consumed	Indicates that a payment for a booking was successfully processed.
Matching.V1.ProviderAssigned	Consumed	Indicates that a provider has been assigned to a booking.

7.1 Published Events

The BMS publishes the following events during the booking lifecycle:

7.1.1 Booking.V1.Created

Description: This event is emitted when a new booking is successfully created.
Schema:

```
{
    "event_id": "uuid",
    "event_type": "Booking.V1.Created",
    "timestamp": "ISO8601 timestamp",
    "data": {
        "booking_id": "string",
        "user_id": "string",
        "status": "CREATED",
        "pickup_details": {
            "location": {
                "address": "string",
                "latitude": "number",
                "longitude": "number"
            },
            "scheduled_pickup_time": "ISO8601 timestamp"
        },
        "delivery_details": {
            "location": {
                "address": "string",
                "latitude": "number",
                "longitude": "number"
            },
            "scheduled_delivery_time": "ISO8601 timestamp"
        }
    }
}
```

Example Payload:

```
{
    "event_id": "123e4567-e89b-12d3-a456-426614174000",
    "event_type": "Booking.V1.Created",
    "timestamp": "2024-11-29T10:00:00Z",
    "data": {
        "booking_id": "B12345",
        "user_id": "U67890",
        "status": "CREATED",
        "pickup_details": {
            "location": {
                "address": "123 Main St, City",
                "latitude": 40.7128,
                "longitude": -74.0060
            },
            "scheduled_pickup_time": "2024-11-30T09:00:00Z"
        },
        "delivery_details": {
            "location": {
                "address": "456 Elm St, City",
                "latitude": 40.7127,
                "longitude": -74.0059
            },
            "scheduled_delivery_time": "2024-11-30T12:00:00Z"
        }
    }
}
```

7.1.2 Booking.V1.Cancelled

Description: This event is emitted when a booking is cancelled.
Schema:

```
{
    "event_id": "uuid",
    "event_type": "Booking.V1.Cancelled",
    "timestamp": "ISO8601 timestamp",
    "data": {
        "booking_id": "string",
        "user_id": "string",
        "reason": "string"
    }
}
```

Example Payload:

```
{
    "event_id": "123e4567-e89b-12d3-a456-426614174001",
    "event_type": "Booking.V1.Cancelled",
    "timestamp": "2024-11-29T11:00:00Z",
    "data": {
        "booking_id": "B12345",
        "user_id": "U67890",
        "reason": "User requested cancellation"
    }
}
```

7.1.3 Booking.V1.PaymentConfirmed

Description: This event is emitted when a payment associated with a booking is successfully processed.
Schema:

```
{
    "event_id": "uuid",
    "event_type": "Booking.V1.PaymentConfirmed",
    "timestamp": "ISO8601 timestamp",
    "data": {
        "booking_id": "string",
        "payment_id": "string",
        "amount": "number",
        "currency": "string"
    }
}
```

Example Payload:

```
{
    "event_id": "123e4567-e89b-12d3-a456-426614174002",
    "event_type": "Booking.V1.PaymentConfirmed",
    "timestamp": "2024-11-29T12:00:00Z",
    "data": {
        "booking_id": "B12345",
        "payment_id": "P98765",
        "amount": 150.00,
        "currency": "USD"
    }
}
```

7.2 Consumed Events

The BMS consumes the following events from other services:

7.2.1 Payment.V1.Successful

Description: Indicates that a payment was successfully processed for a booking.
Schema:

```
{
    "event_id": "uuid",
    "event_type": "Payment.V1.Successful",
    "timestamp": "ISO8601 timestamp",
    "data": {
        "payment_id": "string",
        "booking_id": "string",
        "amount": "number",
        "currency": "string",
        "status": "SUCCESS"
    }
}
```

Service Action: Upon receiving this event, the BMS updates the booking status to `PAID` and triggers the `Booking.V1.PaymentConfirmed` event.

7.2.2 Matching.V1.ProviderAssigned

Description: Indicates that a service provider has been assigned to a booking.
Schema:

```
{
    "event_id": "uuid",
    "event_type": "Matching.V1.ProviderAssigned",
    "timestamp": "ISO8601 timestamp",
    "data": {
        "booking_id": "string",
        "provider_id": "string",
        "provider_name": "string",
        "vehicle_details": {
            "type": "string",
            "license_plate": "string"
        }
    }
}
```

Service Action: Upon receiving this event, the BMS updates the booking metadata with the assigned provider's details and notifies the user.

7.3 Design Principles for Event Handling

- **Idempotency:** Events are processed in an idempotent manner to prevent duplicate state changes if an event is re-delivered.

- **Event Versioning:** All events include a version number (e.g., `V1`) to ensure backward compatibility.
- **Error Handling:** Events that fail to process are moved to a dead-letter queue (DLQ) for further investigation.
- **Monitoring:** Kafka metrics and logging are used to monitor event throughput and latency.

8 Security

Ensuring the security of the Booking Management Service (BMS) is critical for maintaining user trust and protecting sensitive data. This section outlines the authentication, authorization, data privacy, and encryption mechanisms.

8.1 Authentication and Authorization

- **Authentication:** The BMS uses OAuth 2.0 for user and service authentication. Each request must include a valid JSON Web Token (JWT) in the `Authorization` header.
- **Authorization:** Access to APIs is governed by fine-grained scopes embedded in the JWT. The following scopes are required:
  - `booking:read` - For reading booking details.
  - `booking:write` - For creating or updating bookings.
  - `booking:execute` - For actions like starting or completing a booking.
  - `monitoring:read` - For health check and metrics access.
- **Role-Based Access Control (RBAC):** Access to specific resources is restricted based on roles (e.g., `Admin`, `User`, `Service Provider`).

8.2 Data Privacy and Encryption

- **Data in Transit:** All data transmitted between services and clients is encrypted using TLS 1.2 or higher.
- **Data at Rest:** Sensitive fields, such as user information, pickup and delivery locations, and payment details, are encrypted using AES-256.
- **Field-Level Encryption:** The following fields are encrypted in the database:
  - `user.contact_details`
  - `pickup_details.location`
  - `delivery_details.location`
- **Access Logs:** All access to sensitive data is logged, and logs are retained for 90 days for auditing purposes.

8.3 Token Expiration and Rotation

- JWTs have a short expiration time (15 minutes) to reduce the impact of token compromise.
- Refresh tokens are securely stored and used to obtain new access tokens.

8.4 Security Audits and Compliance

- Regular penetration testing is conducted to identify and mitigate vulnerabilities.
- The service adheres to GDPR and CCPA compliance for data privacy.
- Role and permission reviews are conducted quarterly.

9 Non-Functional Requirements

This section specifies the non-functional requirements, focusing on performance, reliability, and fault tolerance.

9.1 Performance Metrics

- **Response Time SLA:**
  - API responses for critical endpoints (e.g., `POST /bookings`) must be under **200ms** for 95% of requests.
  - Background job processing (e.g., event consumption) must complete within **500ms**.
- **Throughput:** The service must handle **1000 concurrent requests per second** under normal load.

9.2 Retry Policies and Fault Tolerance

- **Retry Policies:**
  - For transient errors (e.g., network timeouts), retries are attempted up to **3 times** with an exponential backoff (initial delay of 1s, doubling on each retry).
  - Idempotency keys are used to prevent duplicate actions during retries.
- **Circuit Breaker:** A circuit breaker pattern is implemented to avoid cascading failures. If a downstream service exceeds a failure threshold, further calls are blocked temporarily.

9.3 High Availability Strategies

- **Redundancy:** The BMS is deployed across multiple availability zones (AZs) to ensure resilience against zone-level failures.
- **Database Replication:** The database is configured with a primary-replica setup, enabling failover in case of primary node failure.

- **Load Balancing:** A load balancer distributes incoming requests across instances, ensuring even workload distribution.
- **Health Checks:** Regular health checks are performed on instances, and unhealthy instances are removed from the load balancer.

9.4 Scalability

- The service is designed to scale horizontally by adding more instances during peak traffic.
- Kafka partitions for event topics are scaled dynamically to handle increased message throughput.

9.5 Monitoring and Alerts

- **Metrics:** Key metrics like request latency, error rates, and event processing lag are monitored via Prometheus.
- **Alerting:** Alerts are configured for SLA breaches, high error rates, or unprocessed Kafka messages in a topic.

10 Versioning and Compatibility

Versioning and compatibility are critical to ensuring the long-term maintainability and extensibility of the Booking Management Service (BMS). This section describes the strategies for REST API versioning and Kafka schema evolution.

10.1 REST API Versioning

The BMS uses explicit versioning in API endpoints to manage changes without disrupting clients.

- **Version Format:** API versions are specified in the URL path using a numeric prefix, e.g., `/v1/bookings`.
- **Versioning Guidelines:**
  - **Major Version Increments:** Introduced for breaking changes such as modified resource structures, renamed endpoints, or changes to authentication mechanisms. Example: `/v2/bookings`.
  - **Minor Version Increments:** Used for backward-compatible enhancements, such as adding new fields to responses or introducing optional query parameters. Example: `/v1.1/bookings`.
  - **Deprecation Policy:** Deprecated endpoints are supported for **6 months** before removal. Clients are notified via response headers (`Deprecation: true`) and the API documentation.
- **Content Negotiation:** Clients can specify the desired API version in the `Accept` header, e.g., `application/vnd.bms.v1+json`.

10.1.1 Example: API Versioning

- **Version 1:**

```
GET /v1/bookings/123
Response:
{
    "booking_id": "123",
    "status": "confirmed",
    "user_id": "456",
    "pickup_location": { ... },
    "delivery_location": { ... }
}
```

- **Version 2:** Introduces new fields and nested structures:

```
GET /v2/bookings/123
Response:
{
    "booking_id": "123",
    "status": "confirmed",
    "user": {
        "id": "456",
        "name": "John Doe"
    },
    "pickup_location": { ... },
    "delivery_location": { ... }
}
```

10.2 Kafka Schema Evolution

To ensure seamless communication in event-driven interactions, the BMS employs schema evolution strategies for Kafka messages.

- **Schema Definition:** All Kafka events follow schemas defined in Apache Avro, which provides built-in support for versioning and compatibility checks.
- **Compatibility Strategy:** The following compatibility rules are enforced:
  - **Backward Compatibility:** New consumers can process older event versions without errors.
  - **Forward Compatibility:** Older consumers can ignore unknown fields in newer event versions.
  - **Full Compatibility:** Ensures that both backward and forward compatibility are satisfied for all schema changes.
- **Schema Registry:** The Confluent Schema Registry is used to manage Avro schemas for Kafka topics. Each schema is versioned, and compatibility checks are enforced during updates.

10.2.1 Example: Kafka Schema Evolution

- **Version 1:**

```
{
    "booking_id": "123",
    "status": "confirmed",
    "user_id": "456"
}
```

- **Version 2:** Adds optional fields while maintaining backward compatibility.

```
{
    "booking_id": "123",
    "status": "confirmed",
    "user_id": "456",
    "priority": "high",          // New field
    "metadata": { ... }          // Nested object
}
```

10.3 Testing Versioning and Compatibility

- **Integration Tests:** Each API and Kafka schema version is tested with both older and newer versions of clients and consumers.
- **Consumer Fail-Safe:** Consumers are implemented to ignore unrecognized fields gracefully, ensuring forward compatibility.
- **Contract Testing:** Tools such as Pact are used to verify contracts between API producers and consumers.

10.4 Deprecation and Migration Strategy

- Deprecated API versions and Kafka schemas are announced via release notes and client communication channels.
- Clients are provided with clear migration guides for upgrading to newer versions.

11 Integration Patterns

The Booking Management Service (BMS) integrates with other services to handle complex workflows and notify external systems. This section describes how the BMS interacts with the Saga Orchestration Service and supports webhooks for real-time notifications.

11.1 Interaction with Saga Orchestration Service

The BMS employs the Saga pattern to coordinate multi-step workflows during the booking lifecycle. This pattern ensures data consistency and reliable coordination of distributed transactions.

11.1.1 Saga Workflow Overview

The following diagram illustrates the interaction between the BMS and the Saga Orchestration Service:

`booking/saga_2.png`

Figure 3: Saga workflow between BMS and SOS

11.1.2 Saga Steps Description

- **Step 1: Create Booking** - The Booking Management Service (BMS) initiates the booking process by publishing a `Booking.Created` event to the Saga Orchestration Service. This marks the start of the workflow.
- **Step 2: Initiate Payment** - The Saga Orchestration Service listens to the `Booking.Created` event and triggers the Payment Service to process the payment. This may involve publishing a `Payment.Initiated` event or calling a Payment Service API.

- **Step 3: Payment Confirmation** - Upon successful payment, the Payment Service publishes a `Payment.Confirmed` event to the Saga Orchestration Service. The orchestrator validates this event and determines the next step.
- **Step 4: Notify User** - The Saga Orchestration Service coordinates with the Notification Service by publishing a `Notification.Send` event. This instructs the Notification Service to inform the user of the booking status.
- **Step 5: Send Booking Confirmation** - The Notification Service sends a confirmation event (e.g., `Notification.Confirmed`) to the BMS. Alternatively, it might call a BMS endpoint to mark the booking as confirmed.

11.2 Webhook Support

The BMS provides webhook functionality to notify external systems about changes in booking states. This real-time notification mechanism allows seamless integration with third-party systems.

11.2.1 Webhook Events

Webhooks are triggered on specific booking events. The following table outlines the supported events:

Table 5: Supported Webhook Events

Event	**Description**
`Booking.Created`	Triggered when a new booking is created.
`Booking.Updated`	Triggered when an existing booking is updated.
`Booking.Cancelled`	Triggered when a booking is cancelled.
`Booking.Completed`	Triggered when a booking is marked as completed.

11.2.2 Webhook Payload Example

Webhooks send JSON payloads to the registered endpoint. Below is an example payload for the `Booking.Created` event:
Example Payload:

11.2.3 Webhook Security Measures

To maintain the integrity and confidentiality of webhook notifications, the following security measures are applied:

- **Authentication:** Each payload includes a secret signature header (e.g., `X-Hub-Signature`) to validate the authenticity of the sender.
- **SSL Encryption:** Webhook communications are secured with HTTPS to protect data in transit.
- **Retry Mechanism:** If delivery fails, the BMS retries up to three times with exponential backoff to ensure reliability.

```
{
    "event": "Booking.Created",
    "booking_id": "123",
    "user_id": "456",
    "pickup_location": {
        "address": "123 Main St",
        "latitude": 37.7749,
        "longitude": -122.4194
    },
    "dropoff_location": {
        "address": "456 Elm St",
        "latitude": 37.7849,
        "longitude": -122.4094
    },
    "scheduled_time": "2024-11-27T10:00:00Z",
    "created_at": "2024-11-26T15:00:00Z"
}
```

11.2.4 Webhook Configuration

Clients can register webhooks via the `/api/v1/webhooks` endpoint by providing the event type(s) they want to subscribe to and the callback URL.
Registration Payload:

```
{
    "callback_url": "https://example.com/webhooks",
    "events": ["Booking.Created", "Booking.Completed"]
}
```

Response Payload:

```
{
    "subscription_id": "789",
    "callback_url": "https://example.com/webhooks",
    "events": ["Booking.Created", "Booking.Completed"],
    "created_at": "2024-11-26T15:30:00Z"
}
```

The integration patterns implemented by the Booking Management Service facilitate efficient coordination of booking workflows and provide robust, secure webhook support for real-time notifications. This ensures seamless integration with both internal and external systems.

12 Testing and Documentation

Effective testing and comprehensive documentation are crucial for ensuring the reliability and usability of the Booking Management Service (BMS). This section outlines the processes for generating OpenAPI specifications and Protocol Buffers (‘.proto‘ files) for the APIs.

12.1 OpenAPI Specifications

OpenAPI Specifications (OAS) provide a standardized way to describe RESTful APIs. The BMS utilizes OpenAPI to generate interactive documentation and facilitate testing. The following steps outline how to generate OpenAPI specifications for the BMS APIs:

1. **Annotation in Code:** Use annotations in the codebase to describe endpoints, parameters, request bodies, and response formats. For example, using Swagger annotations in Java:

   ```
   @Operation(summary = "Create a new booking", tags = {"
      Bookings"})
   @PostMapping("/v1/bookings")
   public ResponseEntity<BookingResponse> createBooking(
      @RequestBody BookingRequest bookingRequest) {
       // Method implementation
   }
   ```

2. **Generate OpenAPI Document:** Use a tool like Swagger or Springdoc to automatically generate the OpenAPI specification file from the annotated code.

3. **Serve Documentation:** Host the generated OpenAPI specification using tools like Swagger UI or Redoc, allowing users to interact with the API documentation easily.

12.2 Protocol Buffers (.proto) Files

For gRPC or services using Protocol Buffers, '.proto' files define the service's APIs and message formats. The following steps outline how to create '.proto' files for the BMS:

1. **Define Service and Messages:** Create a '.proto' file specifying the service methods and message types. For example:

   ```
   syntax = "proto3";

   package booking;

   service BookingService {
       rpc CreateBooking (BookingRequest) returns (
          BookingResponse);
       rpc GetBooking (BookingIdRequest) returns (
          BookingResponse);
   }

   message BookingRequest {
       string user_id = 1;
       string pickup_location = 2;
       string dropoff_location = 3;
   }

   message BookingResponse {
       string booking_id = 1;
   ```

```
        string status = 2;
    }
```

2. **Compile the Proto Files:** Use the Protocol Buffers compiler ('protoc') to generate client and server code in the desired programming languages.
3. **Integration and Testing:** Integrate the generated code into the BMS, allowing for strong typing and efficient serialization of requests and responses.

12.3 Automated Testing

To ensure the reliability of the BMS APIs, automated testing is implemented using the following strategies:

- **Unit Testing:** Individual components of the BMS are tested in isolation to verify their functionality.
- **Integration Testing:** The interactions between different components of the BMS and external services are tested to ensure they work together as expected.
- **Contract Testing:** Ensure that the service adheres to the agreed API contracts using tools like Pact or Postman.

12.4 Documentation Maintenance

Regular updates to the API documentation are essential for accuracy and usability. It is recommended to establish a process for maintaining documentation as follows:

- **Version Control:** Use version control for the documentation to track changes over time.
- **Review Process:** Implement a review process to ensure documentation is up-to-date and reflects the current state of the APIs.
- **User Feedback:** Encourage users to provide feedback on documentation for continuous improvement.