Saga Pattern for Distributed Transactions

1. Overview

This RFC proposes the implementation of the **Saga Pattern** to handle distributed transactions across multiple microservices in the Event Booking Platform. The Saga Pattern ensures data consistency and reliability across services like Booking, Payment, Notification, and Event, even when individual service operations fail.

---

2. Problem Statement

In a microservices architecture, each service operates independently with its own database. When a transaction spans multiple services, maintaining consistency is challenging, especially in cases of partial failures (e.g., payment succeeds, but booking fails). Traditional two-phase commits (2PC) are unsuitable due to scalability and performance constraints.

The **Saga Pattern** solves this by dividing a transaction into multiple steps, each with its own compensating action in case of failures.

---

3. Proposed Solution

Implement the **Saga Pattern** using **Choreography** for event-driven communication. The Booking Service will initiate a distributed transaction and trigger events that downstream services (e.g., Payment and Notification) react to. If a step fails, compensating actions will be triggered to rollback partial operations.

---

4. High-Level Design

4.1 Saga Workflow Example: Booking Transaction

1. **Booking Creation**:
   - Booking Service receives a request to book tickets.
   - It reserves tickets and publishes a `BookingCreated` event to Kafka.
2. **Payment Processing**:
   - Payment Service consumes the `BookingCreated` event.
   - It processes the payment and publishes either a `PaymentSuccess` or `PaymentFailed` event.
3. **Notification**:
   - Notification Service consumes the `PaymentSuccess` event and sends a confirmation email.
   - In case of `PaymentFailed`, Notification Service sends a failure email.
4. **Failure Compensation**:
   - If the Payment Service fails (e.g., insufficient funds), Booking Service consumes the `PaymentFailed` event and releases reserved tickets by invoking compensating actions.

4.2 Choreography Implementation

- **Event-Driven Communication**:
  - Each service publishes events after completing its local transaction.
  - Services listen to relevant events and act accordingly.
- **Kafka Topics**:
  - Use dedicated Kafka topics to decouple services:
    - `booking-events` for `BookingCreated`, `BookingCanceled`.
    - `payment-events` for `PaymentSuccess`, `PaymentFailed`.
    - `notification-events` for `NotificationSent`.

4.3 Service Responsibilities

Service	Actions	Compensating Actions
Booking Service	Reserve tickets, publish `BookingCreated`.	Release reserved tickets on failure.
Payment Service	Process payment, publish `PaymentSuccess` or `PaymentFailed`.	None (failure is already published).
Notification Service	Send confirmation/failure notifications.	None (notifications are best-effort).

---

5. Logging and Observability

5.1 Distributed Tracing

- Use **OpenTelemetry** to track transaction flows across services.
- Include `trace_id` and `span_id` in all logs and events to trace the lifecycle of a saga.

5.2 Centralized Logging

- Use the **ELK Stack** to aggregate logs from all services.
- Log key events (e.g., `BookingCreated`, `PaymentFailed`) for debugging and audit purposes.

5.3 Metrics

- Track key saga metrics using **Prometheus**:
  - Total successful transactions.
  - Total failed transactions and compensations triggered.
  - Average saga execution time.

---

6. Compensating Transactions

Each service implements compensating actions to undo partial operations when an error occurs:

Service	Compensating Action
Booking Service	Release reserved tickets.
Payment Service	None (payment rollback is handled externally).
Event Service	None (event data is not transactional).
Notification Service	None (notifications are idempotent).

---

7. Error Handling and Dead Letter Queue

7.1 Retrying Failed Events

- Use exponential backoff for retrying failed events.
- Example: If the Payment Service cannot process `BookingCreated`, retry up to 5 times with increasing delays.

7.2 Dead Letter Queue (DLQ)

- Route unprocessable events to a DLQ for manual inspection and remediation.
- Use Kafka's native DLQ support for storing failed messages.

---

8. Cache and Idempotency

8.1 Idempotency

- Ensure all saga participants (services) can process the same event multiple times without side effects.
- Example: Payment Service must ensure a `PaymentSuccess` event is only processed once.

8.2 Cache Usage

- Use Redis to store intermediate saga states for quick lookups:
  - Key format: `saga:{transaction_id}`.
  - Store metadata like current state, retries, and failures.

---

9. Database Design

Service	Key Tables	Notes
Booking Service	`Bookings`, `ReservedTickets`	Tracks booking and ticket statuses.
Payment Service	`Payments`, `Refunds`	Tracks payment status and refunds.
Notification Service	`Notifications`, `EmailLogs`	Tracks sent notifications.