

# Messaging and Asynchronous Processing Strategy

---

**Target Audience:** Backend Developers, DevOps

**Purpose:** Defining the event-driven architecture that powers decoupling and resilience in the TMS platform.

---

## 1. Why Asynchronous?

We treat HTTP (Request/Response) as a "Blocker". It forces the caller to wait. We use Asynchronous Messaging (RabbitMQ) to:

1. **Decouple Availability:** If the `Notification Service` is down, the `Ticketing Service` can still sell tickets.
  2. **Flatten Traffic Spikes:** During a "Rush Hour" surge, we can queue thousands of "Send SMS" jobs and process them at a steady rate, protecting downstream providers.
  3. **Ensure Eventual Consistency:** Ensuring that all side-effects (Receipts, Analytics, Inventory Sync) happen reliably, eventually.
- 

## 2. Message Types

We strictly distinguish between **Events** (Facts) and **Commands** (Instructions).

### 2.1 Domain Events (Facts)

- **Definition:** Something that *has already happened*. Past tense. Immutable.
- **Behavior:** Broadcast (Fan-out). Multiple services might listen.
- **Examples:**
  - `TicketSold` : "Seat 4A was sold to User 123."
  - `PaymentFailed` : "Transaction X was rejected."
  - `TripCancelled` : "Trip 999 is cancelled."
- **Consumer Action:** "I need to update my local state based on this new reality."

### 2.2 Commands (Intent)

- **Definition:** A request for a specific service to do something.
  - **Behavior:** Point-to-Point. Specific target.
  - **Examples:**
    - `SendSMS` : "Send text 'Hello' to '078...!'"
    - `GenerateInvoicePdf` : "Create a PDF for Order Z."
  - **Consumer Action:** Execute logic. May succeed or fail.
- 

## 3. Producer & Consumer Responsibilities

### 3.1 Producer (The Emitter)

- **Responsibility:** Validate the payload before publishing.
- **Contract:** Must adhere to the `JSON Schema` defined for that Topic.
- **Reliability:** Must use **Publisher Confirms**. If the Broker doesn't ACK the receipt, the Producer must retry or throw an error to the user (preventing "Silent Failure").

### 3.2 Consumer (The Worker)

- **Responsibility:** Process the message successfully OR fail cleanly.
  - **Idempotency: Must be Idempotent.** The Broker guarantees "At Least Once" delivery. The consumer might receive the same "TicketSold" event twice. It must handle duplicates gracefully (e.g., check `if exists return` ).
  - **Acknowledge (ACK):** Only ACK processed messages.
  - **Negative Acknowledge (NACK):** If a temporary error occurs (Db locked), NACK with `queue=True` .
- 

## 4. Delivery Guarantees

We configure RabbitMQ for **At-Least-Once Delivery**.

1. **Durable Queues:** Queues are saved to disk.
2. **Persistent Messages:** Messages are flagged to survive broker restarts.
3. **Manual ACKs:** Messages are not removed until the Consumer acts.

*Trade-off:* We accept the possibility of duplicate messages to specific efficiency, in exchange for never losing data.

---

# 5. Retry and Dead-Letter Strategy (DLQ)

We do not block queues with "Poison Messages" (messages that crash the consumer).

## 5.1 Retry Policy (Exponential Backoff)

1. **Attempt 1:** Consumer fails (e.g., "SMS Provider 500 Error").
2. **Action:** NACK with Requeue? No. We republish to a `retry_queue` with a **TTL (Time To Live)** (e.g., 5 seconds).
3. **Wait:** Message sits in `retry_queue` until TTL expires.
4. **Dead Letter:** TTL expires -> RabbitMQ moves it back to `main_queue`.
5. **Attempt 2:** Consumer tries again.

## 5.2 Dead Letter Queue (The Graveyard)

- If a message fails **5 times** (configurable max\_retries):
  - **Action:** Publish to `dlq.notification_service`.
  - **Alert:** Monitoring system flags "DLQ Size > 0".
  - **Human Intervention:** Developer inspects the payload. (e.g., "Oh, the phone number was NULL"). Fix bug -> Shovel back to main queue.
- 

# 6. Ordering Guarantees

- **General Rule:** We **DO NOT** guarantee global strict ordering. In a distributed system, Packet B might arrive before Packet A.
- **Mitigation:**
  - Timestamps: Messages include `occurred_at` UTC timestamps.
  - Logic: Consumers act based on timestamps.
    - **Scenario:** A `TripUpdated` event (Time: 10:01) arrives after `TripCancelled` (Time: 10:05).
    - **Logic:** Consumer checks `occurred_at`. "10:01 is older than my current state of 10:05. Ignore."

# 7. Operational Topology

- **Exchange:** `amq.topic` (Topic Exchange).
- **Routing Keys:** `domain.entity.action` (e.g., `ticketing.ticket.sold`).

- **Queues:** Named by **Consumer Service**.

- `q.notification.sms_sender` (Listens to `*.ticket.sold`,  
`*.trip.cancelled`)
  - `q.superadmin.analytics` (Listens to `#` - everything)
-