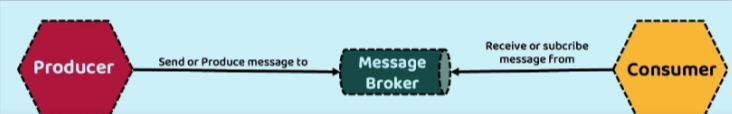
# Rabbit MQ

Rabbitmq is an open-source message broker, which is widely recognized and utilized by most companies. Rabbitmq follows the AMAQ protocol. Advanced Message Queuing Protocol and Rabbitmq offers flexible asynchronous messaging communication between two applications. RabbitMQ which follows the pub/sub model, inside the pub/sub model, we cannot replay the events or messages whenever we use a message broker like Rabbitmq. But in the recent versions of Rabbitmq, event streaming capabilities are also provided with Rabbitmq we can also replay the events or messages. But still, most of the people use Apache Kafka.

**Producer**: The entity responsible for sending messages (also known as the publisher).

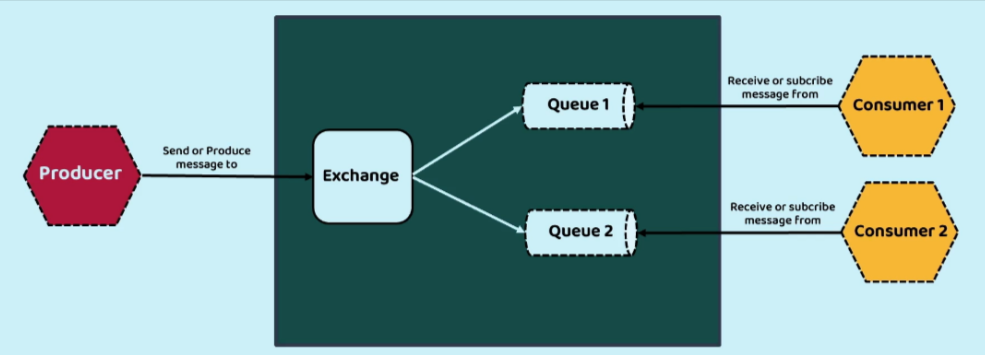
**Consumer**: The entity tasked with receiving messages (also known as the subscriber).

**Message broker**: The middleware that receives messages from producers and directs them to the appropriate consumers.



There can be multiple producers and consumers. The messaging model of AMQP operates on the principles of exchanges and queues. To differentiate multiple producers and consumer exchanges and queues are used. Inside a message broker there can be no exchanges and queues Producers transmit messages to an exchange. Based on a specified routing rule, RabbitMQ determines the queues that should receive a copy of the message. Consumers, in turn, read messages from a queue.

## Exchange Queue



**How They Work Together**

1. A producer sends a message → to an exchange.
2. The exchange applies routing rules(bindings) → delivers message to one or more queues.
3. A consumer reads the message from the queue.

**1. Exchange**

* An **exchange** is like a **post office or router**.
* Producers never send messages directly to a queue. Instead, they send messages to an **exchange**.
* The exchange looks at message attributes and routing rules, then decides **which queue(s) to send the message to**.
* Types of exchanges:
  + **Direct Exchange**  
    Routes messages based on **exact matching** between the routing key and the queue’s binding key.  
    Example: Routing key "error" goes to a queue bound with "error".
  + **Fanout Exchange**  
    Broadcasts messages to **all queues** bound to it (ignores routing key).  
    Example: Send a message → all listening queues receive it.
  + **Topic Exchange**  
    Routes based on **pattern matching** (wildcards).  
    Example: Routing key "order.created" can match bindings like "order.\*" or "order.#".
  + **Headers Exchange**  
    Routes based on **headers/attributes** of the message instead of the routing key.

**2. Queue**

* A **queue** is simply a buffer that stores messages until a consumer takes them.
* It works like a **mailbox**
  + Producers drop letters (messages) into the mailbox.
  + Consumers open the mailbox and read messages, usually in **FIFO** (first in, first out) order.
* Each message is delivered to **only one consumer** of a queue (unless you set up multiple consumers, then RabbitMQ load balances among them).
* Properties of a queue:
  + Named (e.g., "task\_queue")
  + Can be **durable** (survive broker restarts) or transient
  + Can hold unlimited messages (limited by resources)
  + Messages stay until consumed or expired

**3. Bindings**

the rule that tells RabbitMQ “send messages from this exchange to that queue if the routing key matches”. So a binding is like a subscription rule between an exchange and a queue.

Suppose we have:

| **Queue** | **Binding (routing key)** | **What it matches** |
| --- | --- | --- |
| payment-service-queue | payment.\* | any key starting with payment. like payment.created, payment.completed |
| order-service-queue | order.\* | any key starting with order. like order.created, order.cancelled |

**Routing in action**

1. You send message with routingKey = "payment.created".
2. RabbitMQ checks bindings:
   * Does "payment.created" match "payment.\*"? ✅ Yes → message goes into **payment-service-queue**.
   * Does "payment.created" match "order.\*"? ❌ No → skip.

So only **payment-service-queue** gets it.

Now, if you send with routingKey = "order.created":

* "order.created" matches "order.\*" ✅ → goes to order-service-queue.
* "order.created" does not match "payment.\*" ❌ → skip.

So only **order-service-queue** gets it.

**🔹 Multiple queues can match**

Bindings aren’t exclusive — many queues can be subscribed to the same type of event.

For example:

* Queue A binding: "order.\*"
* Queue B binding: "order.created"

Message: "order.created" →  
✅ Matches Queue A (because order.\*)  
✅ Matches Queue B (because exact match order.created)

➡️ So RabbitMQ **duplicates the message** and puts a copy in both queues.  
Each queue then delivers independently to its own consumer.

## Cross domain event

Say we have a use case where order service will create payments i.e. once order service is persisted in DB, it will trigger the payment

So we can do like

1. ❌Order API 🡪 Order exchange 🡪 order-queue🡪 Payment API listener 🡪 read message and trigger payment flow
2. ✅Order API 🡪 Order exchange 🡪 payment-queue🡪 Payment API listener 🡪 read message and trigger payment flow

**1. Exchange vs Queue**

* In RabbitMQ, best practice is:
  + **Services publish to Exchanges**, not directly to Queues.
  + **Consumers bind their own Queues** to the Exchanges for the events they care about.

**2. Principle: Event Ownership**

* **Order Service** owns the **Order domain**.
* Therefore, it is responsible for **publishing order.created events**.
* But it should not decide what consumers (like Payment Service) do with it.

👉 This means Order Service publishes order.created to an exchange in RabbitMQ, not directly to a Payment queue.

**3. Proper Flow in Your Case**

1. **Order Service flow:**
   * Save order in DB.
   * Publish **order.created** event to **order.exchange** (type = topic or fanout).
2. **RabbitMQ flow:**
   * order.exchange has routing keys like order.created, order.cancelled, order.failed.
3. **Payment Service flow:**
   * Declares **its own queue** (e.g., payment.order.created.queue).
   * Binds that queue to order.exchange with routing key = order.created.
   * Consumes the event and triggers payment logic.
   * Then **Payment Service** publishes its own domain events (payment.success, payment.failed) to **payment.exchange**.

## Message based design concept

So, how does RabbitMQ works? In database, I would create an object and that object will persist and any updates will change the state of that object. In RabbitMQ, once I trigger an event and once consumed, it is lost. There is no concept of changing of state of existing object, so that object must be tracked externally. It is just state that we are changing via RabbitMQ. The actual object, it must be persisted somewhere, right? So, say, if I'm updating a person's record, so the person should be persisted in a database, and any state changes will be updated in RabbitMQ.

🔹 Database World (Stateful)

* A database persists objects/entities (rows, documents, etc.).
* Example: INSERT INTO person (id, name, status) VALUES (1, 'Amit', 'ACTIVE');
* If you later update it: UPDATE person SET status = 'INACTIVE' WHERE id = 1;
* The object lives in DB, and changes are stateful mutations of the same entity.
* You can always query back the current state.

🔹 RabbitMQ World (Event-driven, Stateless)

* A message (event) in RabbitMQ is transient:
  + Publisher sends → Exchange → Queue → Consumer reads.
  + Once acknowledged, the message is gone (unless you persist it to another system).
* RabbitMQ does not maintain object state.
* Each message is a fact/event that something happened at a point in time.

**🔹 Kafka’s model (log-based broker)**

* Kafka stores every event in a **commit log** on disk.
* Messages are **retained for a time/size policy** (e.g., 7 days, or 100GB).
* Consumers maintain **offsets** → can replay from the beginning, or from a specific offset.
* This makes Kafka a natural candidate for **Event Sourcing** or as a **system of record**.

**🔹 RabbitMQ’s model (queue-based broker)**

* RabbitMQ is designed for message delivery, not long-term storage.
* Messages live in **queues** until:
  + A consumer **acks** → message removed from queue.
  + TTL (time-to-live) expires.
  + Queue is deleted / broker restarts (if not durable).
* By default: once a message is delivered and acknowledged, it’s **gone forever**.
* Replay is **not possible natively** — unless you manually re-publish from somewhere else.

**🔹** Can RabbitMQ be a “source of truth”?

Not really, at least not in the **same way Kafka can**. Here’s why:

1. **No replayability**
   * Once consumed and acked, the event is gone.
   * You can’t rebuild your DB by replaying past events unless you’ve archived them somewhere.
2. **Persistence is different**
   * RabbitMQ can persist messages to disk if queues and messages are marked as **durable** and **persistent**.
   * But that only protects against broker crashes, not against “oops I want to replay last week’s messages.”
3. **DLQ is not replay**
   * Dead Letter Queues catch failed messages, but that’s for error handling, not for replay/history

**🔹 RabbitMQ vs Kafka — "Source of Truth"**

**✅ RabbitMQ**

* RabbitMQ is a **message broker**:
  + Publisher → Exchange → Queue → Consumer → Message Gone (once ack’d).
* It’s **not a durable event log**.
* Consumers don’t replay past messages unless you implement **manual persistence or DLQs**.
* So in RabbitMQ, the **source of truth is always an external database**.
  + Example: The Order table in Order Service is the truth, RabbitMQ just transports order.created events.

👉 **RabbitMQ = communication channel, not storage.**

**✅ Kafka**

* Kafka is an **append-only log**.
* Every event persisted **durably** on disk and can be **replayed any time**.
* Topics are **partitioned logs** that retain data for days, weeks, or forever.
* Multiple consumers can replay history independently at their own offsets.
* In Event Sourcing systems, Kafka itself can become the **system of record (source of truth)**, because:
  + The log contains **every event ever published**.
  + State can be rebuilt by replaying the log.
  + Example:
    - Event 1: order.created
    - Event 2: payment.completed
    - Event 3: delivery.completed
    - → replay these = reconstruct full order lifecycle.

👉 **Kafka = durable log, can be the truth.**

**🔹 Event Sourcing (why Kafka often is used as source of truth)**

* In **event sourcing**, instead of storing the current state (status=CLOSED), you store **all events** that led to it.
* Current state is a **projection** computed from the event log.
* Example in Kafka: order.created, payment.completed, delivery.completed, order.closed

→ You can always replay to rebuild the order state.

* Since Kafka persists logs durably, it can be treated as the **source of truth**.

**🔹 Why RabbitMQ usually isn’t the source of truth**

* Messages are deleted once consumed.
* It’s not designed for replay or long-term storage.
* You could bolt on persistence (e.g., store messages in a DB before consuming), but then **the DB is the truth, not RabbitMQ**.

**🔹 Enterprise Practice**

* **With RabbitMQ** → DB is the source of truth, RabbitMQ is the delivery mechanism.
* **With Kafka** → Kafka can be the source of truth (event log), and DBs become *projections* (materialized views).

**🔹 Analogy**

* **RabbitMQ** = Postman delivering letters. Once the letter is read and thrown away, it’s gone. You need your filing cabinet (DB) to store what matters.
* **Kafka** = Library archive. Every letter ever sent is permanently stored. You can always go back and read history.

# Patterns

## 1. Saga Pattern

👉 **Problem it solves**:  
When a business transaction spans **multiple microservices**, we need a way to ensure **data consistency** without using distributed transactions (2PC), which don’t scale well.

👉 **How it works**:

* A Saga is a **sequence of local transactions**, where each service updates its own database and then publishes an event.
* If one step fails, the Saga invokes compensating transactions to undo the changes.

👉 **Types of Saga**:

1. **Choreography-based Saga**
   * No central coordinator.
   * Each service listens for events and reacts.
   * Example: Order Service publishes OrderCreated → Payment Service listens → if payment succeeds, publishes PaymentCompleted → Shipping Service listens, etc.
2. **Orchestration-based Saga**
   * A central **orchestrator** tells each service what to do.
   * Example: Order Orchestrator calls Payment Service → if success, calls Shipping Service → if failure, calls Payment Cancel Service.

👉 **Use case**:  
Order placement across services (Order, Payment, Inventory, Shipping).

## 2. Outbox Pattern

👉 **Problem it solves**:  
When using **event-driven architecture**, you need to **reliably publish events** after updating a database. The problem:

* If you update the DB but fail to publish the event → inconsistency.
* If you publish the event but fail to commit DB → inconsistency.

👉 **How it works**:

* Instead of publishing directly, the service writes the event into an **Outbox table** in the same database as part of the local transaction.
* A background process (or Debezium/CDC) reads the Outbox table and publishes the event to the message broker (Kafka, RabbitMQ, etc.).
* This ensures **atomicity** between DB update and event publishing.

👉 **Example**:

1. Order Service saves Order in DB + Outbox entry order.created.
2. Outbox Processor picks it up and publishes to RabbitMQ/Kafka.
3. Other services consume the event.

## 3. Domain-Driven Design (DDD)

👉 **Problem it solves**:  
Large systems become messy if we don’t align **business concepts** with the code.

👉 **Core idea**:

* Software should reflect the **domain (business language)**.
* Developers and domain experts share a **Ubiquitous Language**.
* Code is structured around **business concepts**, not technical details.

👉 **Key concepts**:

* **Entities** → objects with identity (e.g., Customer, Order).
* **Value Objects** → immutable objects without identity (e.g., Money, Address).
* **Aggregates** → cluster of entities treated as a unit (e.g., Order + OrderItems).
* **Repositories** → provide access to aggregates.
* **Bounded Contexts** → each microservice/domain has its own model (Order BC, Payment BC).
* **Domain Events** → express something that happened in the business (OrderPlaced, PaymentFailed).

👉 **Use case**:  
An e-commerce system:

* **Order Context** deals with orders.
* **Payment Context** deals with payments.
* **Shipping Context** deals with deliveries.  
  Each bounded context is modeled separately and communicates via events.

**🔗 How they connect in enterprise systems**

* **DDD** structures the system into clear bounded contexts.
* **Outbox Pattern** ensures reliable event publishing when aggregates change.
* **Saga Pattern** coordinates business transactions across bounded contexts.

✅ **Simple Flow Example (Order Placement)**

1. **Order Service (DDD)**: Creates Order aggregate → saves to DB → adds event to Outbox.
2. **Outbox Pattern**: Event OrderCreated is published reliably.
3. **Saga Pattern**: Payment Service listens → if success, triggers Shipping → if failure, compensates by canceling Order.

## Complexity

**1. Saga Pattern – Most Challenging**

* **Why hard**:
  + Requires **orchestration/choreography design** → deciding who drives the saga.
  + Handling **compensating transactions** is tricky (undoing actions isn’t always simple, e.g., refunding money, restocking inventory).
  + Error handling, retries, and **idempotency** must be carefully managed.
  + Observability is difficult (tracking the saga across services).
* **Enterprise challenge**: Needs careful **business domain knowledge** + robust event-driven infra.
* 🔥 **Most challenging of the three**, especially in large systems.

**2. Domain-Driven Design (DDD) – Conceptually Hard**

* **Why hard**:
  + Requires **deep collaboration** with domain experts.
  + Developers must learn to think in **business language**, not just technical CRUD.
  + Enforcing **bounded contexts** and avoiding “big ball of mud” is tough.
  + Needs discipline in modeling **aggregates, events, and contexts**.
* **Enterprise challenge**: Harder in companies where business and dev teams are siloed.
* ❗ Conceptually challenging but **extremely valuable** once embraced.

**3. Outbox Pattern – Least Challenging**

* **Why easier**:
  + Mostly **technical implementation** problem.
  + Either build your own outbox processor or use **CDC tools like Debezium**.
  + Complexity is in **reliability & performance tuning** (batching, retries, cleanup).
* **Enterprise challenge**: Less about business, more about infra.
* ✅ Technically straightforward once you pick a strategy.

## Dual write problem

The dual-write problem is a common issue in microservices where a service must update its own database and publish an event to a message broker in a single logical operation. A failure between these two writes can result in an inconsistent state, where the database update succeeds but the event is never sent, or vice versa.

This is a distributed transaction problem that traditional, single-database ACID (Atomicity, Consistency, Isolation, Durability) transactions cannot solve because the write operations occur on two separate systems. Ignoring this problem can lead to inconsistent data that is difficult to debug and can cause downstream system failures.

Common solutions to the dual-write problem

1. Transactional Outbox Pattern

The transactional outbox pattern ensures that the database update and the event creation occur atomically by placing both operations within a single, local database transaction.

* How it works:
  1. The microservice creates a new record in its business table and an entry in an outbox table within the same ACID transaction.
  2. A separate, asynchronous process (the "outbox relay") polls the outbox table for new entries.
  3. When the relay finds an event, it publishes it to the message broker (e.g., Kafka) and marks the event as processed.
* Pros:
  1. Guarantees atomicity for both writes, preventing inconsistencies caused by partial failures.
  2. The outbox relay can implement a robust retry mechanism for publishing events, ensuring at-least-once delivery.
* Cons:
  1. Introduces extra infrastructure and components (the outbox table and relay) to manage.
  2. There is a slight delay between the database commit and the event being delivered, leading to eventual consistency.

2. Change Data Capture (CDC)

With CDC, the microservice only writes to its own database. A separate tool monitors the database's transaction log (the "binlog") and streams changes as events to a message broker.

* How it works:
  1. The microservice updates its database.
  2. A CDC tool like Debezium or a database-specific feature (e.g., DynamoDB Streams) captures the change from the database's internal transaction log.
  3. The CDC tool publishes the captured change as an event to the message broker.
* Pros:
  1. Eliminates the need for a separate outbox table and the application code to manage it.
  2. Provides high reliability and low latency for capturing database changes.
* Cons:
  1. Adds extra infrastructure (the CDC tool) to operate and maintain.
  2. Can introduce operational complexity related to managing the database log and the CDC process.

3. Event Sourcing

In an event-sourced system, the current state of an application is not stored directly but is derived from a sequence of events.

* How it works:
  1. Instead of updating a record in place, the microservice appends a new event (e.g., FundsDeposited) to an immutable event store.
  2. This event can then be read by a separate process (like a CDC tool) or the event store itself, which then publishes it to a message broker.
* Pros:
  1. Maintains a complete and immutable history of all state changes, which is useful for auditing and debugging.
  2. The single write to the event store is atomic by nature.
* Cons:
  1. Can be complex to implement and query, as the current state must be reconstructed from the event log.
  2. The read model for querying is typically implemented using the Command Query Responsibility Segregation (CQRS) pattern, which adds complexity.

4. Saga Pattern

The saga pattern is a sequence of local, compensating transactions for managing long-running distributed transactions that span multiple services.

* How it works:
  1. A saga is a chain of local transactions. Each transaction is handled by a single service and publishes an event to trigger the next step.
  2. If any local transaction fails, the saga executes compensating transactions to undo the changes made by preceding steps, returning the system to a consistent state.
* Pros:
  1. Maintains data consistency without using a traditional distributed transaction protocol (like two-phase commit), which is often impractical in microservices.
  2. Supports eventual consistency and is highly scalable.
* Cons:
  1. Adds significant complexity to the system design.
  2. Requires careful design of compensating actions for every possible failure scenario.

5. Two-Phase Commit (2PC)

The 2PC protocol is a traditional approach to distributed transactions that ensures all participating services either commit or abort a transaction together.

* How it works:
  1. A coordinator service sends a "prepare" request to all participants.
  2. If all participants are ready, the coordinator sends a "commit" request.
  3. If any participant fails to prepare, the coordinator sends an "abort" request.
* Pros:
  1. Provides strong consistency across distributed systems.
* Cons:
  1. The blocking nature of 2PC holds locks on resources for long periods, which can cause poor performance and reduce throughput in high-load systems.
  2. The coordinator is a single point of failure and can leave the system in an inconsistent state if it crashes at a critical moment.
  3. Many modern messaging systems and NoSQL databases do not support 2PC.

How to choose the right solution

|  |  |  |  |
| --- | --- | --- | --- |
| Pattern | Consistency Model | Use Case | Considerations |
| Transactional Outbox | Eventual Consistency | Reliable event publishing, especially when combining database writes with message queues. | Requires adding an outbox table and a relay service. Well-suited for many event-driven microservices. |
| Change Data Capture (CDC) | Eventual Consistency | Automatically emitting events for all database changes. Good for decoupling the event-publishing logic from the application. | Requires operating an additional CDC tool. Often lower-impact on service performance. |
| Event Sourcing | Eventual Consistency | Systems where a full, immutable history of state changes is valuable, such as in finance or auditing. | Adds significant design complexity. Best when specific business requirements justify the effort. |
| Saga | Eventual Consistency | Complex, multi-step business transactions spanning multiple microservices. | Increases complexity, especially with compensation logic. Requires careful management of the entire workflow. |
| Two-Phase Commit (2PC) | Strong Consistency | Legacy or specific niche cases requiring strict consistency across multiple databases, often not recommended for modern microservices. | Low performance and scalability. Highly sensitive to network and service failures. |

# Solution

## ACID

ACID transactions refer to database operations that possess four essential properties—Atomicity, Consistency, Isolation, and Durability—to ensure reliability and data integrity, especially during system failures. Atomicity ensures transactions are all-or-nothing, consistency maintains data validity according to predefined rules, isolation prevents concurrent transactions from interfering with each other, and durability guarantees that committed changes are permanent.

The four ACID properties

* **Atomicity:** This principle ensures that a transaction is treated as a single, indivisible unit**. Either all operations within the transaction are successfully completed, or none of them** are; if any part fails, the entire transaction is rolled back, and the database returns to its state before the transaction began.
* **Consistency:** A transaction must start in a valid state and maintain the database's integrity. It ensures that a transaction adheres to all defined rules, constraints, and relationships within the database.
* **Isolation:** This property ensures that the concurrent execution of multiple transactions does not interfere with each other. Each transaction appears to run as if it were the only one executing, preventing anomalies that could occur if they interacted.
* **Durability:** Once a transaction has been successfully committed, its changes are permanent and will survive system failures, such as power outages. This guarantees that the committed data is safely stored.

Why are ACID transactions important?

ACID transactions are crucial for maintaining the reliability and correctness of database systems. They provide a high level of assurance that data will be stored accurately, remain consistent, and be available even when unexpected errors or system failures occur.

While most relational databases adhere to ACID properties, some modern NoSQL databases follow the BASE model, which prioritizes availability and scalability over immediate consistency.

* ACID (Strong Consistency): A database is always in a consistent state. It is crucial for applications that require high data integrity, such as financial systems.
* BASE (Eventual Consistency): The database's state can be temporarily inconsistent, but all data will eventually become consistent. This model is often used for high-volume, distributed systems where immediate consistency is less critical.

In our use case there are multiple entities involved like Order, Payment Delivery. These entities will depend on lot of external factors payment may depend on eternal payment systems; delivery may depend on some delivery partners. And they will have delays also, for example payment will depend on external vendor will take time and can eventually fail. Same for delivery, it will take a lot of time and there will be updates incrementally over a period. So, what if these systems fail? What do we do with the transactions that we have already logged in the system? This will either make the system very cumbersome and can also lead to incorrect status in the system.

## Saga Pattern in this design

The Saga pattern solves **long-running distributed transactions** across multiple services (Order → Payment → Delivery → Order close).  
Instead of a single ACID transaction, Saga coordinates multiple **local transactions** through **events**.

👉 How it’s applied here:

* Orchestration style Saga (via events & RabbitMQ):
  1. Order Service creates order, writes order.created into outbox → published to RabbitMQ.
  2. Payment Service consumes order.created, saves Payment (INITIATED → COMPLETED) in its own DB + outbox, publishes payment.completed.
  3. Delivery Service consumes payment.completed, persists delivery, updates status, publishes delivery.completed.
  4. Order Service consumes delivery.completed → marks order CLOSED → publishes order.closed.
* **Compensation / Failure flows** (Saga rollback):
  1. If **Payment fails** → Payment Service updates status = FAILED, outbox event = payment.failed.
  2. payment.failed event is consumed by **Order Service**, which updates Order to CANCELLED (compensating action).
  3. Similarly, **Delivery failure** → Delivery publishes delivery.failed, retries with TTL, and finally escalates to DLQ + notifies Order Service, which marks Order as FAILED.

✅ Benefit: Instead of one global transaction, each service commits its **local transaction + outbox** safely, and the **Saga flow of events** guarantees that either:

* the whole process completes successfully (**happy path**), or
* compensating actions roll back the business process (e.g. cancel order if payment fails).

## Outbox Pattern in this design

When you use SAGA you must save to DB and publish event. The Outbox pattern solves this **dual-write problem.** If you write to DB and publish to RabbitMQ in two separate steps, you can lose messages (e.g. DB commit succeeds but app crashes before Rabbit publish).

👉 Outbox pattern solves this is two steps

1. **When placing an order**:
   * We write to **orders table** (the business state).
   * At the same time, in the **same DB transaction**, we write an **outbox row** into outbox table (event = order.created, payload = orderId, customerId, etc).
   * This ensures **atomic persistence** — either both are saved, or none.
2. **Publisher (OutboxPublisher)**:
   * Polls outbox table for PENDING rows.
   * Publishes them to RabbitMQ (order.exchange, routing key = order.created).
   * Marks them as SENT in DB so they are not processed again as per step 1.
   * Retries on failure (with attempts counter).
   * This guarantees **eventual delivery** even if RabbitMQ was down when the order was created.

✅ Benefit: You never lose events. If service crashes after DB commit but before Rabbit publish, the **outbox row is still there**, and the publisher will pick it up later.

## Issues with Outbox

**🔹 Why Outbox feels “slower”**

with the polling version of Outbox:

* You **insert** into DB + outbox in one transaction. ✅ (atomic)
* Then you have a **poller** (every 1s, 500ms, etc.) that looks for PENDING rows and publishes.
* The drawbacks of this approach are
  1. Publishing is **asynchronous** (not immediate).
  2. Even if there are no new events, polling still queries DB.
  3. You might feel a small latency overhead (DB → RabbitMQ delay = poll interval).

But! The reason teams still use it: **reliability & simplicity**. Even if RabbitMQ is down, the outbox keeps retrying. Without it, you risk losing events.

**🔹 Can we do better than polling?**

Yes, absolutely. There are **two main evolutions**:

**1. Transaction Log Streaming (Debezium style)**

* Instead of polling, you hook into DB change streams (CDC).
* A connector like **Debezium** captures every insert into outbox table directly from the DB log, and publishes to RabbitMQ/Kafka.
* This removes polling completely.  
  ✅ Near real-time  
  ✅ No wasted queries  
  ❌ More infra (Kafka Connect, Debezium, connectors).

**2. Insert + Notify pattern**

* Some databases (Postgres, MySQL 8, etc.) let you use **NOTIFY/LISTEN** or triggers.
* When a new outbox row is inserted, a DB trigger signals your app.
* Your app immediately publishes it.  
  ✅ Instant event publishing  
  ❌ Extra DB coupling, not portable (H2 won’t support this properly).

**🔹 Where does Dead Letter Queue (DLQ) fit in?**

DLQ doesn’t replace Outbox.  
It’s a **RabbitMQ feature** for handling **failed deliveries** (e.g., consumer can’t process message).

* **Outbox solves producer-side reliability** → “Did the event get published at all?”
* **DLQ solves consumer-side reliability** → “What if consumer can’t process the event?”

So if you drop Outbox and just rely on RabbitMQ + DLQ:

* You still have the **dual-write problem** (DB commit succeeds, publish fails → message lost).
* DLQ won’t help here, because DLQ only catches messages **that made it to RabbitMQ**.

That’s why **Outbox + DLQ work together**, not instead of each other:

1. **Outbox** ensures the event is *published at least once* into RabbitMQ.
2. **DLQ** ensures consumers have a safe place for failed messages.

**🔹 Optimizing the Outbox in your use case**

Since you’re using **Spring WebFlux + R2DBC + RabbitMQ**, you can:

1. **Keep polling simple at first** (like 500ms interval).
   * For learning and initial saga orchestration, it’s enough.
   * Latency = ~0.5–1s, which is fine for business flows (order → payment → delivery).
2. **Later upgrade** to a push model:
   * Use Postgres with LISTEN/NOTIFY (or Debezium CDC).
   * That way you don’t waste cycles polling when no messages exist.
3. **Combine with RabbitMQ DLQ**:
   * For Payment/Delivery services: if processing fails, message goes to DLQ (retry or investigation).
   * This protects **consumers** while Outbox protects **producers**.

## E**vent-Sourced Update pattern**

👉 *Instead of writing to DB + Outbox in one transaction, can I just publish order.created directly to RabbitMQ, then consume my own event and update my orders table?*

This is sometimes called an **“event-sourced update pattern”** or a **“self-consuming event”**. Let’s evaluate it.

**🔹 How it works (your idea)**

1. API call → **publish order.created to RabbitMQ** (no DB write yet).
2. Order Service itself has a **consumer** for order.created.
3. The consumer receives the event and writes to the **orders table**.
4. Other services (Payment, Delivery) also consume the same order.created.

**🔹 Pros**

1. **No Outbox Polling Needed**
   * Immediate publish to RabbitMQ, no DB polling loop.
   * Faster than Outbox with polling.
2. **Event-first architecture**
   * RabbitMQ is the “source of truth,” DB is just a projection.
   * This is closer to **Event Sourcing**.
3. **Simple at first glance**
   * Only one write per API call (to RabbitMQ), instead of DB + outbox.

**🔹 Cons**

1. **Event lost risk (Dual-write in reverse)**
   * What if RabbitMQ is down? The order.created message may never reach the broker.
   * Then nothing is persisted — the order disappears forever.
   * In Outbox pattern, at least the DB has a durable record.
2. **At-least-once duplication**
   * RabbitMQ guarantees *at least once delivery*.
   * Your consumer may see duplicates. If you blindly insert into orders table each time, you’ll get multiple rows for same order.
   * You’d need **idempotency** checks in DB (e.g., unique orderId, check before insert).
3. **Ordering issues**
   * If you extend Saga (e.g., order.updated, order.cancelled), your DB is only eventually consistent with event flow.
   * If your consumer lags behind, API might say “order placed” but DB still doesn’t show it.
4. **Tightly coupling DB with RabbitMQ availability**
   * With Outbox, you can still take orders even if RabbitMQ is down (they sit in outbox until MQ is back).
   * With event-first approach, if RabbitMQ is down → you lose orders, unless you buffer them in memory/disk → reinventing Outbox.
5. **Testing and debugging complexity**
   * Harder to reason about state, because DB is never the "source of truth".
   * If you need to reprocess events (e.g., rebuild DB), you must replay from RabbitMQ (or archived events).

**🔹 When this makes sense**

This approach **can work well** if:

* You adopt **true Event Sourcing** → RabbitMQ (or Kafka) is the *system of record* for all events, and the DB is just a read model/projection.
* You’re OK with **eventual consistency** in the DB.
* You implement **idempotency** (deduplication) in your consumer.

**🔹 When Outbox is better**

* If you need **strong guarantees** that every business transaction (like an order) is durably recorded, even when RabbitMQ is unavailable.
* If your DB is the **system of record** (most transactional systems).
* If you want **auditable, replayable logs** but without building a full Event Sourcing infrastructure.

✅ **Summary**:

* **Your idea (publish first, consume self to update DB)** = Event Sourcing style. Faster, but more fragile unless you accept DB as a projection only.
* **Outbox pattern** = Reliable DB-first approach. Slightly slower (polling), but safe and battle-tested.
* **DLQ** complements both, but doesn’t replace Outbox.

## Hybrid Approach

👉 *When API is called*

1. *publish order.created event*
2. *write to the* ***outbox*** *(no direct write to orders table).*

*Then the same service (Order Service) listens to its own event, consumes it, and updates the orders table.*

So the DB is updated **via consuming the event**, not in the original transaction. Let’s analyze.

**🔹 Flow (your hybrid idea)**

1. API → write **outbox entry only** (order.created event, no DB order or main table yet).
2. OutboxPublisher → publishes event to RabbitMQ.
3. Order Service → consumes order.created → inserts into orders table.
4. Other services (Payment, Delivery) also consume order.created.

**🔹 Pros**

1. **Still have Outbox reliability**
   * The order.created event is durable in DB before publish → you don’t lose events if RabbitMQ is down.
2. **Single source of truth = events**
   * DB becomes a projection of the event log, similar to Event Sourcing.
   * Easier to rebuild orders table from events if needed.
3. **Simpler order flow**
   * You don’t need to write to orders and outbox in one transaction → only outbox.
   * Less dual-write in application logic.
4. **Natural replayability**
   * If you ever wipe the orders table, you can replay order.created events from broker or outbox to rebuild it.

**🔹 Cons**

1. **Eventual consistency (within the same service!)**
   * API call returns success when outbox row is written.
   * But the orders table won’t show the order until Rabbit publish → consume → DB insert happens.
   * This means:
     + API caller might immediately query /orders/{id} and not see it.
     + You need to accept small delays.
2. **More moving parts in one service**
   * You’re adding an internal round-trip via RabbitMQ for something that could have been a simple DB insert.
   * If the consumer is delayed or stuck, your own service lags behind on its own state.
3. **Extra RabbitMQ traffic**
   * Even internal self-updates (orders table) go through RabbitMQ.
   * Higher load compared to writing directly.
4. **Complex error handling**
   * What if your order-consumer fails to process order.created but other services succeed?
   * Payment might have started, but your own DB doesn’t show the order → inconsistent local view.

## Comparison of three approaches

| **Approach** | **How it works** | **Pros** | **Cons** | **Good for** |
| --- | --- | --- | --- | --- |
| **DB-first (classic Outbox)** | Save order + outbox in one tx → publish → consumers update their DB | Strong consistency in service DB, reliable events, works offline if MQ is down | Slight latency (polling), two writes per API call | Traditional transactional systems (orders, payments) |
| **Event-first (publish + consume self)** | Publish event → consume event to update DB | Low latency, DB is just a projection, no outbox table needed | If MQ is down, you lose order; need idempotency; DB only eventually consistent | Event Sourcing systems |
| **Hybrid (your idea)** | Write outbox only → publish → consume self to update DB | Reliability of outbox + event-driven updates, DB can be rebuilt from events | Eventual consistency even inside the service, more MQ traffic, complex error scenarios | Systems leaning toward Event Sourcing but still want Outbox durability |

**🔹 My take**

* If **your DB is the system of record** (most e-commerce/order systems) → use **classic Outbox**.
* If you want to go **full Event Sourcing** → use **event-first**, skip outbox and make broker the source of truth.
* The **hybrid** is like sitting in between → reliable but makes your own service’s state eventually consistent, which often confuses API clients (“I placed an order but don’t see it yet”).

👉 In practice, most companies **don’t choose hybrid** unless they’re consciously moving toward Event Sourcing. They stick with:

* **Outbox** if DB is source of truth.
* **Event Sourcing** if broker is source of truth.

**1. Classic Outbox (DB-first)**

* Flow:  
  API → save order + outbox (same tx) → poller → publish → consumers update their DB.
* Latency:
  + API response: **fast** (just DB write).
  + Event publish: depends on **poll interval** (e.g., 500ms–1s).
  + State in orders table: **immediate** (since order is inserted directly).
* ✅ Fast **for API + local DB state**.
* ❌ Event publication slightly **delayed** due to polling.

**2. Event-first (Publish + consume self to update DB)**

* Flow:  
  API → publish event to RabbitMQ → consumer (self + others) → DB write.
* Latency:
  + API response: **fast** (just Rabbit publish).
  + Event publish: **immediate** (Rabbit gets it right away).
  + State in orders table: **slower** (depends on consumer lag; not guaranteed immediate).
* ✅ Fastest **for inter-service events** (Payment, Delivery see it instantly).
* ❌ Local DB state is **eventually consistent** → slower for your own queries.

**3. Hybrid (Outbox only, then consume self to update DB)**

* Flow:  
  API → save outbox row → poller → publish event → self-consume → DB write.
* Latency:
  + API response: **fast** (just DB write to outbox).
  + Event publish: delayed by poll interval (like Outbox).
  + Local DB update: delayed twice (poll + consume).
* ✅ Reliable, replayable.
* ❌ **Slowest for local DB state** — you don’t see your own order until after Rabbit publish + consume.

**🔹 Ranking by Speed**

* **Fastest events to other services:**  
  ✅ **Event-first** (direct publish → Rabbit).
* **Fastest local DB consistency:**  
  ✅ **Classic Outbox** (order + outbox saved in same tx).
* **Slowest overall:**  
  ❌ **Hybrid** (extra hops even for your own DB).

**⚡ Rule of Thumb**

* If you need **your own DB consistent immediately** (e.g., API client queries /orders right after placing an order) → **Classic Outbox** is fastest.
* If you need **events to reach other services ASAP** (and can tolerate eventual consistency in your own DB) → **Event-first** is fastest.
* Hybrid is slower for both worlds — it’s usually chosen only if you want to drift toward **event-sourcing style**, not for speed.

## Do On Success callback

*Can I just use .doOnSuccess / .doOnError of sender.send(...) to decide whether to update my DB (e.g., mark saga initiated or not)?*

**🔹 What .doOnSuccess / .doOnError actually mean here**

* sender.send(Mono<OutboundMessage>) (from **reactor-rabbitmq**) returns a Mono<Void>.
* .doOnSuccess runs **when the send operation completes successfully** (i.e., RabbitMQ accepted the message into the exchange).
* .doOnError runs **if publish fails immediately** (e.g., bad connection, serialization error).

⚠️ But note:

* RabbitMQ doesn’t guarantee **message is routed to a queue or consumed** — only that the broker accepted it.
* If your routingKey doesn’t match any queue, message may vanish silently (unless you enable mandatory flag + return listener).
* If consumer fails later, you won’t get notified here.

So .doOnSuccess only tells you:  
✅ "RabbitMQ broker accepted the publish."  
❌ It does **not** mean "the saga is safely in progress."

**🔹 Enterprise implications if you rely only on this**

* **Pros**:
  + Simple, reactive, fast feedback.
  + Good enough for low-criticality flows (e.g., sending an email or notification).
* **Cons**:
  + If DB save succeeded but publish failed → you must now rollback or mark saga as “not initiated.” This is the **dual-write problem** all over again.
  + If publish succeeded but later consumer fails → your DB thinks saga is initiated, but in reality it didn’t progress.
  + No retry mechanism → you’re trusting a single fire-and-forget publish.

**🔹 Enterprise-grade alternatives**

1. **Transactional Outbox (reliable)**
   * Save event + DB update in the same transaction.
   * Publisher polls outbox and retries until RabbitMQ acknowledges.
   * This way, DB always knows the saga initiation state correctly.  
     ✅ Used in **Payment Service** (critical path).
2. **Idempotent retry + DLQ (semi-reliable)**
   * Publish directly.
   * If .doOnError, retry with exponential backoff.
   * Consumers are idempotent (can handle duplicate order.created).  
     ✅ Good compromise for **Order Service** or **Delivery Service**.
3. **RabbitMQ publisher confirms (extra reliability)**
   * Reactor RabbitMQ supports publisher confirms (waiting for broker ack).
   * Stronger than just .doOnSuccess.
   * Still doesn’t solve the dual-write problem if DB update and publish are not atomic.

**🔹 Recommendation (for your setup)**

* **Order Service (event-first, learning)**:
  + You *can* use .doOnSuccess/.doOnError to log or mark saga initiation.
  + But accept that you might be inconsistent if publish fails after DB commit.
* **Enterprise answer**:
  + Don’t tie DB updates to .doOnSuccess/.doOnError.
  + Use **Outbox** or **CDC** when saga initiation status is business-critical.
  + Use .doOnError only for retry/backoff or logging.

## Oubbox proper implementation

1. Outbox row is written in the **DB transaction** (✅ good, guarantees DB consistency) publish = false
2. A scheduled poller reads rows where published=false.
3. It **publishes to RabbitMQ** via rabbitTemplate.convertAndSend.
4. If publish succeeds → you mark published=true.
5. If publish fails → you do nothing, row stays published=false → retried later.

*What if sending to Rabbit queue fails and data is updated in outbox table?*

👉 If you set event.setPublished(true) **before confirming RabbitMQ send succeeded**, you lose the event forever (it’s marked as sent but never actually delivered).

That would break **exactly-once** and you’d miss messages → ❌ not acceptable in enterprise systems.

*@Scheduled*(fixedDelay = 2000) *// every 2s  
public void* publishUnsentEvents() {  
 outboxRepository.findByPublishedFalse()  
 .flatMap(event -> {  
 *try* {  
 rabbitTemplate.convertAndSend(exchange, event.getEventType(), event.getPayload()); *// This fails* logger.info("✅ Published event: {}", event);  
 event.setPublished(*true*);  
 *return* outboxRepository.save(event).then();  
 } *catch* (Exception e) {  
 *return* Mono.error(e);  
 }  
 })  
 .subscribe();  
}

**1. At-least-once delivery is the norm**

* Outbox pattern usually guarantees **at-least-once** delivery, not exactly-once.
* That means:
  + An event may be published **twice** (if a retry happens).
  + But it should **never be lost**.

So **consumers must be idempotent** (ignore duplicates based on aggregateId or eventId).

**2. Don’t mark published until RabbitMQ confirms**

* Your code immediately sets published=true after convertAndSend.
* But convertAndSend is **fire-and-forget** (it doesn’t confirm broker receipt).

👉 Enterprise upgrade: enable **publisher confirms** in RabbitMQ (publisher-confirm-type=correlated).

***spring.rabbitmq.publisher-confirm-type=correlated***

Then you can update published=true only after confirmation.

**3. Handle failures safely**

* If RabbitMQ is down, the event should remain published=false.
* Your poller will retry later.
* Never set published=true until success is confirmed.

**4. Enterprise Enhancements**

* **Dead Letter Queue (DLQ):** If publish fails after N retries, move event to DLQ for manual investigation.
* **Retry policy:** Instead of retrying endlessly every 2s, use exponential backoff.
* **Batch publishing:** Send multiple events in one go for efficiency.
* **CDC (Debezium):** In very large systems, replace polling with Change Data Capture on the DB binlog.

## DB First or RabbitMQ first

Even if you are not implementing Outbox what should we do

* Publish to broker then persist to DB
* Persist to DB then publish to broker

**🔎 Lets look at option 1**

1. **Publish event to RabbitMQ first.**
   * Wait for broker confirm (ACK/NACK).
   * If ACK → proceed to insert into DB.
   * If NACK → fail the flow.
2. **Database is updated *after* RabbitMQ confirms.**
   * If publish fails, nothing is persisted.
   * If publish succeeds, DB is updated.

**✅ Pros**

* **Fast**: You don’t wait for DB before sending event.
* **Simplicity**: No Outbox, no pollers.
* **Aligned with event-first thinking** (RabbitMQ as “source of truth”).

**⚠️ Cons (Enterprise Risks)**

1. **Inconsistent state if DB fails after ACK**
   * Imagine RabbitMQ confirms delivery → Payment Service consumes order.created.
   * But your DB insert fails (network, schema, deadlock).
   * Downstream services think the Order exists, but Order DB has no record.
   * ❌ This breaks consistency.
2. **RabbitMQ is not a source of truth**
   * RabbitMQ is **not durable storage** like Kafka.
   * Messages can be lost if queues are deleted or retention expires.
   * Without DB-first persistence, you have no guaranteed history.
3. **Transactions are broken**
   * RabbitMQ ACK and DB insert are not in one atomic transaction.
   * You rely on happy-path success in both places, but failures create drift.

**📦 Enterprise Best Practice**

Even if you don’t want Outbox (which is the gold standard), the **enterprise-safe order** is:

1. **Insert Order into DB first** (system of record).
2. **Publish Event** (with confirm).
3. If publish fails →
   * Retry, or
   * Keep Order in DB with status = PUBLISH\_PENDING.
   * Have a retry job to republish.

👉 This guarantees that **Order Service DB never lies**. At worst, the event is delayed, but never missing.

## Best Practices

**✅ Big-picture guarantees (what enterprise teams expect)**

* **Durability:** no important domain change is “lost”.
* **At-least-once delivery:** messages can be retried and duplicated, consumers must be idempotent.
* **Observability & traceability:** you can trace a request / saga across services.
* **Recoverability:** ability to replay missed events and repair state.
* **Security & governance:** secure transport, auth, schema management and audit.
* **Operational readiness:** LB, retries, DLQ, health checks, metrics, runbooks.

**Patterns & options (tradeoffs)**

1. **Outbox pattern (recommended for critical flows)**
   * Write business row **and an outbox row** in the same DB transaction.
   * A background agent (poller or CDC) publishes events and marks outbox rows published.
   * Guarantees DB = source-of-truth, durable events, replayability.
   * Enterprise: Use CDC (Debezium → Kafka → bridge) instead of DB poller for scale.
2. **DB-first, publish-after (no outbox)**
   * Insert business row, then publish. If publish fails, mark business record PUBLISH\_PENDING and retry.
   * Simpler than outbox but requires a retry/repair process. Not as robust.
3. **Publish-first, then DB (not recommended)**
   * Faster but dangerous: if DB insert fails after broker ACK, state is inconsistent. Avoid for critical flows.
4. **Broker-transaction (RabbitMQ tx) / publisher confirms**
   * RabbitMQ transactions exist, but *publisher confirms* are recommended (lighter and async).
   * Always prefer publisher confirms + application-level handling over broker tx for performance.

**Enterprise best practices — full checklist (grouped + prioritized)**

**MUST-HAVES (do these first)**

1. **DB-first semantics** (unless you have full event sourcing): save your business entity before concluding success to clients.
2. **Outbox or DB-first-with-retry** for any critical financial data (payments). Outbox is recommended.
3. **Publisher confirms** on producers: enable and wait for ACK before marking the event published.
   * Spring (properties): spring.rabbitmq.publisher-confirm-type=correlated and spring.rabbitmq.publisher-returns=true
   * Reactor RabbitMQ: sendWithPublishConfirms()
4. **Manual consumer ACK**: ack only after local DB transaction succeeds. Use AcknowledgeMode.MANUAL or manual basicAck.
5. **Idempotency** on consumers: use messageId/aggregateId dedup store or idempotency keys.
6. **DLQ + retry policy**: have retries and then DLQ for poison/irrecoverable messages (set reasonable TTL/retry-count).
7. **Schema migration**: use Flyway/Liquibase for DB schema and outbox table.
8. **Observability**: propagate traceId, correlationId, causationId in message headers and logs (OpenTelemetry/Zipkin).
9. **Monitoring & alerts**: metrics for publish success rate, outbox lag, queue length, consumer errors, DLQ growth.

**STRONGLY RECOMMENDED**

1. **CDC (Debezium) for outbox** at scale → avoids polling load.
2. **Exponential backoff with jitter** for retries; cap attempts and route to DLQ.
3. **Batch publishing** from outbox for throughput (e.g., publish 50 events per poll).
4. **Idempotent writes**: have unique DB constraints or a processed\_message table to avoid duplicate processing.
5. **Rate limiting and circuit breakers** on external calls & broker publishing.
6. **Message schema versioning** (use a schema registry if you need strict validation).
7. **Contract testing** (Pact), integration tests with Testcontainers (RabbitMQ + MySQL).

**NICE-TO-HAVE / OPTIMIZATIONS**

1. **Outbox cleanup/archive**: delete or archive published rows after some time.
2. **Partitioned outbox / sharding** for very high throughput.
3. **Replay tooling** for re-publishing historic events.
4. **End-to-end trace sampling and dashboards** for latency and saga progress.

**Concrete technical practices & examples**

**RabbitMQ producer side**

* Use **publisher confirms** (reactor-rabbitmq sendWithPublishConfirms) and wait for ACK before marking DB/polling state as published.
* Enable returns so unroutable messages are visible: spring.rabbitmq.publisher-returns=true.

**Reactor RabbitMQ snippet**

sender.sendWithPublishConfirms(Mono.just(outboundMessage))

.next() // get the first confirm for single message

.flatMap(confirm -> {

if (confirm.isAck()) {

// acked -> update DB / mark outbox published

} else {

// NACK -> retry or keep marked as pending

}

return Mono.empty();

});

**Spring AMQP + RabbitTemplate (non-reactive)**

* Configure CachingConnectionFactory confirm type CORRELATED and use CorrelationData to observe confirms.

Properties:

spring.rabbitmq.publisher-confirm-type=correlated

spring.rabbitmq.publisher-returns=true

**RabbitMQ consumer side**

* Use **manual ack** container factory and ack only after DB persisted (and transaction committed).
* Example: SimpleRabbitListenerContainerFactory#setAcknowledgeMode(AcknowledgeMode.MANUAL).
* On non-recoverable errors, basicNack(deliveryTag, false, false) to route message to DLQ.

**DLQ + retry pattern (typical RabbitMQ)**

* Main queue arguments:
  + x-dead-letter-exchange → retry exchange / DLX
  + x-dead-letter-routing-key → routing key to retry queue
* Retry queue has TTL (x-message-ttl) and x-dead-letter-exchange back to main exchange to requeue after TTL.
* After N attempts, move to final DLQ for manual investigation.

Queue argument example:

Map<String,Object> args = new HashMap<>();

args.put("x-dead-letter-exchange","delivery.exchange"); // to route back or to DLQ

args.put("x-dead-letter-routing-key","delivery.retry");

args.put("x-message-ttl", 60000); // 60s TTL in retry queue

**Outbox table (schema + indexes)**

* Columns: id (PK), aggregate\_id, aggregate\_type, event\_type, payload (JSON), created\_at, published (boolean), published\_at, attempts, last\_error.
* Indexes: published (for polling), created\_at, maybe (published, id) for ordering.
* Keep id as surrogate BIGINT AUTO\_INCREMENT for efficient ordering.

SQL example:

CREATE TABLE outbox (

id BIGINT PRIMARY KEY AUTO\_INCREMENT,

aggregate\_id VARCHAR(36),

aggregate\_type VARCHAR(50),

event\_type VARCHAR(100),

payload JSON,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

published BOOLEAN DEFAULT FALSE,

attempts INT DEFAULT 0,

last\_error TEXT,

published\_at TIMESTAMP NULL

);

CREATE INDEX idx\_outbox\_published ON outbox (published, created\_at);

**Idempotency & deduplication**

* Consumer should use either:
  + **processed\_messages table** with message\_id as unique key; insert only if not exists (transactional), or
  + Add unique constraint on business keys so reprocessing cannot create duplicate business rows.
* TTL/cleanup for the dedup store.

**Transactions (reactive)**

* Use TransactionOperator or @Transactional with reactive support; **do not call .block()** in reactive code.
* Example using TransactionOperator:

Mono<Void> tx = transactionOperator.execute(txStatus ->

paymentRepository.save(payment)

.then(outboxRepository.save(event))

.then()

);

**Failure scenarios & how to handle them (practical runbook items)**

1. **Broker ACK but DB insert fails**
   * If you use DB-first + outbox: cannot happen (both persisted in DB).
   * If you publish-first: risk of event delivered but DB missing → unacceptable for critical flows.
2. **DB commit succeeds but publish fails**
   * Outbox row remains; poller retries → eventual publication.
   * Monitor outbox lag and alert.
3. **Message duplicated**
   * Consumer must detect duplicates; use processed\_messages or idempotent update logic.
4. **Poison message (deserialization/validation fails)**
   * Send to DLQ for manual investigation; do not retry forever.
5. **Network partition between app & RabbitMQ**
   * Publisher confirms will fail; keep events in outbox/pending status; exponential backoff & alert.
6. **Broker running out of disk or memory**
   * Configure broker policies & resource alarms; degrade producers with backpressure.

**Observability / Metrics you must have**

* Outbox: unpublished\_count, avg\_time\_to\_publish (created -> published).
* RabbitMQ: queue lengths, unacked messages, publish success rate, return rate.
* Consumers: processing latency, error rate, DLQ count.
* Business: number of PUBLISH\_FAILED orders.
* Traces: distributed trace across services with correlation id.

**Operational & security items**

* **TLS for RabbitMQ** and secure credentials (don’t hardcode). Use vault/secret manager.
* **RBAC**: limit which services can publish/consume particular exchanges/queues.
* **Retention & archival** of outbox processed rows and message payloads (privacy/compliance).
* **Backup/restore** strategy for DB; outbox is part of source-of-truth backup.
* **Schema migrations** via Flyway/Liquibase; include outbox DDL.

**Testing strategy**

* **Unit tests** for idempotency & deserialization.
* **Contract tests** (Pact) between producer and consumer.
* **Integration tests** with Testcontainers (RabbitMQ + MySQL).
* **Chaos testing**: simulate broker down and verify outbox recovery.
* **Load tests** to tune batch size / poll interval / concurrency.

**Simple prioritized adoption plan (practical)**

1. **MUST**: DB-first + publisher confirms + manual consumer ack + DLQ + idempotency.
2. **MUST**: Schema migration + monitoring + trace headers.
3. **NEXT**: Implement Outbox (if critical) + poller or CDC. Add exponential backoff + DLQ thresholds.
4. **NEXT**: Batch publishing, batch cleanup of outbox, replay tooling.
5. **LATER**: CDC → Kafka pipeline if throughput grows; partitioning; schema registry; consumer-driven contracts.

**Short examples you can copy**

**Spring properties**

# Rabbit

spring.rabbitmq.publisher-confirm-type=correlated

spring.rabbitmq.publisher-returns=true

# Consumer concurrency (example)

spring.rabbitmq.listener.simple.concurrency=3

spring.rabbitmq.listener.simple.max-concurrency=10

**Manual ack container factory**

@Bean

public SimpleRabbitListenerContainerFactory manualAckContainerFactory(ConnectionFactory cf) {

var factory = new SimpleRabbitListenerContainerFactory();

factory.setConnectionFactory(cf);

factory.setAcknowledgeMode(AcknowledgeMode.MANUAL);

return factory;

}

**Outbox poller behavior (pseudo)**

* Poll SELECT \* FROM outbox WHERE published=false ORDER BY id LIMIT 100.
* Publish batch → wait confirms → mark published & set published\_at and attempts++.
* On error increment attempts, record last\_error, and if attempts > N move to DLQ table or set flag for operator.

**Final advice (short)**

* For **Payment service** (critical): **Outbox + CDC** (or poller if learning) + publisher confirms + idempotent consumers = enterprise-standard.
* For **Delivery** (less critical): DB-first + publish with confirms + retry/DLQ may be acceptable without outbox, but plan for idempotency.
* Treat **RabbitMQ as transport**, **DB as source of truth**. Use outbox to bridge them reliably.

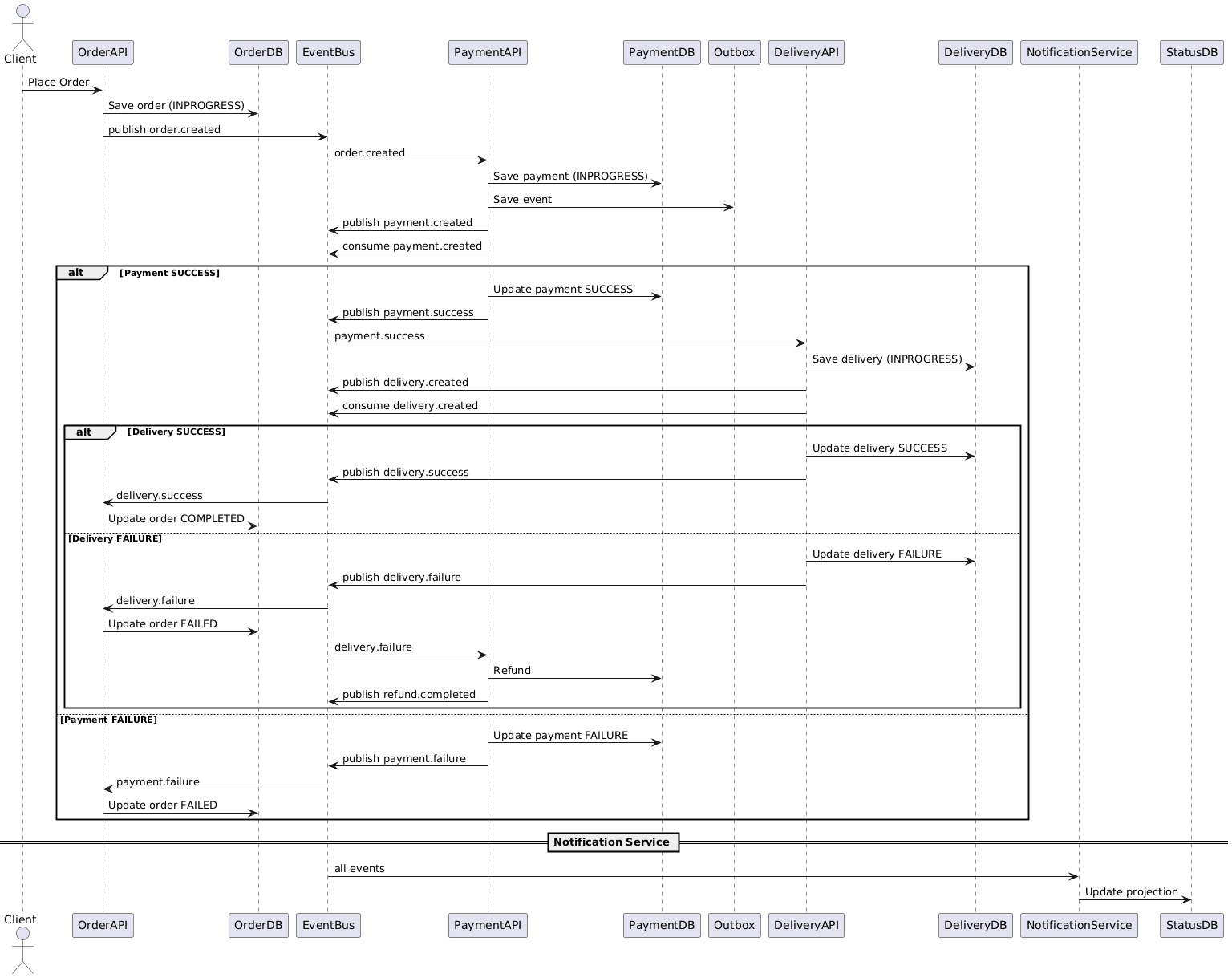
# Implementation

## Sequence diagram



Paste the sequence diagram in the below site to generate diagram

<https://www.plantuml.com/plantuml/uml/SyfFKj2rKt3CoKnELR1Io4ZDoSa700001>



## Application flow

**1️⃣ Order Creation**

1. **Client → OrderAPI**
   * Persist order in DB with orderStatus=INPROGRESS.
   * Publish order.created event → **OrderExchange (event bus)**.
2. **Event Bus**
   * Routes order.created to:
     + payment-service-queue (PaymentAPI)
     + notification-service-queue (NotificationService)

**2️⃣ Payment Flow**

1. **PaymentAPI (on order.created)**
   * Persist Payment record with paymentStatus=INPROGRESS.
   * Persist event to Outbox.
   * Poller publishes payment.created event → **PaymentExchange**.
2. **Event Bus**
   * Routes payment.created to:
     + payment-service-queue (PaymentAPI self-processing)
     + notification-service-queue (NotificationService)
3. **PaymentAPI (on payment.created)**
   * Execute dummy logic (charge card / simulate logic).

**If Success:**

* + Update paymentStatus=SUCCESS.
  + Publish payment.success.

**If Failure:**

* + Update paymentStatus=FAILURE.
  + Publish payment.failure.

1. **Event Bus**
   * Routes:
     + payment.success → delivery-service-queue, order-service-queue, notification-service-queue.
     + payment.failure → order-service-queue, notification-service-queue.

**3️⃣ Delivery Flow (on payment.success)**

1. **DeliveryAPI (on payment.success)**
   * Persist Delivery record with deliveryStatus=INPROGRESS.
   * Publish delivery.created.
2. **Event Bus**
   * Routes delivery.created to:
     + delivery-service-queue (DeliveryAPI self-processing).
     + notification-service-queue (NotificationService).
3. **DeliveryAPI (on delivery.created)**
   * Execute dummy delivery logic.

**If Success:**

* + Update deliveryStatus=SUCCESS.
  + Publish delivery.success.

**If Failure:**

* + Update deliveryStatus=FAILURE.
  + Publish delivery.failure.

1. **Event Bus**

* Routes:
  + delivery.success → order-service-queue, notification-service-queue.
  + delivery.failure → order-service-queue, payment-service-queue (refund), notification-service-queue.

**4️⃣ Order Completion / Compensation**

1. **OrderAPI (on delivery.success)**

* Update orderStatus=COMPLETED.

1. **OrderAPI (on payment.failure or delivery.failure)**

* Update orderStatus=FAILED.

1. **PaymentAPI (on delivery.failure)**

* Initiate refund logic.
* Update DB with paymentStatus=REFUND.
* Optionally publish refund.completed.

**5️⃣ Notification Flow**

1. **NotificationService**

* Subscribes to **all events** (order.\*, payment.\*, delivery.\*, refund.\*).
* Builds a **materialized view** (e.g., order\_status\_view).
* Provides API for client to fetch consolidated status.

order.created [OrderAPI]

└─ payment.created [PaymentAPI]

├─ payment.success [PaymentAPI]

│ └─ delivery.created [DeliveryAPI]

│ ├─ delivery.success [DeliveryAPI]

│ │ └─ order.COMPLETED [OrderAPI]

│ └─ delivery.failure [DeliveryAPI]

│ ├─ order.FAILED [OrderAPI]

│ └─ refund.completed [PaymentAPI]

└─ payment.failure [PaymentAPI]

└─ order.FAILED [OrderAPI]

## Rabbit MQ set up

**🔎Separate topic exchanges**

* You have **separate topic exchanges**:
  + order.exchange
  + payment.exchange
  + delivery.exchange
* Each service publishes to its **own exchange**.
* Bindings connect one service’s exchange to the **next service’s queue**.
  + Example: order.exchange → payment-service-queue for order.created.
  + Example: payment.exchange → delivery-service-queue for payment.completed.

👉 This is a **point-to-point event choreography** — each exchange knows exactly which queue(s) to deliver to.

**Pros**

* Clear ownership: each service owns its exchange.
* Fine-grained control: routing keys are explicit.

**Cons**

* **Tight coupling** → e.g., Payment must know who consumes payment.success.
* Harder to add new subscribers → if tomorrow AnalyticsService also wants payment.success, you must add a new binding.
* More exchanges = more management overhead.

**🔎 Event Bus Model**

* Instead of **per-service exchanges**, you use a **single fanout/topic exchange** → e.g., domain.events.
* All services publish their events (order.created, payment.success, delivery.failure, …) into this **one event bus**.
* Each service has a **queue bound to this bus**, with filters (routing keys).
  + order-service-queue subscribes to: payment.\*, delivery.\*.
  + payment-service-queue subscribes to: order.\*, delivery.failure.
  + delivery-service-queue subscribes to: payment.\*.
  + notification-service-queue subscribes to: # (all events).

👉 This is **pub/sub** → producers don’t need to know about consumers.

**Pros**

* Very **decoupled** — services don’t know about each other.
* Adding new consumers (e.g., Notification, Analytics, Fraud detection) requires **no producer changes**.
* Fits the **“event bus”** pattern (enterprise standard).

**Cons**

* One big exchange → may need routing discipline (order.\*, payment.\*) to avoid event mix-ups.
* Governance needed to prevent “event soup”.