# Rabbit MQ

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 has to 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.

# Our Solution

## Outbox Pattern in this design

The **Outbox pattern** solves the **dual-write problem** (DB + Message Broker consistency).  
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).

👉 How it’s applied here:

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.
   * Example:
   * return orderRepository.save(order)

.then(outboxRepository.save(outbox))

.as(txOperator::transactional);

1. **Publisher (OutboxPublisher)**:
   * Polls outbox table for PENDING rows.
   * Publishes them to RabbitMQ (order.exchange, routing key = order.created).
   * Marks them as SENT in DB.
   * 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.

## 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).

**🔹 Where they fit together**

* **Outbox pattern** ensures **reliable event publishing** (no lost events between DB + Rabbit).
* **Saga pattern** ensures **process consistency across services** (order → payment → delivery → order finalization), using those events as the chain.

In other words:

* **Outbox** makes each service a trustworthy event source.
* **Saga** makes the overall process consistent by sequencing those events.

## Issues with Outbox

In order service and delivery service we are not using outbox pattern , using in payment only

**🔹 Why Outbox feels “slower”**

Yes — 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.
* This means:
  + Publishing is **asynchronous** (not immediate).
  + Even if there are no new events, polling still queries DB.
  + 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**.

## Alternate **event-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**

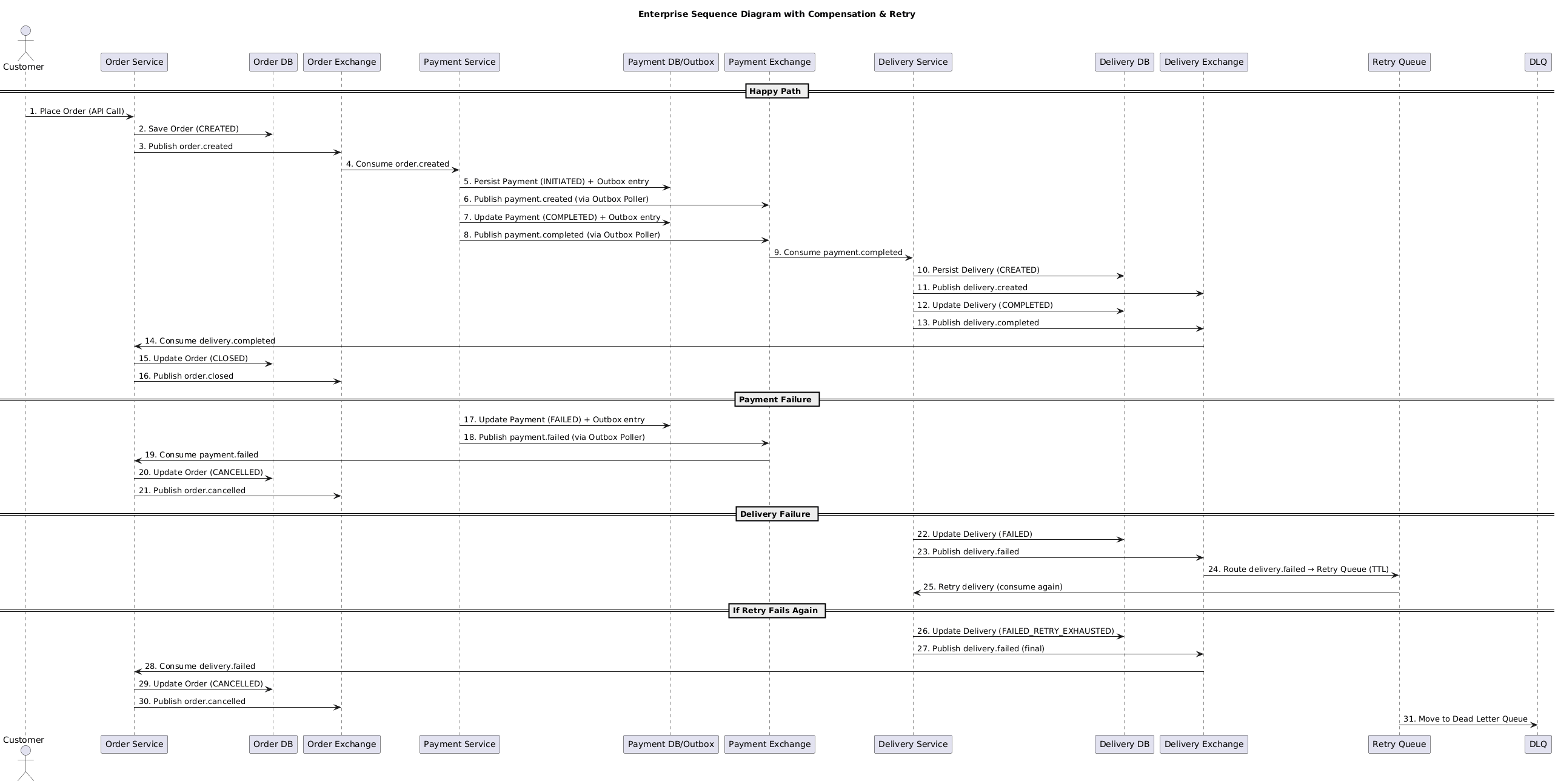
* **Dead Letter Queue (DLQ):** If publish fails after N retries, move event to DLQ for manual investigation.
* **Retry policy:** Instead of retrying endlessly every 2s, use exponential backoff.
* **Batch publishing:** Send multiple events in one go for efficiency.
* **CDC (Debezium):** In very large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

# Implementation



Paste the sequence diagram in the below site to generate diagram

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



| **Step** | **Service** | **Action** | **Database** | **Exchange** | **Queue** | **Routing Key** | **Purpose** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | Client/Actor | Call Order API /orders | — | — | — | — | Place order request |
| 2 | Order Service | Persist Order (status = CREATED) | order | — | — | — | Ensure order saved |
| 3 | Order Service | Publish order.created | — | order.exchange | order-service-queue | order.created | Notify that new order was created |
| 4 | Payment Service | Consume order.created | — | order.exchange | payment-service-queue | order.created | Start payment workflow |
| 5 | Payment Service | Insert Payment record (status = CREATED) + Outbox entry (payment.created) in same transaction | payment, outbox | — | — | — | Atomic write of business + event |
| 6 | Outbox Poller | Reads outbox row (NEW) | outbox | — | — | — | Ready to publish |
| 7 | Outbox Poller | Publish payment.created | — | payment.exchange | payment-service-queue | payment.created | Notify payment entity created |
| 8 | Outbox Poller | Mark outbox row as PROCESSED | outbox | — | — | — | Prevent duplicates |
| 9 | Payment Service | Update Payment (status = IN\_PROGRESS) + Outbox entry (payment.inprogress) | payment, outbox | — | — | — | Move workflow ahead |
| 10 | Outbox Poller | Publish payment.inprogress | — | payment.exchange | payment-service-queue | payment.inprogress | Notify processing started |
| 11 | Payment Service | Update Payment (status = COMPLETED) + Outbox entry (payment.completed) | payment, outbox | — | — | — | Mark payment as successful |
| 12 | Outbox Poller | Publish payment.completed | — | payment.exchange | payment-service-queue | payment.completed | Notify success |
| 13 | Payment Service | Publish delivery.created (trigger Delivery Service) | — | delivery.exchange | delivery-service-queue | delivery.created | Kick off delivery |
| 14 | Delivery Service | Consume delivery.created | — | delivery.exchange | delivery-service-queue | delivery.created | Create delivery entry |
| 15 | Delivery Service | Persist Delivery (status = CREATED) | delivery | — | — | — | Track delivery start |
| 16 | Delivery Service | Publish delivery.inprogress | — | delivery.exchange | delivery-service-queue | delivery.inprogress | Notify delivery started |
| 17 | Delivery Service | Update Delivery (status = COMPLETED) | delivery | — | — | — | Finalize delivery |
| 18 | Delivery Service | Publish delivery.completed | — | delivery.exchange | order-service-queue | delivery.completed | Notify Order Service |
| 19 | Order Service | Consume delivery.completed | — | delivery.exchange | order-service-queue | delivery.completed | Mark order as completed |
| 20 | Order Service | Update Order (status = COMPLETED) | order | — | — | — | End-to-end workflow success ✅ |

| **Step** | **Service** | **Action** | **Database** | **Exchange** | **Queue** | **Routing Key** | **Purpose** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | Payment Service | Consume order.created | — | order.exchange | payment-service-queue | order.created | Start payment process |
| 2 | Payment Service | Insert Payment (status = CREATED) + Outbox entry (payment.created) | payment, outbox | — | — | — | Persist business + event |
| 3 | Outbox Poller | Publish payment.created | — | payment.exchange | payment-service-queue | payment.created | Notify new payment |
| 4 | Payment Service | Attempt to process payment | payment | — | — | — | Actual charge attempt |
| 5 | Payment Service | Failure occurs | payment | — | — | — | Transaction failed |
| 6 | Payment Service | Auto Retry once (e.g., after 30 sec) | — | — | — | Handle transient failure |  |
| 7 | Payment Service | If retry success → update Payment COMPLETED and continue Happy Flow | payment, outbox | — | — | — | Recover automatically |
| 8 | Payment Service | If retry fails again → Update Payment FAILED\_RETRY\_EXHAUSTED + Outbox entry (payment.failed) | payment, outbox | — | — | — | Mark failure final |
| 9 | Outbox Poller | Publish payment.failed | — | payment.exchange | order-service-queue | payment.failed | Notify Order Service |
| 10 | Order Service | Consume payment.failed | — | payment.exchange | order-service-queue | payment.failed | Cancel order |
| 11 | Order Service | Update Order → CANCELLED | order | — | — | — | Mark order cancelled ❌ |
| 12 | External Actor | Call /payments/{id}/retry API | — | — | — | — | Manual retry triggered |
| 13 | Payment Service | Retry payment manually | payment, outbox | — | — | — | Allow human-triggered retry |
| 14 | Outbox Poller | Publish payment.completed (if success) | — | payment.exchange | payment-service-queue | payment.completed | Resume Happy Flow |

| **Step** | **Service** | **Action** | **Database** | **Exchange** | **Queue** | **Routing Key** | **Purpose** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | Payment Service | After payment.completed, publish delivery.created | — | delivery.exchange | delivery-service-queue | delivery.created | Trigger delivery |
| 2 | Delivery Service | Persist Delivery (status = CREATED) | delivery | — | — | — | Begin workflow |
| 3 | Delivery Service | Publish delivery.inprogress | — | delivery.exchange | delivery-service-queue | delivery.inprogress | Notify started |
| 4 | Delivery Service | Failure occurs (e.g., logistics error) | delivery | — | — | — | Delivery failed |
| 5 | Delivery Service | Auto Retry once (e.g., re-dispatch to courier) | — | — | — | Handle transient failure |  |
| 6 | Delivery Service | If retry success → update Delivery COMPLETED and continue Happy Flow | delivery | — | — | — | Recover automatically |
| 7 | Delivery Service | If retry fails again → Update Delivery FAILED\_RETRY\_EXHAUSTED | delivery | — | — | — | Mark failure final |
| 8 | Delivery Service | Publish delivery.failed | — | delivery.exchange | order-service-queue | delivery.failed | Notify Order Service |
| 9 | Order Service | Consume delivery.failed | — | delivery.exchange | order-service-queue | delivery.failed | Cancel order |
| 10 | Order Service | Update Order → CANCELLED | order | — | — | — | Final failure ❌ |
| 11 | External Actor | Call /deliveries/{id}/retry API | — | — | — | — | Manual retry triggered |
| 12 | Delivery Service | Retry delivery manually | delivery | — | — | — | Attempt again |
| 13 | Delivery Service | Publish delivery.completed (if success) | — | delivery.exchange | order-service-queue | delivery.completed | Resume Happy Flow |