RabbitMQ EDA Backend for E-commerce — Step-by-Step Guide (Java)

> **Audience:** Junior+ Java developers
>
> **Goal:** Ship a simple, scalable Event-Driven Architecture (EDA) for an e-commerce backend using **RabbitMQ + Spring Boot 3 + PostgreSQL + Redis**, with clear steps, runnable code snippets, and operational guidance.

Table of Contents

Architecture Overview

Core ideas: - **RabbitMQ** handles event routing (topic exchanges), **queues** per concern, **DLQ** for failures. - **Spring Boot 3 (Java 17/21)** accelerates dev with `spring-boot-starter-amqp`, `data-jpa`, `validation`. - **PostgreSQL** stores transactional data and **Outbox**. - **Redis** caches read models and session-ish data (rate limits, carts). - **Micrometer + Prometheus** for metrics; **OpenTelemetry** for tracing; **JSON logging** for logs.

High-level flow: 1. API receives command (e.g., `POST /orders`). 2. Domain logic writes to Postgres **inside a transaction** and adds an **Outbox** row. 3. Outbox publisher (scheduled or transactional publisher confirms) emits event to RabbitMQ. 4. Downstream services consume, update their stores/materialized views, and possibly emit more events. 5. Clients query **read models** via REST; writes are commands that trigger events.

Domain & Services

Start small; split by clear bounded contexts. Suggested minimal set:

- **Catalog Service**
- Owns `Product` (id, title, price, stockPolicy, etc.)
- Emits events: `ProductCreated`, `ProductPriceChanged`, `InventoryAdjusted` (optional)

- **Inventory Service**

- Owns `StockItem` per product/warehouse.
- Consumes order reservations; emits `InventoryReserved`, `InventoryReservationFailed`, `InventoryReleased`.

- **Orders Service**

- Owns `Order` aggregate with items and state machine (CREATED → PAID → FULFILLED/CANCELLED).
- Emits: `OrderCreated`, `OrderCancelled`, `OrderPaid`, `OrderFulfilled`.

- **Payments Service (mock first)**

- Simulates gateway auth/capture.
- Consumes `OrderCreated`/`PaymentRequested`; emits `PaymentAuthorized`, `PaymentFailed`.

- **Notifications Service**

- Listens to user-visible milestones; sends email/WhatsApp; emits `NotificationSent` (optional).

- **BFF/Checkout API**

- Synchronous API surface for frontend.
- Reads from materialized views (Postgres/Redis) and issues write commands.

  You can start with **Orders + Payments (mock) + Inventory** and add the rest later.

Eventing Model (RabbitMQ)

Exchanges (all durable): - `domain.events` (**topic**) — public domain events (e.g., `order.created`, `payment.authorized`). - `domain.commands` (**topic**) — cross-service commands when you prefer async orchestration. - `domain.dlx` (**topic**) — dead-letter exchange for failures.

Queues (durable) & example bindings: - `orders.q` binds `domain.events` with `order.*` - `payments.q` binds `domain.events` with `payment.*` and `order.created` - `inventory.q` binds `domain.events` with `order.paid` and `inventory.command.*` - `notifications.q` binds `domain.events` with `order.paid`, `order.fulfilled`, `payment.failed`

Retry/DLQ pattern: for each consumer queue `X.q` create - `X.q.retry` (TTL = 30s/5m), DLX → main exchange routing key - `X.q.dlq` (final parking).
Use queue args:

```
"x-dead-letter-exchange": "domain.dlx",
"x-dead-letter-routing-key": "<original-rk>",
"x-message-ttl": 30000
```

Message metadata envelope (JSON):

```
{
  "eventId": "uuid",
  "type": "order.created",
  "occurredAt": "2025-09-09T13:00:00Z",
  "version": 1,
  "correlationId": "uuid",
  "causationId": "uuid",
  "payload": { /* event-specific fields */ }
}
```

- **Ordering:** only guaranteed per queue. If you need strict ordering per aggregate, route by key to a dedicated queue (or keep the aggregate's operations in one service).
- **Idempotency:** consumers must dedupe by `eventId` (store last processed IDs per consumer or per aggregate).

Local Dev Environment (Docker Compose)

Create `infra/docker-compose.yml` with: - `rabbitmq:3-management` (ports 5672, 15672) - `postgres:16` (port 5432) - `redis:7` (port 6379) - `prom/prometheus` + `grafana/grafana` (optional to start) - `otel/opentelemetry-collector` + `jaegertracing/all-in-one` (optional)

Example (minimal):

```
version: "3.9"
services:
  rabbitmq:
    image: rabbitmq:3.13-management
    ports: ["5672:5672", "15672:15672"]
    environment:
      RABBITMQ_DEFAULT_USER: dev
      RABBITMQ_DEFAULT_PASS: dev

  postgres:
    image: postgres:16
    ports: ["5432:5432"]
    environment:
      POSTGRES_USER: app
      POSTGRES_PASSWORD: app
      POSTGRES_DB: ecommerce
    volumes:
      - pgdata:/var/lib/postgresql/data

  redis:
    image: redis:7
    ports: ["6379:6379"]

volumes:
  pgdata:
```

Quick start:

```
cd infra
docker compose up -d
# RabbitMQ UI: http://localhost:15672 (dev/dev)
```

Project Structure (Multi-Module Maven)

```
ecom-eda/
├─ common-contracts/        # JSON Schemas, shared DTOs, validation
├─ common-lib/             # common: Jackson config, tracing, error types
├─ orders-service/         # REST + Outbox + Producer
├─ payments-service/       # Consumer + Producer (mock gateway)
├─ inventory-service/      # Consumer + Producer
├─ notifications-service/  # Consumer (email/SMS mock)
├─ bff-api/                # Read models + REST for frontend
└─ infra/                  # docker-compose, k8s manifests, Makefile
```

- Use **Spring Boot 3** in each service.
- All services depend on `common-contracts` and `common-lib`.

Shared Contracts (JSON Schema) & Versioning

Place JSON Schemas in `common-contracts/schemas` and generate POJOs if desired or validate at runtime.

Example: `order.created.v1.schema.json`

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://ecom.example/schemas/order.created.v1.json",
  "type": "object",
  "required": ["eventId", "type", "occurredAt", "version", "payload"],
  "properties": {
    "eventId": {"type": "string", "format": "uuid"},
    "type": {"const": "order.created"},
    "occurredAt": {"type": "string", "format": "date-time"},
    "version": {"type": "integer", "const": 1},
    "correlationId": {"type": "string"},
    "causationId": {"type": "string"},
    "payload": {
      "type": "object",
      "required": ["orderId", "userId", "items", "total"],
      "properties": {
        "orderId": {"type": "string"},
        "userId": {"type": "string"},
        "items": {
          "type": "array",
          "items": {
            "type": "object",
            "required": ["productId", "qty", "price"],
            "properties": {
              "productId": {"type": "string"},
              "qty": {"type": "integer", "minimum": 1},
              "price": {"type": "number", "minimum": 0}
            }
          }
        },
        "total": {"type": "number", "minimum": 0}
      }
    }
  }
}
```

Schema evolution: bump `version` and keep consumers backward-compatible. Maintain both `v1`, `v2` during rollout.

Spring Boot Setup

Common dependencies (pom.xml):

```
<dependencies>
  <dependency>
```

```
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>
  <dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-json</artifactId>
    <optional>true</optional>
  </dependency>
</dependencies>
```

application.yaml (example):

```
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/ecommerce
    username: app
    password: app
  jpa:
    hibernate:
      ddl-auto: update
    open-in-view: false
  rabbitmq:
    host: localhost
    port: 5672
    username: dev
    password: dev
```

```
management:
  endpoints:
    web:
      exposure:
        include: "health,info,prometheus"
```

Transactional Outbox Pattern (Exactly-once Semantics)

Why: avoid **dual-write** (DB commit succeeds but message publish fails, or vice-versa). Strategy: 1. In the same DB transaction where you persist the aggregate (e.g., `Order`), insert an **Outbox** row. 2. A **publisher** reads un-sent outbox rows and publishes to RabbitMQ, then marks them as **SENT** (with confirm).

Outbox table:

```
CREATE TABLE outbox (
  id BIGSERIAL PRIMARY KEY,
  aggregate_type VARCHAR(64) NOT NULL,
  aggregate_id VARCHAR(64) NOT NULL,
  event_type VARCHAR(64) NOT NULL,
  event_version INT NOT NULL,
  payload JSONB NOT NULL,
  event_id UUID NOT NULL UNIQUE,
  occurred_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
  status VARCHAR(16) NOT NULL DEFAULT 'PENDING', -- PENDING | SENT | FAILED
  attempts INT NOT NULL DEFAULT 0
);
```

Publisher strategies: - **Polling scheduler** (every ~200ms–1s) — simplest. - **DB notify/listen** (Postgres) — push style. - Use **publisher confirms** in RabbitMQ to ensure delivery; retry with backoff.

Publishing Events (Producers)

Rabbit configuration (Orders service):

```
@Configuration
public class RabbitConfig {
  public static final String DOMAIN_EXCHANGE = "domain.events";

  @Bean
  TopicExchange domainEventsExchange() { return new
TopicExchange(DOMAIN_EXCHANGE, true, false); }

  @Bean
  Jackson2JsonMessageConverter jacksonConverter() { return new
Jackson2JsonMessageConverter(); }

  @Bean
  RabbitTemplate rabbitTemplate(ConnectionFactory cf,
Jackson2JsonMessageConverter conv) {
    RabbitTemplate tpl = new RabbitTemplate(cf);
    tpl.setMessageConverter(conv);
    tpl.setMandatory(true);
    tpl.setReturnsCallback(r -> {/* log returned */});
    tpl.setConfirmCallback((corr, ack, cause) -> {/* mark SENT/FAILED */});
    return tpl;
  }
}
```

Outbox publisher (simplified):

```
@Service
@RequiredArgsConstructor
public class OutboxPublisher {
  private final OutboxRepository repo; // Spring Data JPA
  private final RabbitTemplate rabbit;

  @Transactional
  @Scheduled(fixedDelayString = "500")
  public void publishPending() {
    List<Outbox> batch = repo.lockNextPendingBatch(PageRequest.of(0, 100));
    for (Outbox o : batch) {
      try {
        var envelope = Map.of(
            "eventId", o.getEventId().toString(),
            "type", o.getEventType(),
            "occurredAt", o.getOccurredAt(),
            "version", o.getEventVersion(),
            "payload", o.getPayload());
        rabbit.convertAndSend(RabbitConfig.DOMAIN_EXCHANGE,
o.getEventType(), envelope);
        o.setStatus("SENT");
      } catch (Exception ex) {
        o.setAttempts(o.getAttempts() + 1);
        o.setStatus(o.getAttempts() > 5 ? "FAILED" : "PENDING");
      }
    }
  }
}
```

Creating an order (controller → service):

```
@Service
@RequiredArgsConstructor
public class OrderService {
  private final OrderRepository orders;
  private final OutboxRepository outbox;

  @Transactional
  public Order create(CreateOrderCmd cmd) {
    Order o = Order.create(cmd);
    orders.save(o);

    Outbox evt = Outbox.from(
      aggregateType: "Order", aggregateId: o.getId(),
      eventType: "order.created", eventVersion: 1,
      payload: Map.of("orderId", o.getId(), "userId", o.getUserId(),
"items", o.getItems(), "total", o.getTotal())
    );
    outbox.save(evt);

    return o;
  }
}
```

Consuming Events (Consumers, Retries & DLQ)

Queue topology (Inventory service):

```
@Configuration
public class InventoryRabbit {
  static final String Q = "inventory.q";
  static final String Q_RETRY = "inventory.q.retry";
  static final String Q_DLQ = "inventory.q.dlq";

  @Bean
  Queue main() { return QueueBuilder.durable(Q)
    .withArgument("x-dead-letter-exchange", "domain.dlx")
    .withArgument("x-dead-letter-routing-key", "inventory.retry")
    .build(); }

  @Bean Queue retry() { return QueueBuilder.durable(Q_RETRY)
    .withArgument("x-dead-letter-exchange", "domain.events")
    .withArgument("x-dead-letter-routing-key", "order.paid")
    .withArgument("x-message-ttl", 30000)
    .build(); }

  @Bean Queue dlq() { return QueueBuilder.durable(Q_DLQ).build(); }
  @Bean Binding bindMain(TopicExchange domainEventsExchange) {
    return
BindingBuilder.bind(main()).to(domainEventsExchange).with("order.paid");
  }

  @Bean Binding bindDlx(TopicExchange dlx) {
    return BindingBuilder.bind(dlq()).to(dlx).with("inventory.retry");
  }

  @Bean TopicExchange dlx() { return new TopicExchange("domain.dlx", true,
false); }
}
```

Idempotent consumer with `@RabbitListener`:

```
@Service
@RequiredArgsConstructor
public class InventoryConsumer {
  private final ProcessedEventRepository processed;
  private final InventoryService inventory;

  @RabbitListener(queues = InventoryRabbit.Q)
  public void onOrderPaid(Map<String, Object> envelope, Channel channel,
Message message) throws Exception {
    String eventId = (String) envelope.get("eventId");
    if (processed.existsById(eventId)) {
      channel.basicAck(message.getMessageProperties().getDeliveryTag(),
false);
      return; // idempotent
    }

    try {
      Map<String, Object> payload = (Map<String, Object>)
envelope.get("payload");
      inventory.reserveStock(payload);
      processed.save(new ProcessedEvent(eventId));
      channel.basicAck(message.getMessageProperties().getDeliveryTag(),
false);
    } catch (TransientException ex) {
      channel.basicNack(message.getMessageProperties().getDeliveryTag(),
false, false); // send to DLX → retry/dlq
    } catch (Exception ex) {
      channel.basicNack(message.getMessageProperties().getDeliveryTag(),
false, false);
    }
  }
}
```

Processed events table:

```
CREATE TABLE processed_events (
  event_id UUID PRIMARY KEY,
  processed_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
```

Materialized Views & Read Models

- Update **read models** (Postgres tables or Redis hashes) in consumers.
- Examples: `order_summary` view for BFF; `product_availability` cache in Redis.
- Use **compaction keys** (e.g., upsert by `orderId`) to keep latest state.

HTTP APIs (OpenAPI + Controllers)

Start with a small HTTP surface:

- **Orders Service**
- `POST /orders` → create order
- `GET /orders/{id}` → current order state
- **BFF**
- `GET /me/orders` → list of order summaries

OpenAPI fragment (orders):

```
openapi: 3.0.3
info: { title: Orders API, version: 1.0.0 }
paths:
  /orders:
    post:
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/CreateOrder'
      responses:
        '201': { description: Created }
  /orders/{id}:
    get:
      parameters:
        - in: path
          name: id
          schema: { type: string }
          required: true
      responses:
        '200':
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Order'
components:
  schemas:
    CreateOrder:
      type: object
      required: [userId, items]
      properties:
        userId: { type: string }
        items:
          type: array
          items:
            type: object
            required: [productId, qty]
            properties:
              productId: { type: string }
              qty: { type: integer, minimum: 1 }
    Order:
      type: object
      properties:
        id: { type: string }
        status: { type: string }
        total: { type: number }
```

Security (JWT/OAuth2)

- Use **OAuth2 Resource Server** in services to validate JWT (e.g., Keycloak, Auth0, Cognito).
- At minimum, validate a **signed JWT** on write APIs; allow public reads if appropriate.
- Propagate **correlationId** and user claims into event metadata.

Example application.yaml snippet:

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          jwk-set-uri: http://localhost:8080/realms/ecom/protocol/openid-connect/certs
```

Observability (Metrics, Logs, Tracing)

- Enable **Spring Actuator** and **Prometheus** endpoint `/actuator/prometheus`.
- Use **OpenTelemetry Java agent** (`-javaagent:opentelemetry-javaagent.jar`) to export traces to local Jaeger/OTel collector.

- Use **JSON logs** with fields: `eventId`, `correlationId`, `routingKey`, `queue`, `service`.
- Alert on **queue depth**, **consumer rejections**, **outbox backlog**.

Testing (Unit, Integration, Contract)

- **Unit tests**: domain logic (state transitions), outbox creation.
- **Integration tests**: with **Testcontainers** for Postgres + RabbitMQ; publish/consume roundtrip.
- **Consumer contract tests**: validate your consumer against **JSON Schemas**.

Testcontainers example (Orders):

```
@Container static RabbitMQContainer rabbit = new RabbitMQContainer("rabbitmq:3.13-management");
@Container static PostgreSQLContainer<?> pg = new PostgreSQLContainer<>("postgres:16");
```

Packaging, CI/CD & Deploy

- **Dockerfile (multi-stage):** build JAR, run as non-root; enable JVM flags.
- **Kubernetes manifests:** `Deployment`, `Service`, `ConfigMap` for configs, `Secret` for creds.
- **Helm**: optional chart per service.
- **CI:**
- compile → unit tests → integration (Testcontainers) → build image → push → deploy.
- run schema validators as part of PR checks.

Dockerfile (Spring Boot):

```
FROM maven:3.9-eclipse-temurin-21 AS build
WORKDIR /app
COPY pom.xml .
COPY src src
RUN mvn -q -DskipTests package

FROM eclipse-temurin:21-jre
RUN useradd -ms /bin/bash app
USER app
WORKDIR /home/app
COPY --from=build /app/target/*.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-XX:MaxRAMPercentage=75", "-jar", "app.jar"]
```

Runbook: Operations

RabbitMQ - Check UI (15672) → Queues: depth, unacked, consumers. - If `*.dlq` grows: inspect payloads, fix consumer, requeue from DLQ. - Scale by adding consumer instances (horizontal pods) — RabbitMQ will round-robin.

Outbox - Monitor `PENDING` count; if growing, publisher is stuck or RabbitMQ unreachable. - Investigate `FAILED` rows; implement backoff and alert.

DB & Cache - Watch slow queries, add indexes on foreign keys (`order_id`, `user_id`). - Redis memory policy: use `volatile-lru` for TTL'd keys (e.g., carts).

Security - Rotate credentials; use per-service RabbitMQ users and **vhosts**. - Apply network policies; only services talk to RabbitMQ.

Appendix: Full Example Snippets

1) Entities (Orders)

```
@Entity
@Table(name = "orders")
@Data @NoArgsConstructor
public class Order {
  @Id private String id;
  private String userId;
  private String status; // CREATED, PAID, FULFILLED, CANCELLED
  private BigDecimal total;
  // items stored in separate table or JSONB column
  public static Order create(CreateOrderCmd cmd) {
    Order o = new Order();
    o.id = UUID.randomUUID().toString();
    o.userId = cmd.userId();
    o.status = "CREATED";
    o.total = cmd.total();
    return o;
  }
}
```

2) Commands & DTOs

```
public record CreateOrderCmd(String userId, List<Item> items, BigDecimal
total) {}
public record Item(String productId, int qty, BigDecimal price) {}
```

3) Outbox Entity & Repository

```
@Entity
@Table(name = "outbox")
@Data @NoArgsConstructor
public class Outbox {
  @Id @GeneratedValue private Long id;
  private String aggregateType;
  private String aggregateId;
  private String eventType;
  private Integer eventVersion;
  @Column(columnDefinition = "jsonb") private String payload;
  private UUID eventId = UUID.randomUUID();
  private Instant occurredAt = Instant.now();
  private String status = "PENDING";
  private Integer attempts = 0;

  public static Outbox from(String aggregateType, String aggregateId, String eventType, int eventVersion, Object payload) {
    Outbox o = new Outbox();
    o.aggregateType = aggregateType;
    o.aggregateId = aggregateId;
    o.eventType = eventType;
    o.eventVersion = eventVersion;
    try { o.payload = new ObjectMapper().writeValueAsString(payload); } catch (Exception e) { throw new RuntimeException(e); }
    return o;
  }
}

public interface OutboxRepository extends JpaRepository<Outbox, Long> {
  @Query(value =
"SELECT * FROM outbox WHERE status='PENDING' ORDER BY id FOR UPDATE SKIP LOCKED LIMIT :limit", nativeQuery = true)
  List<Outbox> lockNextPendingBatch(@Param("limit") int limit);
}
```

4) Controllers (Orders)

```
@RestController
@RequestMapping("/orders")
@RequiredArgsConstructor
public class OrdersController {
  private final OrderService svc;

  @PostMapping
  public ResponseEntity<Order> create(@Valid @RequestBody CreateOrderCmd cmd) {
    Order o = svc.create(cmd);
    return ResponseEntity.status(HttpStatus.CREATED).body(o);
  }

  @GetMapping("/{id}")
  public ResponseEntity<Order> get(@PathVariable String id) {
    return
svc.findById(id).map(ResponseEntity::ok).orElse(ResponseEntity.notFound().build());
  }
}
```

5) Consumer Retry Advice (Spring AMQP)

```
@Bean
public SimpleRabbitListenerContainerFactory
rabbitListenerContainerFactory(ConnectionFactory cf,
Jackson2JsonMessageConverter conv) {
  SimpleRabbitListenerContainerFactory f = new
SimpleRabbitListenerContainerFactory();
  f.setConnectionFactory(cf);
  f.setMessageConverter(conv);
  f.setDefaultRequeueRejected(false); // use DLX, not broker requeue
  f.setPrefetchCount(50);
  f.setConcurrentConsumers(2);
  f.setMaxConcurrentConsumers(8);
  return f;
}
```

6) Redis Read Model (BFF)

```
@Service
@RequiredArgsConstructor
public class OrderReadModelService {
  private final StringRedisTemplate redis;

  public void updateOrderSummary(OrderSummary s) {
    HashOperations<String, String, String> h = redis.opsForHash();
    String key = "order:" + s.id();
    h.put(key, "status", s.status());
    h.put(key, "total", s.total().toString());
    redis.expire(key, Duration.ofHours(24));
  }
}
```

7) Json Validation (Optional)

```
public class JsonSchemaValidator {
  private final JsonSchema schema;
  public JsonSchemaValidator(JsonSchema schema) { this.schema = schema; }
  public void validate(JsonNode node) {
    Set<ValidationMessage> errors = schema.validate(node);
    if (!errors.isEmpty()) throw new
IllegalArgumentException(errors.toString());
  }
}
```

Getting Started Checklist

1. **Spin up infra**: `docker compose up -d` (RabbitMQ, Postgres, Redis).
2. **Bootstrap** `orders-service` with Spring Boot and dependencies.
3. **Create DB schema** (orders + outbox + processed_events).
4. **Implement create order** (controller + service + outbox write).
5. **Implement outbox publisher** (scheduled) → publish `order.created`.
6. **Implement** `payments-service` consumer for `order.created` → emit `payment.authorized` (mock).
7. **Implement** `inventory-service` consumer for `order.paid` → reserve stock (idempotent) → emit `inventory.reserved`.
8. **Add DLQ bindings** and verify retry.
9. **Add metrics & health** endpoints; verify in logs/UI.
10. **Write integration tests** with Testcontainers.

You now have a runnable, extensible EDA backend with RabbitMQ.