System Design Day 4: Event-Driven and Payment Systems

📘 Event-Driven Architecture (EDA) Summary

✅ Definition

Event-Driven Architecture is a design pattern where services communicate through **events** rather than direct calls. An event is emitted when something happens (e.g., user logged in, order placed).

🔄 Message-Driven vs Event-Driven

Feature	Message-Driven	Event-Driven
Intent	Command to do something	Notification of something that happened
Targeted Service	Yes (directed to a specific service)	No (published for anyone to consume)
Mutability	Deleted after processing	Immutable event log
Dependency	Tightly coupled	Loosely coupled

🧱 Core Components

1. **Producer** – Publishes events after a state change
2. **Event Broker** – Distributes events to subscribers (e.g., Kafka)
3. **Consumer** – Listens to specific events and processes them

📌 Pub/Sub Model

- **Publisher**: Emits events without knowing who listens.
- **Broker**: (Kafka) tracks offsets and delivers events.
- **Subscriber**: Registers interest and processes relevant events.

🧠 When to Use

- Auditing/logging user activity
- Asynchronous workflows (e.g., sending confirmation emails)
- Analytics and machine learning ingestion
- Background processing
- Data sync across microservices

🚀 Advantages

- ✅ **Loose Coupling** – Services don’t depend directly on each other
- ✅ **Scalable** – Easily add consumers without impacting publishers
- ✅ **Resilient** – Failures in consumers don’t break the system
- ✅ **Extensible** – Add new event-driven use cases without changing existing services
- ✅ **Dependency Inversion** – Services depend on event contracts, not implementations

---

⚠️ Disadvantages

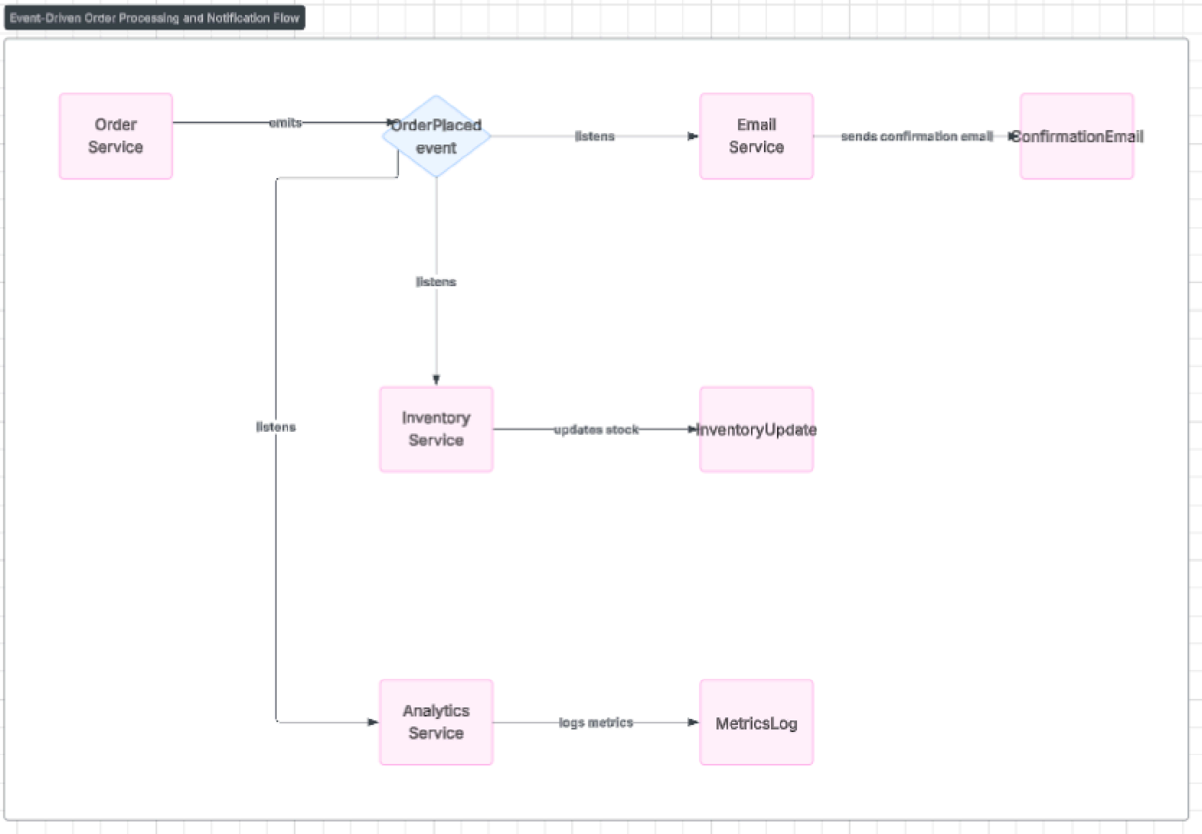
- ⏱️ **Eventual Consistency** – Delays in reflecting the latest state
- 🔁 **Duplicate Events** – Consumers must be idempotent
- 🧩 **Operational Complexity** – More components (brokers, schemas, DLQs, etc.)
- 🐞 **Debugging Difficulty** – Harder to trace events end-to-end

---

🧰 Design Tips

- Use **unique event IDs** for idempotency
- Apply **retry & backoff** strategies on consumer failure
- Maintain a **Dead Letter Queue (DLQ)** for poison messages
- Integrate **observability** (logs, metrics, traces)
- Use **Schema Registry** for consistency in event formats

🧪 Use Case Example: Order Processing



☕ Relate to Java

- ✅ **Used Spring Kafka** to build producers and consumers for Kafka topics.
- ✅ Implemented message listeners using `@KafkaListener`.
- ✅ Managed event serialization/deserialization with **Avro/JSON**.
- ✅ Used **Schema Registry** to validate event formats.
- ✅ Ensured **idempotent event handling** using unique IDs and deduplication logic.
- ✅ Configured **DLQs**, **retry templates**, and **error handling strategies** for reliability.
- ✅ Used **Spring Boot Actuator** and **Micrometer** for monitoring Kafka interactions.

💳 Payment System Design – Full Summary (with JPA & Atomicity)

📖 Definition

A **payment system design** is the architectural and technical strategy used to handle financial transactions between buyers and sellers while ensuring:

- **Security & compliance** (PCI DSS, GDPR).
- **Atomicity** and **idempotency**.
- **Reliability**, **fault tolerance**, and **scalability**.
- **Integration with PSPs**, banks, and third-party services.
- Recovery from **partial failures** and **network issues**.

🧠 High-Level Payment Flow

1. **User places order** → Redirected to payment page.
2. **Payment Gateway (PSP)** handles:
    - Card details collection.
    - Compliance (PCI, GDPR).
    - Fraud detection (risk engine).
3. **Acquiring Bank** processes card info via **Card Network (Visa/Mastercard)**.
4. **Issuing Bank** verifies and approves or declines.
5. Status is returned to **Merchant**, and user is notified.

🧩 Core Components

Component	Role
Payment Service	Initiates payment, calls PSP, emits events
Wallet Service	Updates merchant balance post payment success
Ledger Service	Logs all financial transactions immutably
Kafka or RabbitMQ	Handles async communication, retries, decoupling

Component	Role
PSP Integration	Stripe, Razorpay, PhonePe, Google Pay
Notification	Email/SMS on payment result
Reconciliation Job	Nightly job to match PSP settlement vs internal DB

📌 Functional & Non-Functional Requirements

✅ Functional

- Accept payment requests.
- Process and validate transactions.
- Update account balance.
- Generate transaction records.

⚙️ Non-Functional

- **High availability** (e.g., 99.99% uptime).
- **Low latency** (< 100 ms critical path).
- **Fault tolerance** (service failures shouldn't affect the system).
- **Scalability** (handle traffic spikes).
- **Security** (TLS, tokenization, encryption).

⚙️ JPA and Transactions in Payment System

Why JPA?

- JPA simplifies data persistence using entities and repositories.
- JPA is transactional and declarative via `@Transactional`.

Typical Use Case in Payment:

```
@Transactional
public void processPayment(PaymentRequest request) {
    paymentRepository.save(request.toEntity());
    walletService.debit(request.getMerchantId(), request.getAmount());
```

```
    ledgerService.logTransaction(request);
}
```

- Ensures **atomicity** at the database level.
- If anything fails (e.g., DB insert or external call), the entire operation is **rolled back**.

🔒 Ensuring Atomicity in Payment

Goal:

"Either all steps in a transaction succeed or none do."

Technique	Description
Spring @Transactional	For local DB atomicity. Rollbacks on failure.
Idempotency Key	Prevents double charges on retries.
Event Sourcing + Kafka	Use events for state changes → durable, replayable.
SAGA (Sequence of Asynchronous Gateway Activities) Pattern	For multi-step workflows across services (e.g., payment + wallet + ledger).
Compensation logic	Reverse incomplete actions if any subtask fails.

🔄 Async vs Sync Communication

Type	Pros	Cons	Usage
Sync	Immediate feedback	Tightly coupled, fragile	Card validation, real-time gateways
Async	Decoupled, fault-tolerant, scalable	Eventual consistency	Wallet update, ledger logging, audit

> For large-scale systems, asynchronous with retries is preferred.

🔁 Saga Pattern (Orchestration Example)

> SAGA (Sequence of Asynchronous Gateway Activities) is a pattern to manage distributed transactions across microservices via a series of local transactions with compensating actions.

Workflow Example:

1. **Payment Service**: create order & call PSP.
2. On PSP success: emit `PaymentSuccessEvent`.
3. **Wallet Service**: listens, updates balance.
4. **Ledger Service**: listens, stores log entry.
5. If any step fails → trigger **compensation logic** (e.g., reverse wallet update).

💡 Retry Patterns

- Use **Exponential Backoff with Jitter** to avoid retry storms.
- **Transient failures** (e.g., network blips) → retry.
- **Persistent failures** → Dead Letter Queue.
- Implement **Timeouts** to prevent hanging requests.

🧱 Idempotency (Prevent Double Payments)

- Generate a **UUID idempotency key** at the client.
- On retry, server checks if key exists → avoid re-processing.
- Enforce uniqueness via **DB primary/unique key**.
- Stripe, PayPal, Razorpay use this model.

🔐 Security Architecture

Concern	Solution
Data in transit	TLS, HTTPS, VPN
Data at rest	Disk/DB encryption
Authentication/Access	Role-based auth, 2FA

Concern	Solution
Compliance	PCI DSS, GDPR
Software vulnerabilities	Patch regularly
Password safety	Use hashed + salted passwords

🧪 Data Integrity Monitoring

- Use **Checksums** for DB/file verification.
- Monitor for unauthorized changes.
- Focus on **high-risk data**: credentials, config, key stores.
- Expensive → optimize by monitoring only sensitive resources.

🕸️ Distributed System Considerations

Challenge	Solution
Node failures	Replication, redundancy
Traffic spikes	Load balancing, autoscaling
Data inconsistency	Consistency levels, read/write tuning
Large traffic volume	Horizontal scaling, partitioning

🧠 Final Best Practices Recap

Area	Practice
Atomicity	Use JPA transactions, idempotency keys
Scalability	Use Kafka, async services, retry queues
Reliability	Dead-letter queues, compensation actions, audit logs
Security	TLS, HTTPS, disk encryption, auth controls
Compliance	Avoid storing card data; offload to PSP

🧠 Real-World Payment Flow: Stripe/Razorpay

Here's a **step-by-step walkthrough** of how a payment transaction flows in a system like **Stripe** or **Razorpay**:

💼 Scenario:

A customer wants to pay ₹500 for a product on an e-commerce site.

🔄 Step-by-Step Flow

Step	Service/Component	Action
1	**Frontend Checkout**	Customer fills card details or selects UPI/netbanking.
2	**Payment Gateway (Stripe/Razorpay)**	Securely collects payment credentials via PCI-compliant SDK.
3	**Tokenization**	Card details converted to a token — never stored in merchant DB.
4	**Payment Request**	Frontend POSTs payment request to merchant backend with token.
5	**Merchant Backend**	Validates session, stores payment request with status: `PENDING`.
6	**Initiate PSP Call**	Calls Stripe/Razorpay `/charges` or `/order/pay` API with token.
7	**Payment Processing**	PSP routes to acquiring bank → card network → issuing bank.
8	**Response from PSP**	PSP returns status: `success`, `failed`, or `pending`.
9	**Event Emission**	Emit `PaymentSuccessEvent` or `PaymentFailedEvent` to Kafka.
10	**Wallet Service**	Listens to event, updates merchant balance (eventually consistent).
11	**Ledger Service**	Stores immutable transaction log entry.
12	**Notification Service**	Sends SMS/email/WhatsApp to customer.
13	**Reconciliation Service** (nightly)	Verifies actual bank settlement vs internal records.

🎯 Design Observations

- Razorpay/Stripe handle **PCI** and **bank integrations**, so the merchant **never stores sensitive data**.
- Idempotency is **key** — PSP call is retried safely using `X-Idempotency-Key`.

- Real-time actions (e.g., success page) use **synchronous call** + webhooks.
- Backend state changes and services (wallet, ledger) rely on **event-driven** updates.

💬 Razorpay `order` example:

```
json
CopyEdit
POST /orders
{
  "amount": 50000,
  "currency": "INR",
  "receipt": "rcptid_11",
  "payment_capture": true}
```

- Response includes `order_id`, which is passed to the frontend to continue payment.
- After payment, webhook is triggered: `payment.captured`.

🧩 Components in Architecture

Component	Role
Payment Service	Initiates payment, calls PSP, emits events
Wallet Service	Updates merchant balance post payment success
Ledger Service	Logs all financial transactions immutably
Kafka or RabbitMQ	Handles async communication, retries, decoupling
PSP Integration	Stripe, Razorpay, PhonePe, Google Pay
Notification	Email/SMS on payment result
Reconciliation Job	Nightly job to match PSP settlement vs internal DB
