Technical Design and Architecture Document (TDAD): Simple Payment Processing Service (PPS)

1. System Architecture: Layered Monolith

The PPS will be implemented as a **Layered Spring Boot Monolith** structured for high cohesion and low coupling. This architecture choice ensures rapid development while maintaining clean separation for future migration to microservices.

Layer Breakdown

1. **Controller Layer (API Gateway):** Exposes RESTful endpoints. Responsible for request validation and serialization/deserialization. Maps external requests to internal service calls.
2. **Service Layer (Business Logic):** Contains core logic, including idempotency checks, price calculation (if needed), status transitions, and coordinating with the Gateway layer.
3. **Gateway Layer (External Abstraction):** Handles all communication with third-party PGs (e.g., PaystackClient, FlutterwaveClient). Responsible for translating the PPS's internal request format into the PG's specific API call and handling vendor-specific responses/errors.
4. **Repository Layer (Data Persistence):** Manages interactions with the PostgreSQL database. Uses Spring Data JPA for CRUD operations on the Transaction model.

2. Data Modeling (PostgreSQL)

The core persistence strategy relies on three main entities.

2.1. Transaction Entity (Core Record)

Field Name	Type	Constraints	Description
id	UUID	PK, Generated	Internal unique ID for the transaction.
idempotencyKey	String (255)	Unique Index	Merchant-supplied key to prevent duplicate processing (crucial).
merchantRef	String (255)	Index	Merchant's unique reference for the order.
amount	Decimal	Required	Transaction amount.
currency	Enum	Required	NGN, USD, etc.

pgTransactionRef	String (255)	Index	The reference ID returned by the external Payment Gateway.
status	Enum	Required	PENDING, SUCCESS, FAILED, PROCESSING.
paymentMethod	String		CARD, TRANSFER, USSD.
metadata	JSONB		Flexible storage for non-critical gateway-specific data.
createdAt	Instant		Timestamp of creation.

2.2. Merchant Entity

Used for basic API key management and identifying the calling client.

2.3. WebhookEvent Entity

Used to log and track all inbound and outbound webhook communications for auditability and replay capability.

3. Key Technical Implementations

3.1 Idempotency Guarantee (High Priority)

1. On receiving a request, the **Controller** extracts the IdempotencyKey from the header or payload.
2. The **Service Layer** attempts to find a record with this key.
3. If a record exists and the status is SUCCESS or PENDING, the original transaction response is returned without re-executing the payment logic.
4. If no record exists, a new Transaction record is created with a status of PENDING and the IdempotencyKey is locked (using a database transaction or Redis).
5. The transaction logic proceeds.

3.2 Asynchronous Processing (Reliability)

Webhooks are external events and should be handled asynchronously to prevent blocking the receiving thread.

- **Technology:** We will use a dedicated **WebhookController** for receiving PG notifications. This controller will immediately save the raw event payload to the database (WebhookEvent table) and publish a message to an internal **JMS Queue (simulated via Spring @Async or a dedicated Kafka/Redis layer in production)**.
- A dedicated **WebhookListener** will consume this message, verify the payload signature, and update the associated Transaction status.

3.3 Security Considerations (Java & Spring)

1. **Spring Security:** Mandatory. Implement Basic Auth or API Key validation (using Merchant entity) on all incoming requests to the PPS API.
2. **Environment Variables:** All sensitive keys (PG API keys, Database credentials) must be loaded from secure environment variables, *not* stored in application properties files.
3. **Data at Rest:** Use Java's encryption utilities (e.g., Jasypt or custom Spring integration) to encrypt non-card sensitive fields (like customer email/address) in the database.

4. System Design Diagram (Architectural Flow)

The system is designed around two primary flows: **Transaction Initiation** (synchronous) and **Webhook Processing** (asynchronous).