**Asynchronous Communication Using Kafka**

​The solution demonstrates the implementation of the Outbox and Inbox patterns using Apache Kafka, aiming to ensure reliable and consistent message processing in distributed systems. It also includes a generic logging structure to log the message id and the request trace id for aggregate display of the logging related to a request and a message.

**🔄 Overview of the Outbox and Inbox Patterns**

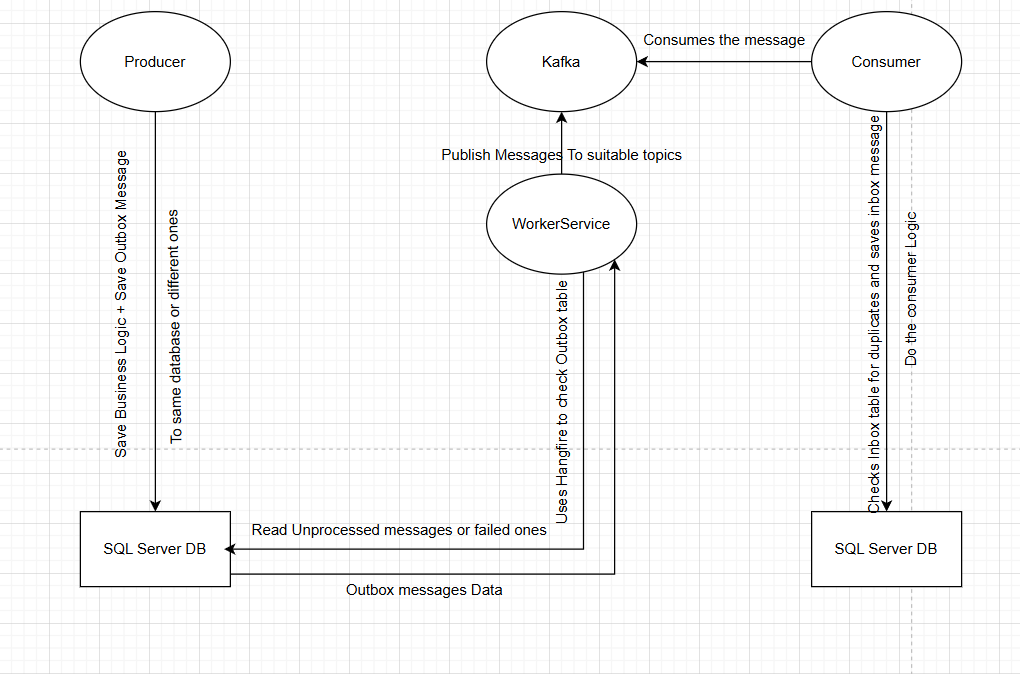
**Outbox Pattern**

The Outbox Pattern ensures that changes to the database and the emission of corresponding events to Kafka occur within a single transaction. Instead of sending events directly to Kafka, the application writes them to an "outbox" table in the database. A separate process ‘WorkerService’ then reads from this table and publishes the events to Kafka, ensuring consistency and reliability.

**Inbox Pattern**

The Inbox Pattern handles incoming messages from Kafka, ensuring that each message is processed exactly once. It involves recording the receipt and processing status of messages in an "inbox" table, preventing duplicate processing and maintaining idempotency.​

**🧭 Detailed Flow Description**



**1. Business Operation and Outbox Entry Creation**

When a business operation occurs (e.g., creating an order), the application performs the following within a single transaction:​

* Updates the relevant business data in the database.
* Creates a new entry in the outbox table with details of the event to be published.​

This ensures that either both operations succeed, or neither does, maintaining consistency.​

**2. Outbox Event Publishing**

A WorkerService checks the outbox table for new entries every few seconds. Upon detecting a new event or failed events that didn’t pass the retry threshold:​

* It reads the event data from the outbox table.
* Publishes the event to the appropriate Kafka topic.

 On success:

* Updates ProcessedOn with the timestamp.

 On failure:

* Increments RetryCount.
* Stores error message in Error.

This decouples the event publishing from the main business logic, allowing for retries and fault tolerance.​

**3. Kafka Consumer and Inbox Processing**

Services consuming events from Kafka perform the following:

* Upon receiving a message, they check the inbox table to determine if the message has already been processed.
* If not, they process the message and record its processing status in the inbox table.
* This ensures that each message is processed exactly once, even in the face of retries or failures.

**🧱 Components Breakdown**

**Outbox Message Structure**

public class OutboxMessage

{

public Guid Id { get; set; }

public Guid? TraceId { get; set; } // Correlation ID

public DateTime OccurredOn { get; set; } // Event creation time

public string Type { get; set; } = null!; // Event type (e.g., "OrderCreated")

public string Content { get; set; } = null!; // Serialized event data (e.g., JSON)

public DateTime? ProcessedOn { get; set; } // Time when sent to Kafka

public int RetryCount { get; set; } // Retry attempts in case of failures

public string? Error { get; set; } // Last error encountered (if any)

}

**Inbox Message Structure**

public class InboxMessage

{

public Guid Id { get; set; } // Kafka Message ID

public Guid? TraceId { get; set; } // Correlation ID for tracking

public DateTime ReceivedOn { get; set; } // Time received from Kafka

public string ConsumerType { get; set; } = null!; // Name of consumer class

}

**✅ Benefits of This Implementation**

* **Atomicity**: Ensures that database changes and event publishing occur together.
* **Reliability**: Decouples event publishing, allowing for retries and fault tolerance.
* **Idempotency**: Prevents duplicate processing of messages.
* **Scalability**: Supports distributed systems and microservices architectures.