**White Paper: Building a Resilient Kafka Producer Using the Transactional Outbox Pattern and Kafka Consumer in Event Store Pattern in .NET Core**

**Abstract**

In modern microservices architectures, ensuring reliable communication between services is critical. Apache Kafka is a popular distributed streaming platform that supports building real-time data pipelines and streaming applications. However, integrating Kafka with microservices can be challenging, especially when ensuring data consistency and resilience in case of failures. This white paper presents a solution to these challenges by implementing the Transactional Outbox Pattern for Kafka producers and the Event Store Pattern for Kafka consumers in a .NET Core environment. We will provide a detailed overview of these patterns, followed by a complete implementation example in .NET Core.

**Table of Contents**

1. Introduction
2. Problem Statement
3. Overview of Patterns
   * Transactional Outbox Pattern
   * Event Store Pattern
4. Implementation in .NET Core
   * Setting Up Kafka in .NET Core
   * Implementing the Transactional Outbox Pattern
   * Implementing the Event Store Pattern
   * Integration Testing
5. Best Practices
6. Conclusion
7. References

**1. Introduction**

Apache Kafka is widely used for building robust and scalable messaging systems. However, building a resilient Kafka producer and consumer involves ensuring that messages are delivered exactly once, even in the presence of failures. This paper focuses on two key patterns to achieve this: the Transactional Outbox Pattern for producers and the Event Store Pattern for consumers. The goal is to provide a robust and resilient messaging system in a microservices architecture using .NET Core.

**2. Problem Statement**

In distributed systems, ensuring data consistency and reliability in the face of network partitions, service crashes, or other failures is crucial. Kafka, being a distributed platform, can face issues where messages are lost or duplicated if not handled correctly. The primary challenges include:

1. **Atomicity:** Ensuring that a message is sent to Kafka only if the database transaction is successful.
2. **Consistency:** Guaranteeing that messages are processed exactly once by consumers.
3. **Resilience:** Ensuring the system recovers gracefully from failures.

**3. Overview of Patterns**

**Transactional Outbox Pattern**

The Transactional Outbox Pattern ensures that database changes and sending messages to Kafka are handled in a single transaction. This pattern helps in maintaining consistency between the database and Kafka by using an outbox table to temporarily store messages that need to be sent.

**How It Works:**

1. **Outbox Table:** A separate table in the database is used to store messages that need to be sent to Kafka.
2. **Transaction:** The database operations and the insertion into the outbox table happen within the same transaction.
3. **Message Dispatcher:** A background service reads from the outbox table and sends messages to Kafka.
4. **Cleanup:** Once the message is successfully sent, it is removed from the outbox table.

**Event Store Pattern**

The Event Store Pattern is used on the consumer side to store all the events that have been consumed from Kafka. This pattern ensures that events are processed exactly once, even if the consumer crashes or there are network failures.

**How It Works:**

1. **Event Store Table:** A table in the database that stores the events that have been consumed.
2. **Idempotency:** Before processing an event, the consumer checks if the event has already been processed by querying the event store.
3. **Processing:** If the event is new, it is processed and then stored in the event store table.
4. **Ack:** After processing and storing the event, the consumer acknowledges the message in Kafka.

**4. Implementation in .NET Core**

**Setting Up Kafka in .NET Core**

To begin, we need to set up Kafka and configure it in a .NET Core application. We can use the Confluent.Kafka library for this purpose.

1. **Install Kafka and Zookeeper:** Kafka requires Zookeeper to manage its clusters. You can install Kafka and Zookeeper locally or use Docker.
2. **Install the Confluent.Kafka NuGet package:**

bash

Copy code

dotnet add package Confluent.Kafka

1. **Configuration:** Configure Kafka in your .NET Core application using the appsettings.json file.

json

Copy code

{

"Kafka": {

"BootstrapServers": "localhost:9092",

"GroupId": "event\_store\_group"

}

}

**Implementing the Transactional Outbox Pattern**

1. **Create the Outbox Table:** In your database, create an Outbox table to store messages that need to be sent to Kafka.

sql

Copy code

CREATE TABLE Outbox (

Id INT PRIMARY KEY IDENTITY,

AggregateId UNIQUEIDENTIFIER,

Payload NVARCHAR(MAX),

CreatedAt DATETIME2

);

1. **Transactional Outbox Implementation:** In your application, whenever a business transaction occurs, insert a message into the Outbox table within the same transaction.

csharp

Copy code

using (var transaction = await \_dbContext.Database.BeginTransactionAsync())

{

// Perform business operations

\_dbContext.Orders.Add(order);

// Add a message to the outbox

var outboxMessage = new OutboxMessage

{

AggregateId = order.Id,

Payload = JsonConvert.SerializeObject(orderEvent),

CreatedAt = DateTime.UtcNow

};

\_dbContext.Outbox.Add(outboxMessage);

await \_dbContext.SaveChangesAsync();

await transaction.CommitAsync();

}

1. **Message Dispatcher Service:** Implement a background service that reads messages from the Outbox table and sends them to Kafka.

csharp

Copy code

public class OutboxDispatcherService : BackgroundService

{

private readonly IServiceProvider \_serviceProvider;

public OutboxDispatcherService(IServiceProvider serviceProvider)

{

\_serviceProvider = serviceProvider;

}

protected override async Task ExecuteAsync(CancellationToken stoppingToken)

{

while (!stoppingToken.IsCancellationRequested)

{

using (var scope = \_serviceProvider.CreateScope())

{

var dbContext = scope.ServiceProvider.GetRequiredService<AppDbContext>();

var messages = await dbContext.Outbox.ToListAsync(stoppingToken);

foreach (var message in messages)

{

// Send to Kafka

await \_kafkaProducer.ProduceAsync("order\_topic", new Message<string, string>

{

Key = message.AggregateId.ToString(),

Value = message.Payload

});

// Remove the message from the outbox

dbContext.Outbox.Remove(message);

await dbContext.SaveChangesAsync(stoppingToken);

}

await Task.Delay(TimeSpan.FromSeconds(5), stoppingToken);

}

}

}

}

**Implementing the Event Store Pattern**

1. **Create the Event Store Table:** In your database, create an EventStore table to keep track of consumed events.

sql

Copy code

CREATE TABLE EventStore (

EventId UNIQUEIDENTIFIER PRIMARY KEY,

EventType NVARCHAR(100),

EventData NVARCHAR(MAX),

ConsumedAt DATETIME2

);

1. **Kafka Consumer Implementation:** Implement a Kafka consumer that checks the EventStore table before processing an event.

csharp

Copy code

public class KafkaConsumerService : BackgroundService

{

private readonly IServiceProvider \_serviceProvider;

public KafkaConsumerService(IServiceProvider serviceProvider)

{

\_serviceProvider = serviceProvider;

}

protected override async Task ExecuteAsync(CancellationToken stoppingToken)

{

var config = new ConsumerConfig

{

GroupId = "event\_store\_group",

BootstrapServers = "localhost:9092",

AutoOffsetReset = AutoOffsetReset.Earliest

};

using var consumer = new ConsumerBuilder<Ignore, string>(config).Build();

consumer.Subscribe("order\_topic");

while (!stoppingToken.IsCancellationRequested)

{

var consumeResult = consumer.Consume(stoppingToken);

using (var scope = \_serviceProvider.CreateScope())

{

var dbContext = scope.ServiceProvider.GetRequiredService<AppDbContext>();

var eventExists = await dbContext.EventStore.AnyAsync(e => e.EventId == Guid.Parse(consumeResult.Message.Key), stoppingToken);

if (eventExists)

{

// Event has already been processed

continue;

}

// Process the event

var orderEvent = JsonConvert.DeserializeObject<OrderEvent>(consumeResult.Message.Value);

// Add event to EventStore

var eventStoreEntry = new EventStore

{

EventId = Guid.Parse(consumeResult.Message.Key),

EventType = orderEvent.GetType().Name,

EventData = consumeResult.Message.Value,

ConsumedAt = DateTime.UtcNow

};

dbContext.EventStore.Add(eventStoreEntry);

await dbContext.SaveChangesAsync(stoppingToken);

}

}

}

}

**Integration Testing**

To ensure the reliability of the system, it's crucial to perform integration tests that simulate different failure scenarios, such as:

* Network failures between the producer and Kafka.
* Database transaction failures.
* Consumer crashes.

**5. Best Practices**

* **Idempotency:** Ensure that operations on the consumer side are idempotent, meaning that processing the same event multiple times has no side effects.
* **Monitoring:** Use Kafka monitoring tools like Prometheus and Grafana to monitor the health and performance of your Kafka cluster.
* **Retry Logic:** Implement retry mechanisms in both the producer and consumer to handle transient failures gracefully.

**6. Conclusion**

Implementing the Transactional Outbox Pattern and Event Store Pattern in a .NET Core application allows for a

**Updated Kafka Consumer Implementation**

After inserting the event into the EventStore table, you can add logic to process the event by polling from the EventStore. This approach ensures that events are processed in an asynchronous and decoupled manner, allowing for better scalability and resilience.

csharp

Copy code

public class KafkaConsumerService : BackgroundService

{

private readonly IServiceProvider \_serviceProvider;

public KafkaConsumerService(IServiceProvider serviceProvider)

{

\_serviceProvider = serviceProvider;

}

protected override async Task ExecuteAsync(CancellationToken stoppingToken)

{

var config = new ConsumerConfig

{

GroupId = "event\_store\_group",

BootstrapServers = "localhost:9092",

AutoOffsetReset = AutoOffsetReset.Earliest

};

using var consumer = new ConsumerBuilder<Ignore, string>(config).Build();

consumer.Subscribe("order\_topic");

while (!stoppingToken.IsCancellationRequested)

{

var consumeResult = consumer.Consume(stoppingToken);

using (var scope = \_serviceProvider.CreateScope())

{

var dbContext = scope.ServiceProvider.GetRequiredService<AppDbContext>();

var eventExists = await dbContext.EventStore.AnyAsync(

e => e.EventId == Guid.Parse(consumeResult.Message.Key),

stoppingToken);

if (eventExists)

{

// Event has already been processed

continue;

}

// Add event to EventStore

var eventStoreEntry = new EventStore

{

EventId = Guid.Parse(consumeResult.Message.Key),

EventType = "OrderCreated",

EventData = consumeResult.Message.Value,

ConsumedAt = DateTime.UtcNow

};

dbContext.EventStore.Add(eventStoreEntry);

await dbContext.SaveChangesAsync(stoppingToken);

// Trigger the polling and processing logic after storing the event

await ProcessEventsFromEventStoreAsync(dbContext, stoppingToken);

}

}

}

private async Task ProcessEventsFromEventStoreAsync(AppDbContext dbContext, CancellationToken cancellationToken)

{

// Poll the EventStore for unprocessed events

var unprocessedEvents = await dbContext.EventStore

.Where(e => e.ConsumedAt != null && e.ProcessedAt == null)

.OrderBy(e => e.ConsumedAt)

.ToListAsync(cancellationToken);

foreach (var eventEntry in unprocessedEvents)

{

try

{

// Example: Deserialize and process the order event

var orderEvent = JsonConvert.DeserializeObject<OrderEvent>(eventEntry.EventData);

// Add your processing logic here

await ProcessOrderEventAsync(orderEvent);

// Mark the event as processed

eventEntry.ProcessedAt = DateTime.UtcNow;

dbContext.EventStore.Update(eventEntry);

await dbContext.SaveChangesAsync(cancellationToken);

}

catch (Exception ex)

{

// Handle processing errors (e.g., log the error, retry, etc.)

Console.WriteLine($"Error processing event {eventEntry.EventId}: {ex.Message}");

}

}

}

private Task ProcessOrderEventAsync(OrderEvent orderEvent)

{

// Implement your business logic to handle the order event

Console.WriteLine($"Processing Order ID: {orderEvent.OrderId}");

// Example: Update order status, notify other services, etc.

return Task.CompletedTask;

}

}

**Key Points in the Updated Implementation:**

1. **Polling from Event Store:** After storing the event in the EventStore, the ProcessEventsFromEventStoreAsync method is called. This method queries the EventStore table for unprocessed events and processes them sequentially.
2. **Event Processing Logic:** The ProcessOrderEventAsync method contains the business logic for processing the order event. This could involve updating the order status, notifying other services, or performing any necessary operations.
3. **Marking Events as Processed:** Once an event is successfully processed, it is marked as processed by updating the ProcessedAt timestamp in the EventStore table.
4. **Error Handling:** Any errors during event processing are caught and logged. You can implement additional error-handling mechanisms, such as retries or dead-letter queues.

**Conclusion**

By adding polling and processing logic after storing events in the Event Store, you ensure that your application can handle events in a resilient and scalable manner. This approach also decouples event processing from message consumption, allowing for better resource management and fault tolerance.