Tutorial: Implementing Kafka Flow for Event-Driven Microservices

Introduction

In modern microservices architectures, **event-driven communication** is essential for building scalable and decoupled systems. **Apache Kafka** is a distributed event streaming platform that enables reliable, real-time data processing. In this tutorial, we'll explore how Kafka is implemented in a booking system to handle **booking confirmation events**. We'll cover:

1. **Kafka Basics**: How Kafka works.
2. **Kafka Configuration**: Setting up producers and consumers.
3. **Implementation in the Booking System**: Code walkthrough for Kafka Flow.

1. Kafka Basics

What is Kafka?

Kafka is a distributed event streaming platform that allows:

- **Producers** to publish events to **topics**.
- **Consumers** to subscribe to topics and process events.
- **Brokers** to store and manage topics.
- **Schema Registry** to enforce data schemas for events.

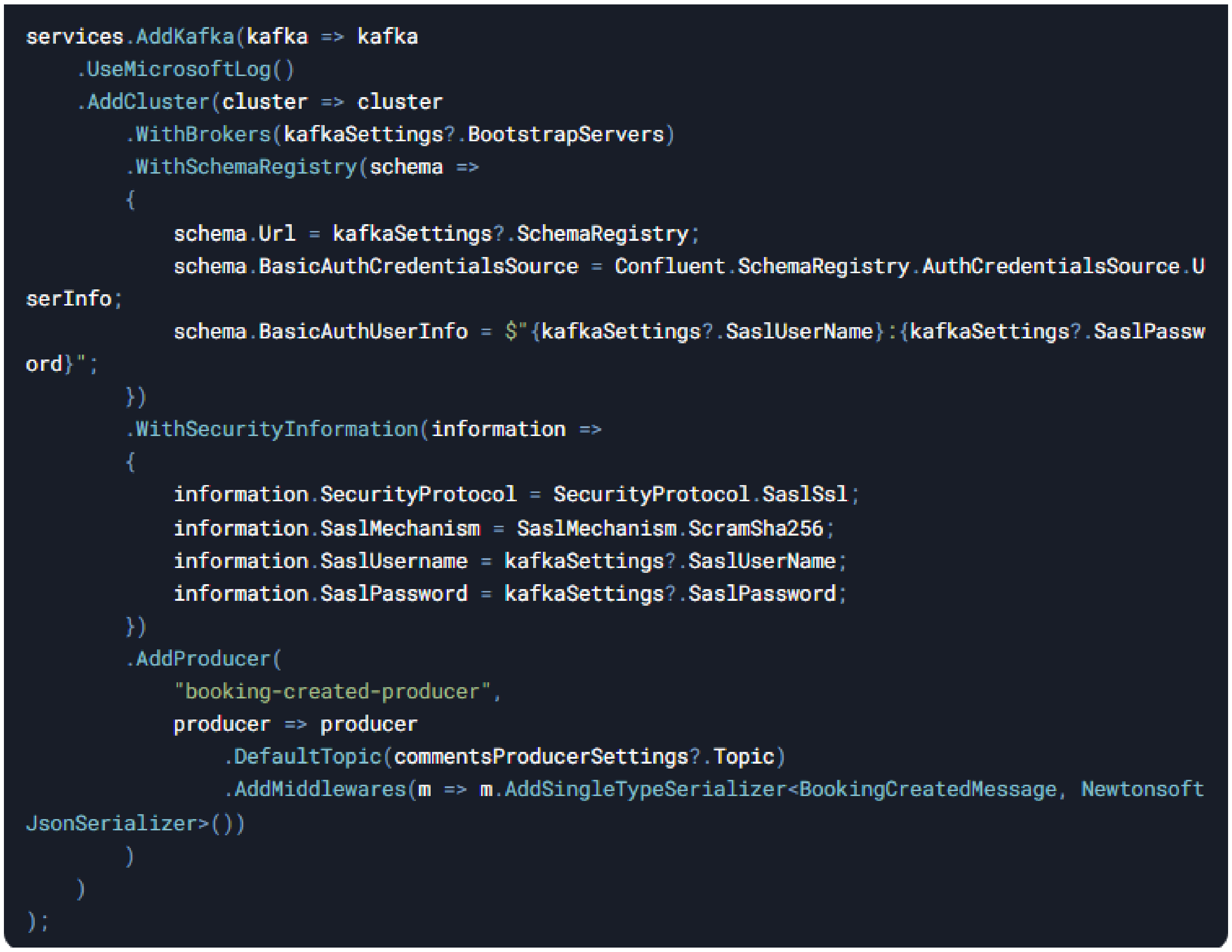
Key Concepts

- **Topic**: A category or feed name to which events are published.
- **Producer**: Publishes events to a topic.
- **Consumer**: Subscribes to a topic and processes events.
- **Broker**: A Kafka server that stores topics and manages event distribution.
- **Schema Registry**: Ensures events conform to a predefined schema.

2. Kafka Configuration

Producer Configuration

The producer is responsible for publishing booking events to a Kafka topic. Here’s how it’s configured in my system:



```
services.AddKafka(kafka => kafka
    .UseMicrosoftLog()
    .AddCluster(cluster => cluster
        .WithBrokers(kafkaSettings?.BootstrapServers)
        .WithSchemaRegistry(schema =>
        {
            schema.Url = kafkaSettings?.SchemaRegistry;
            schema.BasicAuthCredentialsSource = Confluent.SchemaRegistry.AuthCredentialsSource.U
serInfo;
            schema.BasicAuthUserInfo = $"{kafkaSettings?.SaslUserName}:{kafkaSettings?.SaslPassw
ord}";
        })
        .WithSecurityInformation(information =>
        {
            information.SecurityProtocol = SecurityProtocol.SaslSsl;
            information.SaslMechanism = SaslMechanism.ScramSha256;
            information.SaslUsername = kafkaSettings?.SaslUserName;
            information.SaslPassword = kafkaSettings?.SaslPassword;
        })
        .AddProducer(
            "booking-created-producer",
            producer => producer
                .DefaultTopic(commentsProducerSettings?.Topic)
                .AddMiddlewares(m => m.AddSingleTypeSerializer<BookingCreatedMessage, Newtonsoft
JsonSerializer>())
        )
    )
);
```

Explanation

- **Brokers**: Specifies the Kafka broker addresses.
- **Schema Registry**: Ensures events conform to a schema.
- **Security**: Configures SASL/SSL for secure communication.
- **Producer**: Defines a producer for the BookingCreatedMessage event, serialized using NewtonsoftJsonSerializer.

Consumer Configuration

The consumer subscribes to the Kafka topic and processes booking events. Here’s how it’s configured:

```
services.AddKafkaFlowHostedService(kafka => kafka
    .UseMicrosoftLog()
    .AddCluster(cluster => cluster
        .WithBrokers(kafkaSettings?.BootstrapServers)
        .WithSchemaRegistry(schema =>
        {
            schema.Url = kafkaSettings?.SchemaRegistry;
            schema.BasicAuthCredentialsSource = Confluent.SchemaRegistry.AuthCredentialsSource.U
serInfo;
            schema.BasicAuthUserInfo = $"{kafkaSettings?.SaslUserName}:{kafkaSettings?.SaslPassw
ord}";
        })
        .WithSecurityInformation(information =>
        {
            information.SecurityProtocol = SecurityProtocol.SaslSsl;
            information.SaslMechanism = SaslMechanism.ScramSha256;
            information.SaslUsername = kafkaSettings?.SaslUserName;
            information.SaslPassword = kafkaSettings?.SaslPassword;
        })
        .AddConsumer(consumer => consumer
            .Topic(commentsConsumerSettings?.Topic)
            .WithName(commentsConsumerSettings?.WorkerName)
            .WithGroupId(commentsConsumerSettings?.GroupId)
            .WithBufferSize(kafkaSettings!.BufferSize)
            .WithWorkersCount(kafkaSettings!.WorkersCount)
            .WithAutoOffsetReset(AutoOffsetReset.Latest)
            .AddMiddlewares(middlewares => middlewares
                .AddSingleTypeDeserializer<BookingCreatedEvent, NewtonsoftJsonDeserializer>()
                .Add<EventErrorHandlingMiddleware>()
                .AddTypedHandlers(handlers => handlers
                    .AddHandler<BookingCreatedHandler>()
                    .WhenNoHandlerFound(context =>
                        Console.WriteLine($"Messages from partition {context.ConsumerContext.Par
tition} or Offset {context.ConsumerContext.Offset} are unhandled"))
                )
            )
        )
    )
);
```

Explanation

- **Brokers**: Specifies the Kafka broker addresses.
- **Schema Registry**: Ensures events conform to a schema.
- **Security**: Configures SASL/SSL for secure communication.
- **Consumer**: Subscribes to the topic, deserializes events, and processes them using BookingCreatedHandler.

3. Implementation in the Booking System

Event Flow

1. **Booking Creation**:
   - A booking is created via the BookingsController.
   - The BookingCreatedMessage event is published to Kafka.
2. **Event Consumption**:
   - The BookingCreatedHandler processes the event.
   - An email confirmation is sent to the customer.

BookingCreatedHandler

The handler processes the BookingCreatedEvent and sends an email confirmation:

```
public class BookingCreatedHandler : IMessageHandler<BookingCreatedEvent>
{
    private readonly ILogger<BookingCreatedHandler> _logger;
    private readonly IMailService _mailService;

    public BookingCreatedHandler(
        ILogger<BookingCreatedHandler> logger,
        IMailService mailService)
    {
        _logger = logger;
        _mailService = mailService;
    }

    public async Task Handle(IMessageContext context, BookingCreatedEvent message)
    {
        _logger.LogInformation("Processing BookingCreatedEvent: {BookingId}", message.BookingId);

        try
        {
            // Send email confirmation
            await _mailService.SendConfirmationEmailAsync(message.CustomerEmail, message.BookingDetails);
            _logger.LogInformation("Email confirmation sent for booking: {BookingId}", message.BookingId);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed to send email confirmation for booking: {BookingId}", message.BookingId);
        }
    }
}
```

Explanation

- **Logging**: Logs the event processing and any errors.
- **Email Service**: Sends an email confirmation using IMailService.

Conclusion

In this tutorial, we explored how Kafka is implemented in a booking system to handle **booking confirmation events**. We covered:

- **Kafka Basics**: Key concepts and how Kafka works.
- **Kafka Configuration**: Setting up producers and consumers.
- **Implementation**: Code walkthrough for Kafka Flow in the booking system.

By using Kafka, the system achieves **scalability**, **reliability**, and **decoupling** of microservices. The event-driven architecture ensures that booking confirmations are processed in real-time.

Git Repository

The full implementation is available in the public repository:

BookingService – contains Kafka Producer https://github.com/ioni2001/BookingService
BookingService.BookingConfirmation contains Kafka Consumer
https://github.com/ioni2001/BookingService.BookingConfirmation