Spring Boot Microservices with Apache Kafka Example

MICROSERVICES SPRING BOOT SPRING CLOUD

In this tutorial, we will create two Spring Boot microservices that communicate with each other using Apache Kafka. Kafka is a distributed streaming platform that can handle real-time data feeds. This guide is intended for beginners and includes detailed explanations for each step.

Introduction to Apache Kafka

Apache Kafka is an open-source stream-processing software platform developed by LinkedIn and donated to the Apache Software Foundation. Kafka is used for building real-time data pipelines and streaming applications. It is horizontally scalable, fault-tolerant, and highly available.

Prerequisites

- JDK 17 or later
- Maven or Gradle
- IDE (IntelliJ IDEA, Eclipse, etc.)
- Apache Kafka server (You can run Kafka using Docker)

Step 1: Set Up Apache Kafka Server

You can run Kafka using Docker with the following commands:

```
docker network create kafka-net

docker run -d --net=kafka-net --name=zookeeper -e ZOOKEEPER_CLIENT_PORT=2181 confluentinc/cp-zookeeper

docker run -d --net=kafka-net --name=kafka -e KAFKA_ZOOKEEPER_CONNECT=zookeeper:2181 -e KAFKA_ADVERTISED_L
```

This command will start Kafka and Zookeeper in Docker containers.

Step 2: Create the Projects

We'll create two Spring Boot projects: `producer-service` and `consumer-service`.

Step 3: Set Up `producer-service`

3.1 Create the Project

Use Spring Initializr to create a new project with the following dependencies:

- Spring Web
- Spring Boot Actuator

- Spring for Apache Kafka

3.2 Configure `application.properties`

Set up the application properties for `producer-service`.

```
spring.application.name=producer-service
spring.kafka.bootstrap-servers=localhost:9092
```

Explanation:

- `spring.application.name=producer-service`: Names the application.
- `spring.kafka.bootstrap-servers=localhost:9092`: Specifies the Kafka server address.

3.3 Configure Kafka Producer

Create a configuration class to define the Kafka producer settings.

```
package com.example.producerservice;

import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.core.DefaultKafkaProducerFactory;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.core.ProducerFactory;
import org.springframework.kafka.support.serializer.ErrorHandlingDeserializer;
import org.springframework.kafka.support.serializer.JsonSerializer;

import java.util.HashMap;
import java.util.Map;

@Configuration
public class KafkaProducerConfig {

    @Bean
    public ProducerFactory<String, String> producerFactory() {
        Map<String, Object> configProps = new HashMap<>();
        configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        return new DefaultKafkaProducerFactory<>(configProps);
    }

    @Bean
    public KafkaTemplate<String, String> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
```

```
    }
}
```

Explanation:

- `@Configuration`: Indicates that this class contains Spring configuration.
- `@Bean`: Marks this method as a bean producer.
- `ProducerFactory<String, String>`: Configures the Kafka producer factory with server address and serializers.
- `KafkaTemplate<String, String>`: Creates a Kafka template for sending messages.

3.4 Create a Message Producer

Create a service to send messages to Kafka.

```
package com.example.producerservice;

import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;

@Service
public class MessageProducer {

    private static final String TOPIC = "exampleTopic";
    private final KafkaTemplate<String, String> kafkaTemplate;

    public MessageProducer(KafkaTemplate<String, String> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    public void sendMessage(String message) {
        kafkaTemplate.send(TOPIC, message);
    }
}
```

Explanation:

- `@Service`: Marks this class as a service component.
- `KafkaTemplate<String, String> kafkaTemplate`: Injects the `KafkaTemplate` for sending messages to Kafka.
- `sendMessage(String message)`: Sends a message to the `exampleTopic`.

3.5 Create a Controller

Create a controller to handle HTTP requests and send messages to Kafka.

```
package com.example.producerservice;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
```

```
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MessageController {

    private final MessageProducer messageProducer;

    public MessageController(MessageProducer messageProducer) {
        this.messageProducer = messageProducer;
    }

    @GetMapping("/send")
    public String sendMessage(@RequestParam String message) {
        messageProducer.sendMessage(message);
        return "Message sent: " + message;
    }
}
```

Explanation:

- `@RestController`: Marks this class as a REST controller.
- `@GetMapping("/send")`: Maps GET requests to `/send` to this method.
- `@RequestParam String message`: Extracts the `message` parameter from the request.
- `messageProducer.sendMessage(message)`: Sends the message to Kafka.

Step 4: Set Up `consumer-service`

4.1 Create the Project

Use Spring Initializr to create a new project with the following dependencies:

- Spring Web
- Spring Boot Actuator
- Spring for Apache Kafka

4.2 Configure `application.properties`

Set up the application properties for `consumer-service`.

```
spring.application.name=consumer-service
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.consumer.group-id=group_id
```

Explanation:

- `spring.application.name=consumer-service`: Names the application.
- `spring.kafka.bootstrap-servers=localhost:9092`: Specifies the Kafka server address.
- `spring.kafka.consumer.group-id=group_id`: Specifies the Kafka consumer group ID.

4.3 Configure Kafka Consumer

Create a configuration class to define the Kafka consumer settings.

```
package com.example.consumerservice;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.annotation.EnableKafka;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.listener.ConcurrentMessageListenerContainer;

import java.util.HashMap;
import java.util.Map;

@EnableKafka
@Configuration
public class KafkaConsumerConfig {

    @Bean
    public ConsumerFactory<String, String> consumerFactory() {
        Map<String, Object> configProps = new HashMap<>();
        configProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        configProps.put(ConsumerConfig.GROUP_ID_CONFIG, "group_id");
        configProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        configProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        return new DefaultKafkaConsumerFactory<>(configProps);
    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, String> kafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory = new ConcurrentKafkaListenerConta
        factory.setConsumerFactory(consumerFactory());
        return factory;
    }
}
```

Explanation:

- `@EnableKafka`: Enables Kafka listener annotations.
- `@Configuration`: Indicates that this class contains Spring configuration.
- `@Bean`: Marks this method as a bean producer.
- `ConsumerFactory<String, String>`: Configures the Kafka consumer factory with server address and deserializers.

- `ConcurrentKafkaListenerContainerFactory<String, String>`: Creates a Kafka listener container factory.

4.4 Create a Message Listener

Create a service to listen to messages from Kafka.

```
package com.example.consumerservice;

import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

@Service
public class MessageListener {

    @KafkaListener(topics = "exampleTopic", groupId = "group_id")
    public void listen(String message) {
        System.out.println("Received message: " + message);
    }
}
```

Explanation:

- `@Service`: Marks this class as a service component.
- `@KafkaListener(topics = "exampleTopic", groupId = "group_id")`: Annotates a method to listen to messages from the `exampleTopic`.
- `listen(String message)`: Processes the received message and prints it to the console.

Step 5: Run the Microservices

1. **Start Kafka and Zookeeper**: Ensure Kafka and Zookeeper are running using the Docker commands mentioned above.
2. **Start `producer-service`**: Run the `ProducerServiceApplication` class.
3. **Start `consumer-service`**: Run the `ConsumerServiceApplication` class.

Step 6: Test the Communication

1. Open your browser or use a tool like Postman to send a GET request to `producer-service`:
    - URL: `http://localhost:8081/send?message=Hello`
    - This will send the message "Hello" to the `exampleTopic`.
2. Check the console logs of `consumer-service` to see the received message:
    - You should see **Received message: Hello** in the logs.

Conclusion

You have successfully set up two Spring Boot microservices that communicate asynchronously using Apache Kafka. This setup allows you to decouple the services and enable asynchronous processing, which can improve the scalability and resilience of your system. This example can be expanded to include more complex message handling, additional microservices, and advanced Kafka configurations.