

Kafka with Spring

O Apache Kafka é um sistema de processamento de fluxo distribuído e tolerante a falhas.

O **Spring Kafka** traz o modelo de programação de modelo Spring simples e típico com um *KafkaTemplate* e **POJOs** orientados a mensagens por meio da anotação **@KafkaListener**.

Tópicos de configuração

Anteriormente, executávamos ferramentas de linha de comando para criar tópicos no Kafka:

```
$ bin/kafka-topics.sh --create \  
  --zookeeper localhost:2181 \  
  --replication-factor 1 --partitions 1 \  
  --topic mytopic
```

Mas com a introdução do *AdminClient* no Kafka, agora podemos criar tópicos programaticamente.

Precisamos adicionar o bean *KafkaAdmin* Spring, que adicionará tópicos automaticamente para todos os beans do tipo *NewTopic*:

```
@Configuration  
public class KafkaTopicConfig {  
  
    @Value(value = "${spring.kafka.bootstrap-servers}")  
    private String bootstrapAddress;  
  
    @Bean  
    public KafkaAdmin kafkaAdmin() {  
        Map<String, Object> configs = new HashMap<>();  
        configs.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapAddress);  
        return new KafkaAdmin(configs);  
    }  
  
    @Bean  
    public NewTopic topic1() {  
        return new NewTopic("baeldung", 1, (short) 1);  
    }  
}
```

Produção de Mensagens

Para criar mensagens, primeiro precisamos configurar uma *ProducerFactory*. Isso define a estratégia para criar instâncias do Kafka *Producer*.

Em seguida, precisamos de um *KafkaTemplate*, que envolve uma instância do *Producer* e fornece métodos convenientes para enviar mensagens para tópicos Kafka.

As instâncias do produtor são thread-safe. Portanto, usar uma única instância em um contexto de aplicativo proporcionará maior desempenho. Consequentemente, as instâncias *KafkaTemplate* também são thread-safe e o uso de uma instância é recomendado.

Configuração do Produtor

```
@Configuration
public class KafkaProducerConfig {

    @Bean
    public ProducerFactory<String, String> producerFactory() {
        Map<String, Object> configProps = new HashMap<>();
        configProps.put(
            ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            bootstrapAddress);
        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());
    }
}
```

Publicando mensagens

Podemos enviar mensagens usando a classe *KafkaTemplate*:

```
@Autowired
private KafkaTemplate<String, String> kafkaTemplate;

public void sendMessage(String msg) {
    kafkaTemplate.send(topicName, msg);
}
```

A **API de envio** retorna um objeto **CompletableFuture**. Se quisermos bloquear o thread de envio e obter o resultado sobre a mensagem enviada, podemos chamar a API `get` do objeto **CompletableFuture**. O thread aguardará o resultado, mas desacelerará o produtor.

Kafka é uma plataforma de processamento de fluxo rápido. Portanto, é melhor manipular os resultados de forma assíncrona para que as mensagens subsequentes não esperem pelo resultado da mensagem anterior.

Podemos fazer isso através de um callback:

```
public void sendMessage(String message) {
    CompletableFuture<SendResult<String, String>> future = kafkaTemplate.send(topicName, message);
    future.whenComplete((result, ex) -> {
        if (ex == null) {
            System.out.println("Sent message=[" + message +
                "] with offset=[" + result.getRecordMetadata().offset() + "]");
        } else {
            System.out.println("Unable to send message=[" +
                message + "] due to : " + ex.getMessage());
        }
    });
}
```

Consumindo Mensagens

Configuração do consumidor

Para consumir mensagens, precisamos configurar uma *ConsumerFactory* (que define a estratégia de criação de uma instância do Kafka Consumer) e uma *KafkaListenerContainerFactory*. Depois que esses beans estiverem disponíveis na fábrica de bean Spring, os consumidores baseados em POJO podem ser configurados usando a anotação *@KafkaListener*.

KafkaListenerContainerFactory: Em resumo, o `KafkaListenerContainerFactory` é uma interface do Spring que permite criar e configurar contêineres de ouvintes de Kafka, fornecendo uma abstração para personalização e criação de contêineres de ouvintes específicos. Ele desempenha um papel importante na configuração e gerenciamento dos consumidores de mensagens do Apache Kafka em uma aplicação Spring.

A anotação @EnableKafka é necessária na classe de configuração para habilitar a detecção da anotação @KafkaListener em beans gerenciados pelo Spring:

```
@EnableKafka
@Configuration
public class KafkaConsumerConfig {

    @Bean
    public ConsumerFactory<String, String> consumerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(
            ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
            bootstrapAddress);
        props.put(
            ConsumerConfig.GROUP_ID_CONFIG,
            groupId);
        props.put(
            ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class);
        props.put(
            ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class);
        return new DefaultKafkaConsumerFactory<>(props);
    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, String>
        kafkaListenerContainerFactory() {

        ConcurrentKafkaListenerContainerFactory<String, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory());
        return factory;
    }
}
```

Consumindo mensagens

```
@KafkaListener(topics = "topicName", groupId = "foo")
public void listenGroupFoo(String message) {
    System.out.println("Received Message in group foo: " + message);
}
```

Podemos implementar vários ouvintes para um tópico , cada um com um ID de grupo diferente. Além disso, um consumidor pode ouvir mensagens de vários tópicos:

```
@KafkaListener(topics = "topic1, topic2", groupId = "foo")
```

O Spring também suporta a recuperação de um ou mais cabeçalhos de mensagem usando a anotação `@Header` no ouvinte:

```
@KafkaListener(topics = "topicName")
public void listenWithHeaders(
    @Payload String message,
    @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition) {
    System.out.println(
        "Received Message: " + message
        + "from partition: " + partition);
}
```

Consumindo mensagens de uma partição específica

Observe que criamos o tópico *baeldung* com apenas uma partição.

No entanto, para um tópico com várias partições, um `@KafkaListener` pode se inscrever explicitamente em uma partição específica de um tópico com um deslocamento inicial:

```
@KafkaListener(
    topicPartitions = @TopicPartition(topic = "topicName",
    partitionOffsets = {
        @PartitionOffset(partition = "0", initialOffset = "0"),
        @PartitionOffset(partition = "3", initialOffset = "0")}),
    containerFactory = "partitionsKafkaListenerContainerFactory")
public void listenToPartition(
    @Payload String message,
    @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition) {
    System.out.println(
        "Received Message: " + message
        + "from partition: " + partition);
}
```

Como o *initialOffset* foi definido como 0 neste ouvinte, todas as mensagens consumidas anteriormente das partições 0 e 3 serão consumidas novamente toda vez que esse ouvinte for inicializado.

Se não precisarmos definir o deslocamento, podemos usar a propriedade *partitions* da anotação `@TopicPartition` para definir apenas as partições sem o deslocamento:

```
@KafkaListener(topicPartitions
    = @TopicPartition(topic = "topicName", partitions = { "0", "1" })))
```

Adicionando filtro de mensagens para ouvintes

Podemos configurar os ouvintes para consumir conteúdo de mensagem específico adicionando um filtro personalizado. Isso pode ser feito definindo um *RecordFilterStrategy* para *KafkaListenerContainerFactory*:

```
@Bean
public ConcurrentKafkaListenerContainerFactory<String, String>
    filterKafkaListenerContainerFactory() {

    ConcurrentKafkaListenerContainerFactory<String, String> factory =
        new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory());
    factory.setRecordFilterStrategy(
        record -> record.value().contains("World"));
    return factory;
}
```

Podemos então configurar um ouvinte para usar esta fábrica de contêineres:

```
@KafkaListener(
    topics = "topicName",
    containerFactory = "filterKafkaListenerContainerFactory")
public void listenWithFilter(String message) {
    System.out.println("Received Message in filtered listener: " + message);
}
```

Neste listener, todas as **mensagens que corresponderem ao filtro serão descartadas**.

Conversores de mensagens personalizadas

Até agora, cobrimos apenas o envio e recebimento de Strings como mensagens. No entanto, também podemos enviar e receber objetos Java personalizados. Isso requer a configuração do serializador apropriado em **ProducerFactory** e um desserializador em **ConsumerFactory**.

Vejamos uma classe de bean simples , que enviaremos como mensagens:

```
public class Greeting {

    private String msg;
    private String name;

    // standard getters, setters and constructor
}
```

Produzindo mensagens personalizadas

Neste exemplo, usaremos o **JsonSerializer**.

Vejam os códigos para **ProducerFactory** e **KafkaTemplate**:

```
@Bean
public ProducerFactory<String, Greeting> greetingProducerFactory() {
    // ...
    configProps.put(
        ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
        JsonSerializer.class);
    return new DefaultKafkaProducerFactory<>(configProps);
}

@Bean
public KafkaTemplate<String, Greeting> greetingKafkaTemplate() {
    return new KafkaTemplate<>(greetingProducerFactory());
}
```

Podemos usar este novo **KafkaTemplate** para enviar a mensagem de saudação:

```
kafkaTemplate.send(topicName, new Greeting("Hello", "World"));
```

Consumindo mensagens personalizadas

Da mesma forma, vamos modificar **ConsumerFactory** e **KafkaListenerContainerFactory** para desserializar a mensagem de saudação corretamente:

```
@Bean
public ConsumerFactory<String, Greeting> greetingConsumerFactory() {
    // ...
    return new DefaultKafkaConsumerFactory<>(
        props,
        new StringDeserializer(),
        new JsonDeserializer<>(Greeting.class));
}

@Bean
public ConcurrentKafkaListenerContainerFactory<String, Greeting>
greetingKafkaListenerContainerFactory() {

    ConcurrentKafkaListenerContainerFactory<String, Greeting> factory =
        new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(greetingConsumerFactory());
    return factory;
}
```

O serializador e desserializador JSON spring-kafka usam a biblioteca Jackson, que também é uma dependência Maven opcional para o projeto spring-kafka.

Então, vamos adicioná-lo ao nosso *pom.xml*:

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.9.7</version>
</dependency>
```

Em vez de usar a versão mais recente do Jackson, é recomendável usar a versão adicionada ao *pom.xml* do spring-kafka.

Por fim, precisamos escrever um ouvinte para consumir mensagens *de saudação (greeting)*:

```
@KafkaListener(
  topics = "topicName",
  containerFactory = "greetingKafkaListenerContainerFactory")
public void greetingListener(Greeting greeting) {
  // process greeting message
}
```