Lund&Bendsen
developing developers

Kafka Spring

Kafka & Spring

Kafka ships with low-level Kafka consumers and producers optimized for the control plane protocol that Kafka components use. Spring provides Kafka abstractions classes on top of the low-level API. This allows you to integrate with Kafka in a ‘Spring’ like fashion using CDI, annotations and an opinionated config approach.

A few perquisites must be in place before Kafka can be used in Spring.

The following dependency must be in place if you run maven:

```
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
</dependency>
```

In the `application.properties` you can specify some default consumer and producer properties.

Often you just need to specify a (comma-separated list) of brokers:

```
spring.kafka.consumer.bootstrap-servers=localhost:29092
spring.kafka.consumer.group-id=demo-group
spring.kafka.consumer.auto-offset-reset=earliest
```

Exercise

- Introduce spring project structure
- Copy + rename from template kafka-demo project
- Open IntelliJ and import projects
- How to run programs
- Solutions folder

7.1

Kafka low-level style

- Kafka low-level style
- Kafka Spring style
- Kafka admin client
- Testing Kafka

Low-level producer

```
// Define properties used by the consumer
Map<String, Object> props = new HashMap<>();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServer);
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);

try (KafkaProducer<String, String> producer = new KafkaProducer<>(props)) {
  // Create a ProducerRecord – with the possibility to also specify partition
  ProducerRecord<String, String> producerRecord = ProducerRecord<>(”demo-topic", ”Hello world");

  // Send and flush the buffer
  producer.send(producerRecord);
  producer.flush();
}
```

Exercise

7.2

Create vanilla java producer

- Create a new topic and assign 2 partitions
- Write a java producer that produces a single message to the above kafka topic
- Create a console consumer that listens to the above topic. Use `--from-beginning --property print.partition=true --property print.offset=true`
- Verify that the message is consumed.
- Change to producer to write to partition 0 only (use ProducerRecord constructor)
- Verify that the messages get consumed in the console consumer and that the partitions are all from partition 0

Low-level consumer

```
// Define properties used by the consumer
Map<String, Object> props = new HashMap<>();
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServer);
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
props.put(ConsumerConfig.GROUP_ID_CONFIG, groupId); // Consumer group

try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props)){
  consumer.subscribe(Arrays.asList("topic")); // Subscribe to one or more topics

  while (true) { // Endless loop with polling
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records) {
      … Do something
    }
  }
}
```

Exercise

Create vanilla java consumer

- Create a new topic with 2 partitions
- Write a java consumer that consumes from the above topic. Output the consumed messages
- Consider using `props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");`
- Create a consoles producer that allows you to specify key and value. Use `--property parse.key=true --property key.separator=":”`
- Write some key/values to the topic from the console producer
- Verify that the messages are consumed in the java consumer
- Change the program to also output the key, topic-name, partition and offset and rerun it

7.3

Low-level consumer – Manual Ack

Lund&Bendsen
developing developers

```
Map<String, Object> props = new HashMap<>();
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServer);
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
props.put(ConsumerConfig.GROUP_ID_CONFIG, groupId); // Consumer group
properties.put("enable.auto.commit", "false");
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(properties);
consumer.subscribe(Collections.singletonList("topic"));
  try {
    while (true) {
      ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
      for (ConsumerRecord<String, String> record : records) {
        // process records
        consumer.commitSync(Collections.singletonMap(new TopicPartition(record.topic(), record.partition()),
          new OffsetAndMetadata(record.offset() + 1)));
       }
      }
    } finally {
      consumer.close();
    }
  }
}
```

Manual commit of offsets

Next offset to read from

Exercise

7.4

Create vanilla java consumer with manual acknowledgement

- Open a new PowerShell and go to <project-folder>/docker
- Type **./kafka-producer-perf-test --topic demo-topic --num-records 1000 --record-size 2 --throughput 2**
- Start your consumer from the previous exercise
- What happens to the group offsets? – how often do they update?
- Change your consumer to do manual commits
- What happens to the group offsets? – how often do they update?

Kafka Spring style

- Kafka low-level style
- Kafka Spring style
- Kafka admin client
- Testing Kafka

High-level producer

```
@Service
public class KafkaProducerService {

    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    public void sendSyncMessage(String topic, String message) {
        SendResult<String, String> result = kafkaTemplate.send(topic, message).get();
    }

    public void sendAsyncMessageWithCallback(String topic, String message) {
        CompleteableFuture<SendResult<String, String>> future = kafkaTemplate.send(topic, message);
        future.thenAccept(sendresult -> {...})
}
}
```

Autowire the template previously initialized

Block the thread while waiting for ACT

Better. No hanging threads

Common Kafka producer properties

Lund&Bendsen
developing developers

spring.kafka.producer.bootstrap-servers: A comma-separated list of broker addresses. For example, "localhost:9092".

spring.kafka.producer.client-id: An ID for the Kafka producer client.

spring.kafka.producer.key-serializer: The serializer class for the message keys. For example, "org.apache.kafka.common.serialization.StringSerializer".

spring.kafka.producer.value-serializer: The serializer class for the message values. For example, "org.apache.kafka.common.serialization.StringSerializer".

spring.kafka.producer.acks: The number of acknowledgments the producer requires the leader to have received before considering a write complete. This can be set to "0", "1", or "all".

spring.kafka.producer.retries: The number of times the producer will retry sending a message on failure.

spring.kafka.producer.batch-size: The size of a batch of messages before they are sent to Kafka.

spring.kafka.producer.buffer-memory: The total memory that the producer can use to buffer records waiting to be sent to the server.

spring.kafka.producer.linger-ms: The producer will wait for this amount of time (in milliseconds) before sending a batch of records.

spring.kafka.producer.max-request-size: The maximum size of a request in bytes.

spring.kafka.producer.request-timeout-ms: The maximum amount of time the producer will wait for the acknowledgment of a request.

spring.kafka.producer.timeout: The maximum amount of time the producer will block during a send call.

spring.kafka.producer.ssl.*: Various SSL configuration properties for secure communication with the Kafka brokers.

spring.kafka.producer.properties.*: Custom producer properties that you want to set.

Exercise

7.5

Create spring producer

- Create a new topic and assign 2 partitions
- Write a spring producer that writes a single message to the above kafka topic in a non-blocking way. Output the partition and offset that the record was given
- Create a console consumer that listens to the above topic. Use `--from-beginning --property print.partition=true --property print.offset=true`
- Verify that the message is consumed.

High-level consumers

```
@Component
@EnableKafka
class KafkaListenersExample {

  @KafkaListener(topics = “input-topic-1")
  void asString(String data) {
  }
  @KafkaListener(topics = “input-topic-2")
  void asRecord(ConsumerRecord<?,?> record) {
     // From record object you have access to metadata (key, value, topic, partition & offset)
  }

  @KafkaListener(topics = “input-topic-3“, groupId=“my-group”)
  void inGroup(ConsumerRecord<?,?> record) {
  }

  @KafkaListener(topics = “input-topic-4“, batch=“true”)
  void asBatch(List<ConsumerRecord<?,?>> record) {
  }
```

Instruct spring to scan for @KafkaListener annotations

Stringify record value

Pass record

Specify group.id

Process batches

High-level consumer - Acknowledge

Lund&Bendsen
developing developers

```
@KafkaListener(topics = “input-topic-3“, containerFactory=“myContainerFactory”)
void withAck(ConsumerRecord<?,?> record, Acknowledgement ack) {
  log.info("Try to process message");
  try {
    //Some code
    log.info("Processed value: " + record.value());
    ack.acknowledge();
  } catch (SocketTimeoutException e) {
    log.error("Error while processing message. Try again later");
    ack.nack(Duration.ofSeconds(5));
  } catch (Exception e) {
    log.error("Error while processing message: {} [illegible]()));
    ack.acknowledge();
  }
}
```

Specify factory if more fine-grained control is needed

Allows for explicit ack. Spring supports a number of different ACT settings (batch, record, time etc)

Negative ack. Sleep & reseek partitions

Acknowledge

High-level consumer - Factory

```
@Bean
public ConcurrentKafkaListenerContainerFactory <?> myContainerFactory() {
  Map<String, Object> props = new HashMap<>();
  props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
  props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
  props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
  props.put(ConsumerConfig.GROUP_ID_CONFIG, ”my-group");

  var factory = new ConcurrentKafkaListenerContainerFactory<>();
  factory.setConsumerFactory(new DefaultKafkaConsumerFactory(props));
  factory.setBatchListener(true);
  factory.setConcurrency(3);
  return factory;
}
```

Place to specify group.id

Batch style needs to be specified in factory

Process partitions concurrently

Common Kafka consumer properties

spring.kafka.consumer.bootstrap-servers: Comma-separated list of Kafka broker addresses.

spring.kafka.consumer.group-id: The consumer group ID. It identifies a group of consumers that work together to consume Kafka messages.

spring.kafka.consumer.auto-offset-reset: Determines where a new consumer group should start consuming messages. Common values are "earliest" (from the beginning) or "latest" (from the latest).

spring.kafka.consumer.key-deserializer: Deserializer for the message keys.

spring.kafka.consumer.value-deserializer: Deserializer for the message values.

spring.kafka.consumer.max-poll-records: Maximum number of records to poll in each poll().

spring.kafka.consumer.fetch-min-size: Minimum amount of data the server should return for a fetch request.

spring.kafka.consumer.fetch-max-wait: Maximum amount of time the server should wait for more data to arrive.

spring.kafka.consumer.max-poll-interval-ms: Maximum time between polls.

spring.kafka.consumer.enable-auto-commit: If set to true, the consumer will automatically commit offsets.

spring.kafka.consumer.auto-commit-interval: The frequency at which the consumer's offsets are automatically committed.

spring.kafka.consumer.properties: Additional Kafka consumer properties. You can use this property to set any custom Kafka consumer configuration.

spring.kafka.listener.concurrency: The number of threads to use for message consumption.

spring.kafka.listener.poll-timeout: The timeout for each poll() operation.

spring.kafka.consumer.auto-commit-interval: The frequency at which offsets are committed when enable-auto-commit is set to true.

Exercise

7.6

- Create spring consumer
  - Create a new topic with 2 partitions
  - Write a spring consumer that consumes from the above topic. Output the consumed messages
  - Open a new PowerShell and go to <project-folder>/docker
  - type **./kafka-producer-perf-test --topic demo-topic --num-records 10000 --record-size 2 --throughput 2**
  - Verify that the messages are consumed in the java consumer
  - Change the program to do batch consumption and output the size of the batches.
  - Rerun the producer with this setting **./kafka-producer-perf-test --topic demo-topic --num-records 100000 --record-size 2 --throughput 10000**

Kafka admin client

- Kafka low-level style
- Kafka spring style
- Kafka admin client
- Testing Kafka

Kafka admin

The Kafka client ships with an admin client that allows you to run administrative tasks against your Kafka cluster. Much the same way as CLI operations but just in Java instead.

Once initialized you can do CRUD operations on all Kafka entities like list, create, update and delete topics, consumer-groups etc. The Admin methods often return a `KafkaFuture` object to allow for async calls. To wait for the `KafkaFuture` to complete call the get method on the future:

```
Properties properties = new Properties();
properties.put("bootstrap.servers", "localhost:29092");
properties.put("client.id", "admin-client");

AdminClient client = AdminClient.create(properties);
ListTopicsResult topics = client.listTopics().names().get()
```

Kafka admin – topic operations

`createTopics()`

This method is used to create topics in a Kafka cluster. You can specify the topic name, the number of partitions, replication factor, and other topic configuration settings.

`deleteTopics()`

It allows you to delete topics in the Kafka cluster. You need to specify the names of the topics you want to delete.

`listTopics()`

This method retrieves a list of all the topics in the Kafka cluster.

`describeTopics()`

You can use this method to get detailed information about one or more topics. It provides information about the topic's partitions, replication factors, and configuration.

Exercise

Create new topic using admin client

- Initialize the AdminClient using the example-code in the slides
- Create `NewTopic` object using `NewTopic("topicName",…)` constructor
- Use `client.createTopics(topic)` method to create the kafka topic
- Verify that the topic has been created

7.7

Exercise

7.8

- List all topics using admin client
  - Initialize the AdminClient using the example-code in the slides
  - Use `client.listTopics()` method to list all topics
  - Verify that the listed topics are correct
  - What other topic properties are available besides the name of the topic?

Overview

Lund&Bendsen
developing developers

- Kafka java style
- Kafka spring style
- Kafka admin client
- Testing Kafka

Embedded Kafka

```
@SpringBootTest
@DirtiesContext
@EmbeddedKafka(partitions = 1, brokerProperties = { "listeners=PLAINTEXT://localhost:9092", "port=9092" })
public class EmbeddedKafkaTest {


   @Autowired
   private KafkaConsumer consumer;
   @Autowired
   private KafkaProducer producer;


   @Test
   public void sendAndTestIfMessageIsReceived() throws Exception {
      String data = ”test-message";
      producer.send(”test-topic", data);
      boolean messageConsumed = consumer.getLatch().await(10, TimeUnit.SECONDS);
      assertTrue(messageConsumed);
   }
}
```

Ensure that our test bootstraps the Spring application context

Context is cleaned and reset between different tests
& Instantiate Embedded Kafka broker

TestContainers

```java
@RunWith(SpringRunner.class)
@SpringBootTest()
@DirtiesContext
public class TestContainersTest {
    @ClassRule
    public static KafkaContainer kafka = new KafkaContainer(DockerImageName.parse("confluentinc/cp-kafka:latest"));
    @Autowired
    private KafkaConsumer consumer;
    @Autowired
    private KafkaProducer producer;

    @Test
    public void sendAndTestIfMessageIsReceived() throws Exception {
        String data = ”test-message";
        producer.send(”test-topic", data);
        boolean messageConsumed = consumer.getLatch().await(10, TimeUnit.SECONDS);
        assertTrue(messageConsumed);
    }
}
```

Run Kafka in a container

Consumer wrapper

```
@Component
public class KafkaConsumer {
   private static final Logger LOGGER = LoggerFactory.getLogger(KafkaConsumer.class);
   private CountDownLatch latch = new CountDownLatch(1);

   @KafkaListener(topics = "${test.topic}")
   public void receive(ConsumerRecord<?, ?> consumerRecord) {
      LOGGER.info("received payload='{}'", consumerRecord.toString());
      latch.countDown();
   }

   public void getLatch() {
      return latch;
   }
   public void resetLatch() {
      latch = new CountDownLatch(1);
   }
}
```