https://reflectoring.io/spring-boot-kafka/

In this article, we’ll look at how to integrate a Spring Boot application with Apache Kafka and start sending and consuming messages from our application. We’ll be going through each section with code examples.

**Code Example**

This article is accompanied by a working code example [on GitHub](https://github.com/thombergs/code-examples/tree/master/spring-boot/spring-boot-kafka).

**Why Kafka?**

Traditional messaging queues like ActiveMQ, RabbitMQ can handle high throughput usually used for long-running or background jobs and communicating between services.

Kafka is a stream-processing platform built by LinkedIn and currently developed under the umbrella of the Apache Software Foundation. Kafka aims to provide low-latency ingestion of large amounts of event data.

**We can use Kafka when we have to move a large amount of data and process it in real-time**. An example would be when we want to process user behavior on our website to generate product suggestions or monitor events produced by our micro-services.

Kafka is built from ground up with horizontal scaling in mind. We can scale by adding more brokers to the existing Kafka cluster.

**Kafka Vocabulary**

Let’s look at the key terminologies of Kafka:

* **Producer**: A producer is a client that sends messages to the Kafka server to the specified topic.
* **Consumer**: Consumers are the recipients who receive messages from the Kafka server.
* **Broker**: Brokers can create a Kafka cluster by sharing information using Zookeeper. A broker receives messages from producers and consumers fetch messages from the broker by topic, partition, and offset.
* **Cluster**: Kafka is a distributed system. A Kafka cluster contains multiple brokers sharing the workload.
* **Topic**: A topic is a category name to which messages are published and from which consumers can receive messages.
* **Partition**: Messages published to a topic are spread across a Kafka cluster into several partitions. Each partition can be associated with a broker to allow consumers to read from a topic in parallel.
* **Offset**: Offset is a pointer to the last message that Kafka has already sent to a consumer.

**Configuring a Kafka Client**

We should have a Kafka server running on our machine. If you don’t have Kafka setup on your system, take a look at the [Kafka quickstart guide](https://kafka.apache.org/quickstart). Once we have a Kafka server up and running, a Kafka client can be easily configured with Spring configuration in Java or even quicker with Spring Boot.

Let’s start by adding spring-kafka dependency to our pom.xml:

<dependency>

<groupId>org.springframework.kafka</groupId>

<artifactId>spring-kafka</artifactId>

<version>2.5.2.RELEASE</version>

</dependency>

**Using Java Configuration**

Let’s now see how to configure a Kafka client using Spring’s Java Configuration. To split up responsibilities, we have separated KafkaProducerConfig and KafkaConsumerConfig.

Let’s have a look at the producer configuration first:

**@Configuration**

**class** **KafkaProducerConfig** **{**

**@Value(**"${io.reflectoring.kafka.bootstrap-servers}"**)**

**private** **String** bootstrapServers**;**

**@Bean**

**public** **Map<String,** **Object>** **producerConfigs()** **{**

**Map<String,** **Object>** props **=** **new** **HashMap<>();**

props**.**put**(ProducerConfig.**BOOTSTRAP\_SERVERS\_CONFIG**,**

bootstrapServers**);**

props**.**put**(ProducerConfig.**KEY\_SERIALIZER\_CLASS\_CONFIG**,**

**StringSerializer.**class**);**

props**.**put**(ProducerConfig.**VALUE\_SERIALIZER\_CLASS\_CONFIG**,**

**StringSerializer.**class**);**

**return** props**;**

**}**

**@Bean**

**public** **ProducerFactory<String,** **String>** **producerFactory()** **{**

**return** **new** **DefaultKafkaProducerFactory<>(**producerConfigs**());**

**}**

**@Bean**

**public** **KafkaTemplate<String,** **String>** **kafkaTemplate()** **{**

**return** **new** **KafkaTemplate<>(**producerFactory**());**

**}**

**}**

The above example shows how to configure the Kafka producer to send messages. ProducerFactory is responsible for creating Kafka Producer instances.

KafkaTemplate helps us to send messages to their respective topic. We’ll see more about KafkaTemplate in the [sending messages](https://reflectoring.io/spring-boot-kafka/#sending-messages) section.

In producerConfigs() we are configuring a couple of properties:

* BOOTSTRAP\_SERVERS\_CONFIG - Host and port on which Kafka is running.
* KEY\_SERIALIZER\_CLASS\_CONFIG - Serializer class to be used for the key.
* VALUE\_SERIALIZER\_CLASS\_CONFIG - Serializer class to be used for the value. We are using StringSerializer for both keys and values.

Now that our producer config is ready, let’s create a configuration for the consumer:

**@Configuration**

**class** **KafkaConsumerConfig** **{**

**@Value(**"${io.reflectoring.kafka.bootstrap-servers}"**)**

**private** **String** bootstrapServers**;**

**@Bean**

**public** **Map<String,** **Object>** **consumerConfigs()** **{**

**Map<String,** **Object>** props **=** **new** **HashMap<>();**

props**.**put**(ConsumerConfig.**BOOTSTRAP\_SERVERS\_CONFIG**,**

bootstrapServers**);**

props**.**put**(ConsumerConfig.**KEY\_DESERIALIZER\_CLASS\_CONFIG**,**

**StringDeserializer.**class**);**

**return** props**;**

**}**

**@Bean**

**public** **ConsumerFactory<String,** **String>** **consumerFactory()** **{**

**return** **new** **DefaultKafkaConsumerFactory<>(**consumerConfigs**());**

**}**

**@Bean**

**public** **KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<String,** **String>>** **kafkaListenerContainerFactory()** **{**

**ConcurrentKafkaListenerContainerFactory<String,** **String>** factory **=**

**new** **ConcurrentKafkaListenerContainerFactory<>();**

factory**.**setConsumerFactory**(**consumerFactory**());**

**return** factory**;**

**}**

**}**

We use ConcurrentKafkaListenerContainerFactory to create containers for methods annotated with @KafkaListener. The KafkaListenerContainer receives all the messages from all topics or partitions on a single thread. We’ll see more about message listener containers in the [consuming messages](https://reflectoring.io/spring-boot-kafka/#consuming-messages) section.

**Using Spring Boot Auto Configuration**

**Spring Boot does most of the configuration automatically**, so we can focus on building the listeners and producing the messages. It also provides the option to override the default configuration through application.properties. The Kafka configuration is controlled by the configuration properties with the prefix spring.kafka.\*:

spring.kafka.bootstrap-servers=localhost:9092

spring.kafka.consumer.group-id=myGroup

**Creating Kafka Topics**

A topic must exist to start sending messages to it. Let`s now have a look at how we can create Kafka topics:

**@Configuration**

**class** **KafkaTopicConfig** **{**

**@Bean**

**public** **NewTopic** **topic1()** **{**

**return** **TopicBuilder.**name**(**"reflectoring-1"**).**build**();**

**}**

**@Bean**

**public** **NewTopic** **topic2()** **{**

**return** **TopicBuilder.**name**(**"reflectoring-2"**).**build**();**

**}**

**...**

**}**

A KafkaAdmin bean is responsible for creating new topics in our broker. **With Spring Boot, a KafkaAdmin bean is automatically registered.**

For a non Spring Boot application we have to manually register KafkaAdmin bean:

**@Bean**

**KafkaAdmin** **admin()** **{**

**Map<String,** **Object>** configs **=** **new** **HashMap<>();**

configs**.**put**(AdminClientConfig.**BOOTSTRAP\_SERVERS\_CONFIG**,** **...);**

**return** **new** **KafkaAdmin(**configs**);**

**}**

To create a topic, we register a NewTopic bean for each topic to the application context. If the topic already exists, the bean will be ignored. We can make use of TopicBuilder to create these beans. KafkaAdmin also increases the number of partitions if it finds that an existing topic has fewer partitions than NewTopic.numPartitions.

**Sending Messages**

**Using KafkaTemplate**

KafkaTemplate provides convenient methods to send messages to topics:

**@Component**

**class** **KafkaSenderExample** **{**

**private** **KafkaTemplate<String,** **String>** kafkaTemplate**;**

**...**

**@Autowired**

**KafkaSenderExample(KafkaTemplate<String,** **String>** kafkaTemplate**,** **...)** **{**

**this.**kafkaTemplate **=** kafkaTemplate**;**

**...**

**}**

**void** **sendMessage(String** message**,** **String** topicName**)** **{**

kafkaTemplate**.**send**(**topicName**,** message**);**

**}**

**...**

**}**

All we need to do is to call the sendMessage() method with the message and the topic name as parameters.

Spring Kafka also allows us to configure an async callback:

**@Component**

**class** **KafkaSenderExample** **{**

**...**

**void** **sendMessageWithCallback(String** message**)** **{**

**ListenableFuture<SendResult<String,** **String>>** future **=**

kafkaTemplate**.**send**(**topic1**,** message**);**

future**.**addCallback**(new** **ListenableFutureCallback<SendResult<String,** **String>>()** **{**

**@Override**

**public** **void** **onSuccess(SendResult<String,** **String>** result**)** **{**

LOG**.**info**(**"Message [{}] delivered with offset {}"**,**

message**,**

result**.**getRecordMetadata**().**offset**());**

**}**

**@Override**

**public** **void** **onFailure(Throwable** ex**)** **{**

LOG**.**warn**(**"Unable to deliver message [{}]. {}"**,**

message**,**

ex**.**getMessage**());**

**}**

**});**

**}**

**}**

The send() method of KafkaTemplate returns a ListenableFuture<SendResult>. We can register a ListenableFutureCallback with the listener to receive the result of the send and do some work within an execution context.

If we don’t want to work with Futures, we can register a ProducerListener instead:

**@Configuration**

**class** **KafkaProducerConfig** **{**

**@Bean**

**KafkaTemplate<String,** **String>** **kafkaTemplate()** **{**

**KafkaTemplate<String,** **String>** kafkaTemplate **=**

**new** **KafkaTemplate<>(**producerFactory**());**

**...**

kafkaTemplate**.**setProducerListener**(new** **ProducerListener<String,** **String>()** **{**

**@Override**

**public** **void** **onSuccess(**

**ProducerRecord<String,** **String>** producerRecord**,**

**RecordMetadata** recordMetadata**)** **{**

LOG**.**info**(**"ACK from ProducerListener message: {} offset: {}"**,**

producerRecord**.**value**(),**

recordMetadata**.**offset**());**

**}**

**});**

**return** kafkaTemplate**;**

**}**

**}**

We configured KafkaTemplate with a ProducerListener which allows us to implement the onSuccess() and onError() methods.

**Using RoutingKafkaTemplate**

We can use RoutingKafkaTemplate when we have **multiple producers with different configurations** and we want to select producer at runtime based on the topic name.

**@Configuration**

**class** **KafkaProducerConfig** **{**

**...**

**@Bean**

**public** **RoutingKafkaTemplate** **routingTemplate(GenericApplicationContext** context**)** **{**

*// ProducerFactory with Bytes serializer*

**Map<String,** **Object>** props **=** **new** **HashMap<>();**

props**.**put**(ProducerConfig.**BOOTSTRAP\_SERVERS\_CONFIG**,**

bootstrapServers**);**

props**.**put**(ProducerConfig.**KEY\_SERIALIZER\_CLASS\_CONFIG**,**

**StringSerializer.**class**);**

props**.**put**(ProducerConfig.**VALUE\_SERIALIZER\_CLASS\_CONFIG**,**

**ByteArraySerializer.**class**);**

**DefaultKafkaProducerFactory<Object,** **Object>** bytesPF **=**

**new** **DefaultKafkaProducerFactory<>(**props**);**

context**.**registerBean**(DefaultKafkaProducerFactory.**class**,** "bytesPF"**,** bytesPF**);**

*// ProducerFactory with String serializer*

props**.**put**(ProducerConfig.**VALUE\_SERIALIZER\_CLASS\_CONFIG**,**

**StringSerializer.**class**);**

**DefaultKafkaProducerFactory<Object,** **Object>** stringPF **=**

**new** **DefaultKafkaProducerFactory<>(**props**);**

**Map<Pattern,** **ProducerFactory<Object,** **Object>>** map **=** **new** **LinkedHashMap<>();**

map**.**put**(Pattern.**compile**(**".\*-bytes"**),** bytesPF**);**

map**.**put**(Pattern.**compile**(**"reflectoring-.\*"**),** stringPF**);**

**return** **new** **RoutingKafkaTemplate(**map**);**

**}**

**...**

**}**

RoutingKafkaTemplate takes a map of java.util.regex.Pattern and ProducerFactory instances and routes messages to the first ProducerFactory matching a given topic name. If we have two patterns ref.\* and reflectoring-.\*, the pattern reflectoring-.\* should be at the beginning because the ref.\* pattern would “override” it, otherwise.

In the above example, we have created two patterns .\*-bytes and reflectoring-.\*. The topic names ending with ‘-bytes’ and starting with reflectoring-.\* will use ByteArraySerializer and StringSerializer respectively when we use RoutingKafkaTemplate instance.

**Consuming Messages**

**Message Listener**

**A KafkaMessageListenerContainer receives all messages from all topics on a single thread.**

A ConcurrentMessageListenerContainer assigns these messages to multiple KafkaMessageListenerContainer instances to provide multi-threaded capability.

**Using @KafkaListener at Method Level**

The @KafkaListener annotation allows us to create listeners:

**@Component**

**class** **KafkaListenersExample** **{**

**Logger** LOG **=** **LoggerFactory.**getLogger**(KafkaListenersExample.**class**);**

**@KafkaListener(**topics **=** "reflectoring-1"**)**

**void** **listener(String** data**)** **{**

LOG**.**info**(**data**);**

**}**

**@KafkaListener(**

topics **=** "reflectoring-1, reflectoring-2"**,**

groupId **=** "reflectoring-group-2"**)**

**void** **commonListenerForMultipleTopics(String** message**)** **{**

LOG**.**info**(**"MultipleTopicListener - {}"**,** message**);**

**}**

**}**

To use this annotation we should add the @EnableKafka annotation on one of our @Configuration classes. Also, it requires a listener container factory, which we have configured in KafkaConsumerConfig.java.

Using @KafkaListener will make this bean method a listener and wrap the bean in MessagingMessageListenerAdapter. We can also specify multiple topics for a single listener using the topics attribute as shown above.

**Using @KafkaListener at Class Level**

We can also use the @KafkaListener annotation at class level. If we do so, we need to specify @KafkaHandler at the method level:

**@Component**

**@KafkaListener(**id **=** "class-level"**,** topics **=** "reflectoring-3"**)**

**class** **KafkaClassListener** **{**

**...**

**@KafkaHandler**

**void** **listen(String** message**)** **{**

LOG**.**info**(**"KafkaHandler[String] {}"**,** message**);**

**}**

**@KafkaHandler(**isDefault **=** **true)**

**void** **listenDefault(Object** object**)** **{**

LOG**.**info**(**"KafkaHandler[Default] {}"**,** object**);**

**}**

**}**

When the listener receives messages, it converts them into the target types and tries to match that type against the method signatures to find out which method to call.

In the example, messages of type String will be received by listen() and type Object will be received by listenDefault(). Whenever there is no match, the default handler (defined by isDefault=true) will be called.

**Consuming Messages from a Specific Partition with an Initial Offset**

We can configure listeners to listen to multiple topics, partitions, and a specific initial offset.

For example, if we want to receive all the messages sent to a topic from the time of its creation on application startup we can set the initial offset to zero:

**@Component**

**class** **KafkaListenersExample** **{**

**...**

**@KafkaListener(**

groupId **=** "reflectoring-group-3"**,**

topicPartitions **=** **@TopicPartition(**

topic **=** "reflectoring-1"**,**

partitionOffsets **=** **{** **@PartitionOffset(**

partition **=** "0"**,**

initialOffset **=** "0"**)** **}))**

**void** **listenToPartitionWithOffset(**

**@Payload** **String** message**,**

**@Header(KafkaHeaders.**RECEIVED\_PARTITION\_ID**)** **int** partition**,**

**@Header(KafkaHeaders.**OFFSET**)** **int** offset**)** **{**

LOG**.**info**(**"Received message [{}] from partition-{} with offset-{}"**,**

message**,**

partition**,**

offset**);**

**}**

**}**

Since we have specified initialOffset = "0", **we will receive all the messages starting from offset 0 every time we restart the application.**

We can also retrieve some useful metadata about the consumed message using the @Header() annotation.

**Filtering Messages**

Spring provides a strategy to filter messages before they reach our listeners:

**class** **KafkaConsumerConfig** **{**

**@Bean**

**KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<String,** **String>>**

**kafkaListenerContainerFactory()** **{**

**ConcurrentKafkaListenerContainerFactory<String,** **String>** factory **=**

**new** **ConcurrentKafkaListenerContainerFactory<>();**

factory**.**setConsumerFactory**(**consumerFactory**());**

factory**.**setRecordFilterStrategy**(**record **->**

record**.**value**().**contains**(**"ignored"**));**

**return** factory**;**

**}**

**}**

Spring wraps the listener with a FilteringMessageListenerAdapter. It takes an implementation of RecordFilterStrategy in which we implement the filter method. **Messages that match the filter will be discarded before reaching the listener.**

In the above example, we have added a filter to discard the messages which contain the word “ignored”.

**Replying with @SendTo**

Spring allows sending method’s return value to the specified destination with @SendTo:

**@Component**

**class** **KafkaListenersExample** **{**

**...**

**@KafkaListener(**topics **=** "reflectoring-others"**)**

**@SendTo(**"reflectoring-1"**)**

**String** **listenAndReply(String** message**)** **{**

LOG**.**info**(**"ListenAndReply [{}]"**,** message**);**

**return** "This is a reply sent after receiving message"**;**

**}**

**}**

The Spring Boot default configuration gives us a reply template. Since we are overriding the factory configuration above, the listener container factory must be provided with a KafkaTemplate by using setReplyTemplate() which is then used to send the reply.

In the above example, we are sending the reply message to the topic “reflectoring-1”.

**Custom Messages**

Let’s now look at how to send/receive a Java object. We’ll be sending and receiving User objects in our example.

**class** **User** **{**

**private** **String** name**;**

**...**

**}**

**Configuring JSON Serializer & Deserializer**

To achieve this, we must configure our producer and consumer to use a JSON serializer and deserializer:

**@Configuration**

**class** **KafkaProducerConfig** **{**

**...**

**@Bean**

**public** **ProducerFactory<String,** **User>** **userProducerFactory()** **{**

**...**

configProps**.**put**(ProducerConfig.**VALUE\_SERIALIZER\_CLASS\_CONFIG**,**

**JsonSerializer.**class**);**

**return** **new** **DefaultKafkaProducerFactory<>(**configProps**);**

**}**

**@Bean**

**public** **KafkaTemplate<String,** **User>** **userKafkaTemplate()** **{**

**return** **new** **KafkaTemplate<>(**userProducerFactory**());**

**}**

**}**

**@Configuration**

**class** **KafkaConsumerConfig** **{**

**...**

**public** **ConsumerFactory<String,** **User>** **userConsumerFactory()** **{**

**Map<String,** **Object>** props **=** **new** **HashMap<>();**

props**.**put**(ConsumerConfig.**BOOTSTRAP\_SERVERS\_CONFIG**,** bootstrapServers**);**

props**.**put**(ConsumerConfig.**GROUP\_ID\_CONFIG**,** "reflectoring-user"**);**

**return** **new** **DefaultKafkaConsumerFactory<>(**

props**,**

**new** **StringDeserializer(),**

**new** **JsonDeserializer<>(User.**class**));**

**}**

**@Bean**

**public** **ConcurrentKafkaListenerContainerFactory<String,** **User>** **userKafkaListenerContainerFactory()** **{**

**ConcurrentKafkaListenerContainerFactory<String,** **User>** factory **=**

**new** **ConcurrentKafkaListenerContainerFactory<>();**

factory**.**setConsumerFactory**(**userConsumerFactory**());**

**return** factory**;**

**}**

**...**

**}**

Spring Kafka provides JsonSerializer and JsonDeserializer implementations that are based on the Jackson JSON object mapper. It allows us to convert any Java object to bytes[].

In the above example, we are creating one more ConcurrentKafkaListenerContainerFactory for JSON serialization. In this, we have configured JsonSerializer.class as our value serializer in the producer config and JsonDeserializer<>(User.class) as our value deserializer in the consumer config.

For this, we are creating a separate Kafka listener container userKafkaListenerContainerFactory(). If we have multiple Java object types to be serialized/deserialized, we have to create a listener container for each type as shown above.

**Sending Java Objects**

Now that we have configured our serializer and deserializer, we can send a User object using the KafkaTemplate:

**@Component**

**class** **KafkaSenderExample** **{**

**...**

**@Autowired**

**private** **KafkaTemplate<String,** **User>** userKafkaTemplate**;**

**void** **sendCustomMessage(User** user**,** **String** topicName**)** **{**

userKafkaTemplate**.**send**(**topicName**,** user**);**

**}**

**...**

**}**

**Receiving Java Objects**

We can listen to User objects by using the @KafkaListener annotation:

**@Component**

**class** **KafkaListenersExample** **{**

**@KafkaListener(**

topics **=** "reflectoring-user"**,**

groupId**=**"reflectoring-user"**,**

containerFactory**=**"userKafkaListenerContainerFactory"**)**

**void** **listener(User** user**)** **{**

LOG**.**info**(**"CustomUserListener [{}]"**,** user**);**

**}**

**}**

Since we have multiple listener containers, we are specifying which container factory to use.

If we don’t specify the containerFactory attribute it defaults to kafkaListenerContainerFactory which uses StringSerializer and StringDeserializer in our case.

**Conclusion**

In this article, we covered how we can leverage the Spring support for Kafka. Build Kafka based messaging with code examples that can help to get started quickly.

You can play around with the code [on GitHub](https://github.com/thombergs/code-examples/tree/master/spring-boot/spring-boot-kafka).