

Testing an Apache Kafka Integration within a Spring Boot Application

October 12, 2018

Java (https://blog.mimacom.com/tag/java/) Spring (https://blog.mimacom.com/tag/spring/)

by Valentin Zickner (https://blog.mimacom.com/author/valentinzickner/)

Kafka (https://blog.mimacom.com/tag/kafka/) Testing (https://blog.mimacom.com/tag/testing/)

Integrating external services into an application is often challenging. Instead of doing the testing manually, the setup could be tested also automated. In case you are using Spring Boot, for a couple of services there exist an integration. This blog post will show how you can setup your Kafka tests to use an embedded Kafka server.

Project Setup

Either use your existing Spring Boot project or generate a new one on start.spring.io (https://start.spring.io). In addition to the normal Kafka dependencies you need to add the `spring-kafka-test` dependency:


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


Class Configuration

The most basic test is just to test the integration. Therefore you need to use Kafka to publish a message and afterward you could read the message from the topic.

You need to annotate your test class with at least the following annotations:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@EmbeddedKafka
public class SimpleKafkaTest {
   // ...
}
```

Until now, the `@SpringBootTest` annotation is not really necessary. Later you would like to have it to configure Kafka for the actual implementation. It is possible to restrict `@SpringBootTest` to the necessary classes.

Our `@EmbeddedKafka` is now available in our test class. It will be bootstrapped before our first test case of this class is executed and killed after the last test case. There are a couple of properties available to influence the behavior and size of the embedded Kafka node:

- `count`: number of brokers, the default is `1`
- `controlledShutdown`, the default is `false`
- `partitions`, the default is `2`
- `topics` names of the topics to be created at the startup
- `brokerProperties / brokerPropertiesLocation` additional properties for the Kafka broker

As a next step, you can autowire the running embedded Kafka instance. Since Kafka is running on a random port, it's necessary to get the configuration for your producers and consumers:

```
@Autowired
private KafkaEmbedded kafkaEmbedded;
```

Configure Kafka Consumer

Now you are able to configure your consumer or producer:

```
Map<String, Object> configs = new HashMap<>(KafkaTestUtils.consumerProps("consumer", "false", kafkaEmbedded));
Consumer<String, String> consumer = new DefaultKafkaConsumerFactory<>(configs, new StringDeserializer(), new StringDeserializer()).createConsumer();
consumer.subscribe(singleton(TOPIC));
```

`KafkaTestUtils.consumerProps` is providing you almost all the properties you need. The first parameter is the name of your consumer group, the second is a flag to set auto commit and the last parameter is the `KafkaEmbedded` instance.

Since a new consumer subscribed to the topic, Kafka is triggering now a rebalance of our consumers. This is done in the background and we might not receive a message from our topic until this is done. We would like to avoid timing issues, therefore we have two possible options:

1. We could configure our consumer to always start from the beginning. Therefore we would need to set the property `AUTO_OFFSET_RESET_CONFIG` to `earliest`:

```
configs.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
```

   This would do the job pretty well in our simple example but has some disadvantages in case we would like to ignore some of the messages we have seen before.

2. We can call `consumer.poll(0)`, which would actually wait until we are subscribed, even with the timeout `0` (first parameter). Because of the timeout `0`, we would not wait for our actual message which we haven't send yet.

   This would result in the following code:

```
Map<String, Object> configs = new HashMap<>(KafkaTestUtils.consumerProps("consumer", "false", kafkaEmbedded));
consumer = new DefaultKafkaConsumerFactory<>(configs, new StringDeserializer(), new StringDeserializer()).createConsumer();
consumer.subscribe(singleton(TOPIC));
consumer.poll(0);
```

Afterward, you are able to configure your consumer with the Spring wrapper `DefaultKafkaConsumerFactory` or with the Kafka Java API.

After execution the test you should close the consumer with `consumer.close()`.

Configure Kafka Producer

Configuring the Kafka Producer is even easier than the Kafka Consumer:

```
Map<String, Object> configs = new HashMap<>(KafkaTestUtils.producerProps(kafkaEmbedded));
Producer<String, String> producer = new DefaultKafkaProducerFactory<>(configs, new StringSerializer(), new StringSerializer()).createProducer();
```

In case you don't have a `KafkaEmbedded` instance you could also use `KafkaTestUtils.senderProps(String brokers)` to get actual properties.

Produce and Consume Messages

Since we now have a consumer and a producer, we are actually able to produce messages:

```
producer.send(new ProducerRecord<>(TOPIC, "my-aggregate-id", "my-test-value"));
producer.flush();
```

And also consume messages and doing assertions on them:

```
ConsumerRecord<String, String> singleRecord = KafkaTestUtils.getSingleRecord(consumer, TOPIC);
assertThat(singleRecord).isNotNull();
assertThat(singleRecord.key()).isEqualTo("my-aggregate-id");
assertThat(singleRecord.value()).isEqualTo("my-test-value");
```

For consuming records there are the following methods on `KafkaTestUtils`:

- `getSingleRecord(Consumer<K, V> consumer, String topic): ConsumerRecord<K, V>`
- `getSingleRecord(Consumer<K, V> consumer, String topic, long timeout): ConsumerRecord<K, V>`
- `getRecords(Consumer<K, V> consumer): ConsumerRecords<K, V>`
- `getRecords(Consumer<K, V> consumer, long timeout): ConsumerRecords<K, V>`

The default timeout is 60 seconds, what is pretty long for testing. You might like to specify a smaller timeout, the unit therefore is milliseconds.



Serialize and Deserialize Key and Value

Above you can configure your serializers and de-serializers as you want. In case you have inheritance and you have an abstract parent class or an interface your actual implementation might be in the test case. In this case, you will get the following exception:

```
Caused by: java.lang.IllegalArgumentException: The class 'com.example.kafkatestsample.infrastructure.kafka.TestDomainEvent' is not in the trusted packages: [java.util,
java.lang, com.example.kafkatestsample.event]. If you believe this class is safe to deserialize, please provide its name. If the serialization is only done by a trusted
source, you can also enable trust all (*).
```

You can solve that by adding the specific package or all packages as trusted.

(https://www.mimacom.com/blog/about#officesanchor)

```
1 JsonDeserializer<DomainEvent> domainEventJsonDeserializer = new JsonDeserializer<>(DomainEvent.class);
2 domainEventJsonDeserializer.addTrustedPackages("*");
```

Conclusion

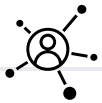
It's easy to test a Kafka integration once you have your setup working. The `@EmbeddedKafka` is providing a handy annotation to get started. With the running embedded Kafka, there are a couple of tricks necessary like the `consumer.poll(0)` and the `addTrustedPackages` that you would not necessarily experience when you are testing manually. You can also check out the complete test source code at GitHub (https://gist.github.com/vzickner/577c53164a97b9918a49e6c0235813f4).

About the author: Valentin Zickner

Is working since 2016 at mimacom as a Software Engineering. He has a lot of experience with cloud technologies, in addition he is specialized to Spring and Elasticsearch.

COMMENTS








Are you creative and passionate about software development? Do you think unconventionally and act with initiative? Do you want to achieve great things within our team?

Check out Our Job Offers (http://www.mimacom.com/en/jobs/)