(https:

anchor

Testing an Apache Kafka Integration within a Spring Boot Application and JUnit 5

March 29, 2020

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/)

Almost two years have passed since I wrote my first integration test for a Kafka Spring Boot application. It took me a lot of research to write this first integration test and I eventually ended up to write a blog post on testing Kafka with Spring Boot (https://blog.mimacom.com/testing-apache-kafka-with-spring-boot/). There was not too much information out there about writing those tests and at the end it was really simple to do it, but undocumented. I have seen a lot of feedback and interaction with my previous blog post and the GitHub Gist (https://gist.github.com/vzickner/577c53164a97b9918a49e6c0235813f4). Since then `spring-kafka-test` changed two times the usage pattern and JUnit 5 has been introduces. That means the code is now out of date and with that, also the blog post. This is the reason why I decided to create a revised version of the previous blog post.

Project Setup

Either use your existing Spring Boot project or generate a new one on start.spring.io (https://start.spring.io). When you select `Spring for Apache Kafka` at start.spring.io (https://start.spring.io) it automatically adds all necessary dependency entries into the maven or gradle file. By now it comes with JUnit 5 as well, so you are ready to go. However, if you have an older project you might 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 easiest way to start a test is to simply annotate the class with `@EmbeddedKafka`. This allows you to inject the `EmbeddedKafkaBroker` to either your test method or in a setup method at the beginning.

```
@EmbeddedKafka
public class SimpleKafkaTest {

    private EmbeddedKafkaBroker embeddedKafkaBroker;

    @BeforeEach
    void setUp(EmbeddedKafkaBroker embeddedKafkaBroker) {
        this.embeddedKafkaBroker = embeddedKafkaBroker;
    }

    // ...

}
```

You might have recognized that there is no Spring annotation in this class. Without annotating it with `@ExtendWith(SpringExtension.class)` or an extension which implies this (e.g. `@SpringBootTest`) the test is executed outside of the spring context and for example expressions might not be resolved.

There are a couple of properties available to influence the behavior and size of the embedded Kafka node. Including the following:

- `count`: number of brokers, the default is `1`
- `controlledShutdown`, the default is `false`
- `ports`, list of ports in case you would like to access those brokers from another instance
- `partitions`, the default is `2`
- `topics` names of the topics to be created at the startup
- `brokerProperties / brokerPropertiesLocation` additional properties for the Kafka broker

Class Configuration with Spring Context

Assuming you would also like to have the advantages of the Spring context, you need to add the `@SpringBootTest` annotation to the above test case. However, when you are not changing to the Spring context you also need to change the way how to autowire your `EmbeddedKafkaBroker`, otherwise you will get the following error:

```
org.junit.jupiter.api.extension.ParameterResolutionException:
Failed to resolve parameter [org.springframework.kafka.test.EmbeddedKafkaBroker embeddedKafkaBroker]
in method [void com.example.demo.SimpleKafkaTest.setUp(org.springframework.kafka.test.EmbeddedKafkaBro
Could not find embedded broker instance
```

The resolution is quiet simple, you need to change the autowiring from the JUnit 5 way to the `@Autowired` annotation from Spring:

```
@EmbeddedKafka
@ExtendWith(SpringExtension.class)
public class SimpleKafkaTest {

    @Autowired
    private EmbeddedKafkaBroker embeddedKafkaBroker;

    // ...

}
```

Configure Kafka Consumer

Now you are able to configure your consumer or producer, let's start with the consumer:

```
Map<String, Object> configs = new HashMap<>(KafkaTestUtils.consumerProps("consumer", "false", embedded
DefaultKafkaConsumerFactory<String, String> consumerFactory = new DefaultKafkaConsumerFactory<>(
    configs,
    new StringDeserializer(),
    new StringDeserializer()
);
```

(https:

anchor

`KafkaTestUtils.consumerProps` is providing you everything what you need to do the configuration. 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 `EmbeddedKafkaBroker` instance.

Afterwards, you can configure your consumer with the Spring wrapper `DefaultKafkaConsumerFactory`.

We could now go ahead and subscribe the consumer to a topic. Since it's the first consumer which is subscribing, it is going ahead and doing an initial assignment by the coordinator. The coordinator is assign the available partitions to the available consumers. In the previous blog post (https://blog.mimacom.com/testing-apache-kafka-with-spring-boot/) I have shown to options how to handle the waiting period. After revisiting those methods, I don't like both of them considering the current possibilities:

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`. This however doesn't work in case you would like to ignore previous messages.
2. We can call `consumer.poll(0)`, which would actually wait until we are subscribed, even with the timeout `0` (first parameter). This did and is doing the job pretty well. However, the method is marked as deprecated in version 2.0 and the reason therefore is that it could cause infinite blocking (https://cwiki.apache.org/confluence/x/5kiHB). `consumer.poll(0)` was waiting until the meta data was updated without counting it against the timeout. There is a replacement method which is `consumer.poll(Duration)`. This method is supposed to wait only until the timeout until the assignment is done. In practice, this method hasn't always worked as I expected since sometimes the metadata update was too fast and it waited for the first message.

Nowadays the Kafka Test documentation (https://docs.spring.io/spring-kafka/reference/html/#example) is recommending another approach which allows us to wait by using a `KafkaMessageListenerContainer`:

```
1 KafkaMessageListenerContainer<String, String> container = new KafkaMessageListenerContainer<>(consumer
2 BlockingQueue<ConsumerRecord<String, String>> records = new LinkedBlockingQueue<>();
3 container.setupMessageListener((MessageListener<String, String>) records::add);
4 container.start();
5 ContainerTestUtils.waitForAssignment(container, embeddedKafkaBroker.getPartitionsPerTopic());
```

This container has a message listener and writing them as soon as they are received to a queue. In our test itself, we can read the consumer records from the queue and the queue will block until we are receiving the first record. By using the `ContainerTestUtil.waitForAssignment` we are waiting for the initial assignment, since we wait explicitly for it.

We also need to `stop()` our container afterwards, to ensure that we have a clean context in a multi-test scenario. This is how the complete setup could look like:



(https:

anchor

```
@EmbeddedKafka
@ExtendWith(SpringExtension.class)
public class SimpleKafkaTest {

    private static final String TOPIC = "domain-events";

    @Autowired
    private EmbeddedKafkaBroker embeddedKafkaBroker;

    BlockingQueue<ConsumerRecord<String, String>> records;

    KafkaMessageListenerContainer<String, String> container;

    @BeforeEach
    void setUp() {
        Map<String, Object> configs = new HashMap<>(KafkaTestUtils.consumerProps("consumer", "false",
        DefaultKafkaConsumerFactory<String, String> consumerFactory = new DefaultKafkaConsumerFactory<
        ContainerProperties containerProperties = new ContainerProperties(TOPIC);
        container = new KafkaMessageListenerContainer<>(consumerFactory, containerProperties);
        records = new LinkedBlockingQueue<>();
        container.setupMessageListener((MessageListener<String, String>) records::add);
        container.start();
        ContainerTestUtils.waitForAssignment(container, embeddedKafkaBroker.getPartitionsPerTopic());
    }

    @AfterEach
    void tearDown() {
        container.stop();
    }

    // our tests...
}
```

(https:

anchor

Configure Kafka Producer

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

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

In case you don't have an `EmbeddedKafkaBroker` 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 = records.poll(100, TimeUnit.MILLISECONDS);
assertThat(singleRecord).isNotNull();
assertThat(singleRecord.key()).isEqualTo("my-aggregate-id");
assertThat(singleRecord.value()).isEqualTo("my-test-value");
```

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:

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

Improve Execution Performance for Multiple Tests

In case we have multiple tests, our setup is starting and stopping the Kafka Broker for each test. To improve this behavior, we can use a JUnit 5 feature to say that we would like to have the same class instance. This can be done with the annotation `@TestInstance(TestInstance.Lifecycle.PER_CLASS)`. We can convert our `@BeforeEach` and `@AfterEach` to `@BeforeAll` and `@AfterAll`. The only thing that we need to ensure is, that each test in the class is consuming all messages, which are produced in the same test. The setup looks now like this:

(https:

anchor

```
@EmbeddedKafka
@ExtendWith(SpringExtension.class)
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
public class SimpleKafkaTest {

    private static final String TOPIC = "domain-events";

    @Autowired
    private EmbeddedKafkaBroker embeddedKafkaBroker;

    BlockingQueue<ConsumerRecord<String, String>> records;

    KafkaMessageListenerContainer<String, String> container;

    @BeforeAll
    void setUp() {
        Map<String, Object> configs = new HashMap<>(KafkaTestUtils.consumerProps("consumer", "false",
        DefaultKafkaConsumerFactory<String, String> consumerFactory = new DefaultKafkaConsumerFactory<
        ContainerProperties containerProperties = new ContainerProperties(TOPIC);
        container = new KafkaMessageListenerContainer<>(consumerFactory, containerProperties);
        records = new LinkedBlockingQueue<>();
        container.setupMessageListener((MessageListener<String, String>) records::add);
        container.start();
        ContainerTestUtils.waitForAssignment(container, embeddedKafkaBroker.getPartitionsPerTopic());
    }

    @AfterAll
    void tearDown() {
        container.stop();
    }

    // …
}
```

(https:

anchor

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 JUnit 5 approach you can do similar tests for the usage with the Spring context and without. Since JUnit 5 allows us to specify how the class is executed, we can improve the execution performance for a single class easily. Once the running embedded Kafka is running, there are a couple of tricks necessary, e.g. bootstrapping the consumer and the `addTrustedPackages`. Those you would not necessarily experience when you are testing manually. You can also check out the complete source code of my example on testing Kafka with Spring Boot and JUnit 5 (https://gist.github.com/vzickner/577c53164a97b9918a49e6c0235813f4) in this GitHub Gist.

About the author: Valentin Zickner

our team!

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

(https:

anchor

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, Elasticsearch and Flowable.

COMMENTS

(https:

anchor

ALSO ON **MIMACOM**

Autocomplete with Elasticsearch - Part …

a year ago • 1 comment

In the previous articles, we look into Prefix Queries and Edge NGram Tokenizer …

DRY Kubernetes with Helm

2 years ago • 2 comments

One of the coolest facets of Kubernetes is the declarative deployment …

Code Folding for JSON Data

4 months ago • 1 comment

In Pair Programming sessions, we often exchange tips and tricks. …

Performin Text Class

2 years ago •

Text classific common tas machine lea

1 Comment mimacom Disqus' Privacy Policy 1 Login

Recommend Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Michael • 16 days ago

Great article which shows the basic principles of using an embedded Kafka server with a Spring Boot application. There are some topics which are left up to the reader, like for example an application typically has some @Configuration values which setup the Kafka producer and in a test like this you would need to override the relevant bean definitions. But this is just boilerplate work that most developers understand how to do.

Regarding JUnit5's @BeforeAll and @AfterAll, these annotations can only appear on static methods. So the examples probably won't work.

  • Reply • Share ›

(https:

anchor

JOIN US

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?