(https:

anchor

Probar una integración de Apache Kafka dentro de una aplicación Spring Boot y JUnit 5

29 de marzo de 2020

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

Primavera (https://blog.mimacom.com/tag/spring/)

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

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

Pruebas (https://blog.mimacom.com/tag/testing/)

Han pasado casi dos años desde que escribí mi primera prueba de integración para una aplicación Kafka Spring Boot. Me llevó mucha investigación escribir esta primera prueba de integración y finalmente terminé escribiendo una publicación de blog sobre probar Kafka con Spring Boot (https://blog.mimacom.com/testing-apache-kafka-with-spring-boot/) . No había demasiada información sobre cómo escribir esas pruebas y al final fue muy simple hacerlo, pero indocumentado. He visto muchos comentarios e interacción con mi publicación de blog anterior y el GitHub Gist (https://gist.github.com/vzickner/577c53164a97b9918a49e6c0235813f4) . Desde entonces `spring-kafka-test` cambió dos veces el patrón de uso y se introdujo JUnit 5. Eso significa que el código ahora está desactualizado y con eso, también la publicación del blog. Esta es la razón por la que decidí crear una versión revisada de la publicación de blog anterior.

Configuración del proyecto

Utilice su proyecto Spring Boot existente o genere uno nuevo en start.spring.io (https://start.spring.io) . Al seleccionar `Spring for Apache Kafka` a start.spring.io (https://start.spring.io) se agrega automáticamente todas las entradas de dependencias necesarias en el experto o archivo Gradle. Por ahora viene con JUnit 5 también, por lo que está listo para comenzar. Sin embargo, si tiene un proyecto anterior, es posible que deba agregar la `spring-kafka-test` dependencia:

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

Configuración de clase

La forma más fácil de comenzar una prueba es simplemente anotar la clase con `@EmbeddedKafka` . Esto le permite inyectar el `EmbeddedKafkaBroker` método de prueba o un método de configuración al principio.

```
@EmbeddedKafka
public class SimpleKafkaTest {

    private EmbeddedKafkaBroker embeddedKafkaBroker;

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

    // ...

}
```

Es posible que haya reconocido que no hay anotaciones Spring en esta clase. Sin anotarlo con `@ExtendWith(SpringExtension.class)` o con una extensión que implique esto (por ejemplo `@SpringBootTest`), la prueba se ejecuta fuera del contexto de Spring y, por ejemplo, las expresiones pueden no resolverse.



(https:

anchor

Hay un par de propiedades disponibles para influir en el comportamiento y el tamaño del nodo Kafka incrustado. Incluyendo lo siguiente:

- `count`: número de corredores, el valor predeterminado es `1`
- `controlledShutdown`, el valor predeterminado es `false`
- `ports`, lista de puertos en caso de que desee acceder a esos intermediarios desde otra instancia
- `partitions`, el valor predeterminado es `2`
- `topics` nombres de los temas que se crearán en el inicio
- `brokerProperties` / `brokerPropertiesLocation` propiedades adicionales para el corredor Kafka

Configuración de clase con contexto de primavera

Suponiendo que también desea tener las ventajas del contexto Spring, debe agregar la `@SpringBootTest` anotación al caso de prueba anterior. Sin embargo, cuando no está cambiando al contexto Spring, también necesita cambiar la forma de conectar automáticamente su `EmbeddedKafkaBroker`, de lo contrario obtendrá el siguiente 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
```

La resolución es silenciosa y simple, debe cambiar el cableado automático de la forma JUnit 5 a la `@Autowired` anotación de Spring:

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

    @Autowired
    private EmbeddedKafkaBroker embeddedKafkaBroker;

    // ...

}
```

Configurar Kafka Consumer

Ahora puede configurar su consumidor o productor, comencemos con el consumidor:

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

`KafkaTestUtils.consumerProps` le proporciona todo lo que necesita para hacer la configuración. El primer parámetro es el nombre de su grupo de consumidores, el segundo es un indicador para establecer la confirmación automática y el último parámetro es la `EmbeddedKafkaBroker` instancia.

Luego, puede configurar su consumidor con Spring wrapper `DefaultKafkaConsumerFactory`.

Ahora podríamos seguir adelante y suscribir al consumidor a un tema. Dado que es el primer consumidor que se suscribe, sigue adelante y realiza una asignación inicial por parte del coordinador. El coordinador asigna las particiones disponibles a los consumidores disponibles. En la publicación de blog anterior, (https://blog.mimacom.com/testing-apache-kafka-with-spring-boot/) le mostré a las opciones cómo manejar el período de espera. Después de revisar esos métodos, no me gustan ambos teniendo en cuenta las posibilidades actuales:

1. Podríamos configurar a nuestro consumidor para que siempre comience desde el principio. Por lo tanto, tendríamos que establecer la propiedad `AUTO_OFFSET_RESET_CONFIG` en `earliest`. Sin embargo, esto no funciona en caso de que desee ignorar los mensajes anteriores.
2. Podemos llamar `consumer.poll(0)`, lo que en realidad esperaría hasta que nos suscribamos, incluso con el tiempo de espera `0` (primer parámetro). Esto hizo y está haciendo el trabajo bastante bien. Sin embargo, el método está marcado como obsoleto en la versión 2.0 y, por lo tanto, puede causar un bloqueo infinito (https://cwiki.apache.org/confluence/x/5kiHB) . `consumer.poll(0)` estaba esperando hasta que los metadatos se actualizaran sin contarlos en el tiempo de espera. Hay un método de reemplazo que es `consumer.poll(Duration)` . Se supone que este método debe esperar solo hasta el tiempo de espera hasta que se complete la asignación. En la práctica, este método no siempre ha funcionado como esperaba, ya que a veces la actualización de metadatos era demasiado rápida y esperaba el primer mensaje.

Hoy en día, la documentación de Kafka Test (https://docs.spring.io/spring-kafka/reference/html/#example) recomienda otro enfoque que nos permite esperar usando `KafkaMessageListenerContainer`:

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

Este contenedor tiene un escucha de mensajes y los escribe tan pronto como se reciben en una cola. En nuestra propia prueba, podemos leer los registros del consumidor de la cola y la cola se bloqueará hasta que recibamos el primer registro. Al usar el `ContainerTestUtil.waitForAssignment`, estamos esperando la asignación inicial, ya que la esperamos



(https:

ancho

explícitamente.

También necesitamos `stop()` nuestro contenedor después, para asegurarnos de tener un contexto limpio en un escenario de prueba múltiple. Así es como podría verse la configuración completa:

(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...
}
```

```
}
```

Configurar Kafka Producer

Configurar Kafka Producer es aún más fácil que Kafka Consumer:

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

En caso de que no tenga una `EmbeddedKafkaBroker` instancia, también podría usarla `KafkaTestUtils.senderProps(String brokers)` para obtener propiedades reales.

Producir y consumir mensajes

Como ahora tenemos un consumidor y un productor, podemos producir mensajes:

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

Y también consumir mensajes y hacer afirmaciones sobre ellos:

```
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");
```

Serializar y deserializar clave y valor

Arriba puede configurar sus serializadores y deserializadores como desee. En caso de que tenga herencia y tenga una clase principal abstracta o una interfaz, su implementación real podría estar en el caso de prueba. En este caso, obtendrá la siguiente excepción:

```
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
(*).
```

Puede resolver eso agregando el paquete específico o todos los paquetes como confiables:

(https:

anchor

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

Mejora el rendimiento de ejecución para múltiples pruebas

En caso de que tengamos varias pruebas, nuestra configuración comenzará y detendrá el Kafka Broker para cada prueba. Para mejorar este comportamiento, podemos usar una función JUnit 5 para decir que nos gustaría tener la misma instancia de clase. Esto se puede hacer con la anotación `@TestInstance(TestInstance.Lifecycle.PER_CLASS)`. Podemos convertir nuestro `@BeforeEach` y `@AfterEach` a `@BeforeAll` y `@AfterAll`. Lo único que debemos asegurar es que cada prueba de la clase consume todos los mensajes, que se producen en la misma prueba. La configuración ahora se ve así:

```
@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();
    }

    // …
}
```

Conclusión

Es fácil probar una integración de Kafka una vez que tenga su configuración funcionando. El `@EmbeddedKafka` proporciona una práctica anotación para comenzar. Con el enfoque JUnit 5 puede hacer pruebas similares para el uso con el contexto Spring y sin él. Dado que JUnit 5 nos permite especificar cómo se ejecuta la clase, podemos mejorar el rendimiento de ejecución para una sola clase fácilmente. Una vez que se está ejecutando el Kafka incrustado en ejecución, hay un par de trucos necesarios, por ejemplo, poner en marcha el consumidor y el `addTrustedPackages`. Aquellos que no necesariamente experimentaría cuando esté probando manualmente. También puede consultar el código fuente completo de mi ejemplo al probar Kafka con Spring Boot y JUnit 5 (https://gist.github.com/vzickner/577c53164a97b9918a49e6c0235813f4) en este GitHub Gist.

(https:

ancho

Sobre el autor: Valentin Zickner

Trabaja desde 2016 en mimacom como ingeniero de software. Tiene mucha experiencia con tecnologías en la nube, además está especializado en Spring, Elasticsearch y Flowable.

COMENTARIOS

(https:

ancho

¿Eres creativo y apasionado por el desarrollo de software? ¿Piensas poco convencionalmente y actúas con iniciativa? ¿Quieres lograr grandes cosas dentro de nuestro equipo?

Mira nuestras ofertas de trabajo (https://www.mimacom.com/en/jobs/)

ALSO ON **MIMACOM**

Elasticsearch Scoring Changes In Action

5 months ago • 2 comments

For the ones started their journey with Elasticsearch before version 5.x …

Persistent Storage with Red Hat OpenShift …

a year ago • 2 comments

This article is a technical summary with my experience of the Red …

Similarity Search in Vector Space with …

a year ago • 1 comment

Full-text search is one of Elasticsearch's strengths. However, what if we …

Autocomp Elasticsea

a year ago • 3

You know se Google or D you start typ

(https:

anchor

0 Comments mimacom **Disqus' Privacy Policy** **Javier Martín Alon…**

Recommend Tweet Share **Sort by Best**

Start the discussion…

Be the first to comment.

Subscribe **Add Disqus to your siteAdd DisqusAdd** **Do Not Sell My Data**

ÚNETE A NOSOTROS