# Testing A Kafka Event Producer With Spring Boot And JUnit 5 Part 1

15 OCTOBER 2020 // **MARTIN HOLT**

A successful continuous delivery (CD) pipeline requires a high level of automated testing. It is essential that tests are reliable to ensure that nothing unexpected slips into your production environment. Swift execution is also desirable to provide timely feedback to developers.

Testing asynchronous processes provide a different set of challenges from testing a synchronous request-response scenario. In this 2 part blog post I will investigate how to test an application that publishes events via Kafka. In part 1 I will demonstrate a method for getting started with integration testing and in part 2 I will look at how this can be made faster.

The scenario presented in these blog posts is inspired by a real-life case. The following link will take you to the source code for the example application.

## SCENARIO

The example application uses Spring Boot to expose a single RESTful api. A call to the `dosomething` endpoint will perform some internal state change, publish some events and finally send a response back to the client.

Events are published in a JSON format that looks something like this:

```
{
  "transaction_id": "91e03b7d-d348-4eb8-8aa4-509d3a7ec762",
  "sequence_id": 1,
  "padding": "some_large_string...."
}
```

In this case the `transaction_id` is intended to be a unique id related to the request. A single transaction will consist pf 5 events and each event will have a unique `sequence_id` value. The `padding` field is intended to simulate a large message.

In the real-life scenario that this example is based on the event stream was used by a downstream application to apply each state change to its own representation of a business object. The intended aim was to maintain a consistent state for the object in both applications. This required that the downstream application consumed events in sequential order. The requirement proved difficult to enforce on the consumer so, as a compromise, the publisher guaranteed to publish events in ascending order.

This requirement has been applied to the example application and the publisher looks something like this:

```
public ResponseEntity<Transaction> doSomething() {
  ...
  // Publish five events
  publisher.publish(transactionId, 1);
  publisher.publish(transactionId, 2);
  publisher.publish(transactionId, 3);
  publisher.publish(transactionId, 4);
  publisher.publish(transactionId, 5);
  ...
}
```

A Kafka topic is split into a number of partitions. As we will see in part 2 sequential consumption is difficult if events are spread across multiple partitions. To make life easier for the consumer the publisher guaranteed to publish all events for a transaction to the same partition. For simplicity the example application will publish all events to a single partition:

```
public void publish(UUID transcationId, Integer sequenceId) {
  ...
  Integer partition = Integer.valueOf(1);
  var record = new ProducerRecord<>(TOPIC, partition, key, message);
  ...
}
```

Now that the publisher is in place the next step is to test that the guarantee of sequential publishing holds.

## INTEGRATION TESTING WITH GRADLE

Both the Spring Boot application and a Kafka broker must be available in order to test the publisher. I will use the term "integration test" for testing how the running Spring Boot application behaves with the various integration points. Integration tests will be performed seperately from unit tests (and then only if unit tests have completed successfully).

The Gradle build tool has been chosen to build and test the application. The Gradle project contains the `Java` plugin which performs unit testing as part of the build. I have extended this by adding a custom `integrationTask` task to run the integration tests using a seperate source set with path `src/it/java`.

Some other tasks that have been added to the Gradle build are:

- `buildImage` - builds a Docker image containing a Java 11 JRE and the Spring Boot application as an executable Jar.
- `startServices` - uses Docker Compose to start a Kafka broker and the Spring Boot application
- `stopServices` - uses Docker Compose to stop the services (cleans up resources after test execution)

Executing the Gradle `integrationTest` task (using `./gradlew integrationTest`) will build the application, start the services, and once the tests are complete stop them all.

## THE FIRST INTEGRATION TEST

The first test will use JUnit5 and the Java HTTP client to send a request to the `/dosomething` endpoint. The test will then verify that the request was accepted successfully and will then check that 5 events have been published and that the events are in sequential order:

```
@Test
public void testForSequentialIntegrity()
  // Given: a request is to be sent to the "do something" endpoint
  HttpRequest request = ...

  // When the request is sent
  HttpResponse<String> response = ...

  // Then the request is successful
  assertEquals(200, response.statusCode());

  // And 5 events have been published
  var events = ...
  assertEquals(5, events.size());

  // Ensure that the sequence of the exents is correct
  var sequenceIds = events.stream()
                          .map(json -> JsonPath.parse(json).read("$['sequenceId']", Integ
                          .collect(Collectors.toList());
  assertEquals(Integer.valueOf(1), sequenceIds.get(0));
  assertEquals(Integer.valueOf(2), sequenceIds.get(1));
  assertEquals(Integer.valueOf(3), sequenceIds.get(2));
  assertEquals(Integer.valueOf(4), sequenceIds.get(3));
  assertEquals(Integer.valueOf(5), sequenceIds.get(4));
```

Running this test with Gradle (using `./gradlew integrationTest`) will more than likely fail. Why?

## TIMING

The tasks above use the Gradle `Exec` task to execute a command line instruction - in this case starting the application in a Docker container. Although the application container starts quickly the Spring Boot application itself takes a few seconds to start up. Gradle is unaware of the state of the application and eagerly starts to run the integration tests which then fails. The integration test needs some way to check the readiness of the application under test.

In this example I will use the Spring Boot Actuator to pause test execution until the application is healthy. A JUnit5 extension is introduced that extends the behaviour of the `BeforeAll` lifecycle phase. Test execution is paused whilst the extension polls the actuator's `healthcheck` endpoint. If the `healthcheck` endpoint is not healthy after a reasonable amount of time, one minute say, then the tests will be aborted:

```
public class CheckAvailabilityExtension implements BeforeAllCallback {

  @Override
  public void beforeAll(ExtensionContext context) {

    final Instant stopTime = Instant.now().plusSeconds(60);

    boolean isAvailable = false;
    while (!isAvailable && stopTime.isAfter(Instant.now())) {
      // call the healthcheck endpoint and check for "UP"
      // if not sleep for 1s
    }
    if (!isAvailable) {
      throw new PreconditionViolationException("Unable to get healthy indicator from app
    }
}
```

This example contains only contains one test class so the simplest way to add the extension is to annotate the test class:

```
@ExtendWith({CheckAvailabilityExtension.class})
public class DoSomethingEndpointTest {
```

Now the integration test can be certain that the Spring Boot application is ready when tests are executed. Another benefit of this approach is that this acts as an indirect test that the healthcheck endpoint is working as expected.

Running this test will give a successful result!

## WHAT ABOUT OUR ASSUMPTIONS?

This test make two assumptions:

- events will be published sequentially to ensure sequential consumption
- all events consumed during test execution are related to the test case

What happens when we challenge these assumptions?

## UNLEASHING THE PRODUCER

As stated before there is a requirement that consumption must be performed sequentially. This is not actually a requirement on the producer - the producer is just being helpful. Being helpful does have consequences - in the real-life case the response was sent to the client once all events had been published adding between 100ms to 200ms to the elapsed time of the request, which was quite significant.

Let's speed up the example application by publishing events asynchronously. First enable asynchronous processing:

```
@EnableAsync
@SpringBootApplication
public class EventsourceApplication {
  ...
}
```

And make publication asynchronous by using the `@Async` annotation. Let's also make things more complicated by allowing the publisher to choose any of the 10 partitions on the topic:

```
@Async
public void publish(UUID transcationId, Integer sequenceId) {
  ...
  Integer partition = ThreadLocalRandom.current().nextInt(10);
  var record = new ProducerRecord<>(TOPIC, partition, key, message);
  ...
```

*Note: this is a naive example where failure to publish a message has no consequences for the request as a whole. Publishing an event becomes a "best effort" task.*

If we look in the application logs we can see that events are no longer being published sequentially:

```
2020-10-11 13:00:36.275  INFO [8f97a0801992901f] task-8: Publishing transaction x and seq
2020-10-11 13:00:36.285  INFO [8f97a0801992901f] task-3: Publishing transaction x and seq
2020-10-11 13:00:36.291  INFO [8f97a0801992901f] task-1: Publishing transaction x and seq
2020-10-11 13:00:36.292  INFO [8f97a0801992901f] task-4: Publishing transaction x and seq
2020-10-11 13:00:36.416  INFO [8f97a0801992901f] task-2: Publishing transaction x and seq
```

Running the test now will result in a consistent failure.

## TESTING WITHOUT SEQUENCE

Sequential publication is no longer a requirement so the test can be altered to verify that in the 5 published events there are 5 unique sequence ids from 1 to 5 (inclusive):

```
@Test
public void testThatAllEventsArePublished() {

  // Given: a request is to be sent to the "do something" endpoint
  HttpRequest request = ...

  // When the request is sent
  HttpResponse<String> response = ...

  // Then the request is successful
  assertEquals(200, response.statusCode());

  // And 5 events have been published
  var events = ...
  assertEquals(5, events.size());

  // And the events had sequence ids from 1 to 5 (inclusive)
  var sequenceIds = events.stream()
                          .map(json -> JsonPath.parse(json).read("$['sequenceId']", Inte
                          .collect(Collectors.toList());
  IntStream.range(1, 6).forEach(i -> assertTrue(sequenceIds.contains(i)));
}
```

And the test succeeds again.

## MAKE SOME NOISE

Let's now challenge the assumption that all published events are related to the test. This is an unlikely assumption in a complicated application and certainly not true in a production environment.

To simulate this we will introduce the `NoiseMaker` that will publish a random event every 3 seconds:

```
public class NoiseMaker {

    @Scheduled(fixedRate = 3000l)
    public void makeSomeNoise() {
    publisher.publish(UUID.randomUUID(), -1);
  }
}
```

Now tests become flakey - sometimes, but not always, a 6th or 7th event is consumed during the 10 seconds of polling. What is needed is a way to seperate events that are related to the test from those that can be ignored.

## LINKING TESTS WITH EVENTS USING SPRING CLOUD SLEUTH

There are many ways to approach linking tests with events. It is not considered good practice to engineer an application to meet the needs of the test harness (but sometimes it is necessary). I have chosen to use Spring Cloud Sleuth to provide a relatively unobtrusive method of linking using distributed tracing. In this case I will argue that a distributed tracing feature can be considered of value to the application.

Spring Cloud Sleuth has been added to the Gradle project:

```
dependencies {
  ...
  implementation("org.springframework.cloud:spring-cloud-starter-sleuth:2.2.5.RELEASE")
```

Every incoming request will start a new Span which effectively assigns each incoming request a unique trace id. In the example logs above you can see that `8f97a0801992901f` is a `trace id`. The next step is to make the test aware of the of the `trace id`.

First the event schema is extended to add some metadata that contains the `trace id`:

```
{
  "metadata": {
    "trace_id": "8f97a0801992901f"
  },
  "transaction_id": "91e03b7d-d348-4eb8-8aa4-509d3a7ec762",
  "sequence_id": 1,
  "padding": "some_large_string...."
}
```

Next a `TraceFilter` filter adds the `trace id` to an Http header on the response called `x-b3-traceid`.

```
@WebFilter("/*")
public class TraceFilter implements Filter {

  private static final String TRACE_ID_HEADER = "x-b3-traceid";

  @Autowired
  Tracer tracer;

  @Override
  public void doFilter(ServletRequest request, ServletResponse response, FilterChain cha

    var httpResponse = (HttpServletResponse) response;
    var traceId = tracer.currentSpan().context().traceIdString();
    httpResponse.setHeader(TRACE_ID_HEADER, traceId);
    chain.doFilter(request, response);
  }
}
```

Now our test can match the `trace id` in the response to that in the metadata of the event. This allows us to simply filter and ignore any events that are not relevant to our test during execution time!

```
@Test
public void testThatAllEventsArePublished() {

  // Given: a request is to be sent to the "do something" endpoint
  HttpRequest request = ...

  // When the request is sent
  HttpResponse<String> response = ...

  // Then the request is successful
  assertEquals(200, response.statusCode());

    // And the response contains a trace id
    var traceIdHeader = response.headers().firstValue(TRACE_ID_HEADER);
    assertTrue(traceIdHeader.isPresent());
    var traceId = traceIdHeader.get();

  // And 5 events have been published
  var events = ...
  assertEquals(5, events.size());

  // And the events had sequence ids from 1 to 5 (inclusive)
  var sequenceIds = events.stream()
                          .filter(json -> traceId.equals(JsonPath.parse(json).read("$['me
                          .map(json -> JsonPath.parse(json).read("$['sequenceId']", Integ
                          .collect(Collectors.toList());
  IntStream.range(1, 6).forEach(i -> assertTrue(sequenceIds.contains(i)));
}
```

And the test succeeds again!

## SUMMARY

This blog post introduces a pattern using JUnit5 for integration testing applications that publish events via Kafka. This pattern can be extended and repeated to provide a rich and full test suite as the application evolves.

In part 2 I will discuss what happens when the test suite grows, look in depth at the Kafka Consumer, and offer one solution for how to reduce long execution times.