

# Apache Kafka

## Integration Testing

### of Spring Boot Microservices



A mini guide to testing  
Kafka Producer and Consumer

By Sergey Kargopolov

<b>About this book.....</b>	<b>4</b>
<b>About the Author .....</b>	<b>5</b>
<b>Source Code .....</b>	<b>6</b>
<b>Configure Testing Support .....</b>	<b>7</b>
Maven Dependencies.....	7
<b>Kafka Producer .....</b>	<b>9</b>
<b>System Under Test.....</b>	<b>9</b>
Method Under Test .....	9
Kafka Producer Configuration .....	12
<b>Testing Kafka Producer .....</b>	<b>14</b>
Creating a Test Class .....	14
Creating New Test Method .....	18
Ways to Run a Test Method .....	20
Test Method Structure. The AAA Pattern. ....	23
Arrange Section.....	23
Act Section .....	24
Consumer Configuration in a Test Class .....	25
Assert Section .....	28
Running the Test Method .....	30
<b>Verify Producer Configuration Properties.....</b>	<b>31</b>
Understanding Idempotent Producers in Apache Kafka.....	31
Configuring Idempotent Producers.....	31
Test Class for Idempotent Producer.....	36
Test Method for Idempotent Producer.....	36
<b>Testing Kafka Consumer.....</b>	<b>39</b>
Kafka Consumer Application Code.....	39
Method Under Test .....	42
Creating a Test Class .....	43
Creating a Test Method .....	44
Running the Test Method .....	49

**Thank You and Next Steps .....50**

# ABOUT THIS BOOK

Welcome to this mini e-book on writing integration tests for Kafka Producer and Kafka Consumer Spring Boot applications.

In 2024, I created a video course for absolute beginners, teaching them how to build Event-Driven Spring Boot Microservices with Apache Kafka. However, the course intentionally does not cover how to write integration tests. It is designed for absolute beginners, and I did not want to take students' attention away from the main content to learn how to write integration tests.

This mini e-book is designed to complement that course. It provides a focused walkthrough on how to write integration tests for basic Kafka Producer and Kafka Consumer Spring Boot applications.

While this book complements the video course, it stands independently. You do not need to enrol in the video course to benefit from this book. However, a basic understanding of Apache Kafka and the workings of Kafka Producer and Kafka Consumer is required to fully grasp the content of this book.

Throughout the book, I explain all the crucial aspects of Producer and Consumer configuration. Additionally, you have the option to download the source code for both applications, as well as the final versions of the test methods from a git repository created specifically for this book.

By the end of this book, you will be able to:

- Write an integration test for a Kafka Producer,
- Verify Producer Configuration properties. For example, we will write an integration test to verify that the Kafka producer is indeed configured as an Idempotent Producer,
- Write an integration test for a Kafka Consumer.

Please refer to the table of contents for additional details.

I hope this mini e-book proves to be a valuable resource for you.

Sincerely,

Sergey Kargopolov

# ABOUT THE AUTHOR

Hi, I'm Sergey Kargopolov, a Software Developer with a passion for teaching.

I create video courses for fellow developers, especially those just starting out. If you would like to stay connected, find me on [LinkedIn](#). I've designed a variety of courses, each aimed at providing maximum value. Feel free to explore them and find one that suits your interests. Click any of them to learn more.



## Apache Kafka for Event-Driven Spring Boot Microservices

Sergey Kargopolov

4.7 ★★★★★ (211)

10 total hours · 142 lectures · Beginner

Highest rated



## Spring Boot Microservices and Spring Cloud. Build & Deploy.

Sergey Kargopolov

4.7 ★★★★★ (7,930)

25 total hours · 338 lectures · All Levels

Highest rated



## Testing Java with JUnit 5 & Mockito

Sergey Kargopolov

4.6 ★★★★★ (307)

7.5 total hours · 113 lectures · All Levels

Highest rated



## OAuth 2.0 in Spring Boot Applications

Sergey Kargopolov

4.5 ★★★★★ (2,464)

11 total hours · 187 lectures · All Levels

Bestseller



## AWS Serverless REST APIs for Java Developers. CI/CD included

Sergey Kargopolov

4.8 ★★★★★ (166)

14.5 total hours · 237 lectures · Beginner

Highest rated



## Event-Driven Microservices, CQRS, SAGA, Axon, Spring Boot

Sergey Kargopolov

4.7 ★★★★★ (1,486)

9 total hours · 164 lectures · Beginner

Highest rated



## RESTful Web Services, Java, Spring Boot, Spring MVC and JPA

Sergey Kargopolov

4.5 ★★★★★ (3,650)

20 total hours · 297 lectures · Beginner

Bestseller



## Deploy Spring Boot Microservices on AWS ECS with Fargate

Sergey Kargopolov

4.8 ★★★★★ (205)

7 total hours · 95 lectures · Beginner

Highest rated

# SOURCE CODE

This book will teach you how to write integration tests for two basic Microservices that act as a Kafka Producer and a Kafka Consumer.

You might want to have a copy of these Microservices on your computer for reference.

Below are the links to both the initial and final versions of these projects.

## Initial version(before adding test methods)

- [Products Microservice\(Producer\)](#),
- [Email Notification Microservice\(Consumer\)](#).

## Final version(After adding test methods)

- [Products Microservice\(Producer\)](#),
- [Email Notification Microservice\(Consumer\)](#).

## Shared dependency

- [Core](#)

If you're interested in seeing me create these two Microservices from scratch, explaining every single configuration, you can check out my video course: "[Apache Kafka for Event-Driven Spring Boot Microservices](#)".

Now that you have the source code, let's move on to the next chapter and start learning.



# CONFIGURE TESTING SUPPORT

In the previous chapter of this book, I shared the source code for two Microservices for which I am going to write integration tests. These Microservices were created using the [Spring Initializr project](#), and testing support was added to them by default.

However, if you do not use Spring Initializr to create Spring Boot projects, this section will explain how to configure your Spring Boot application to support Kafka Integration tests.

## MAVEN DEPENDENCIES

In this section, you will learn about the dependencies that need to be added to enable your Spring Boot application to support integration tests.

We will begin with two basic dependencies, and as we progress with our integration testing, we will add other dependencies as needed.

### Spring Boot Test dependency

The first very important dependency that your Spring Boot project needs for supporting both Unit Testing and Integration Testing is *spring-boot-starter-test*. Please open the *pom.xml* file of your Spring Boot project and check if this dependency is already included.

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

This dependency provides a set of libraries useful for testing Spring Boot applications. This includes libraries like **JUnit** (for unit tests), **Mockito** (for mocking objects in tests), and **Spring Test** (for integration testing support in Spring).

However, this dependency is not enough to write integration tests for the application that works with Apache Kafka.

### Spring Kafka Test Dependency

If you created your project using [Spring Initializr](#) tool and added Apache Kafka as one of the dependencies, then you will most likely already have `spring-kafka-test` dependency added to *pom.xml*. If not, then please add the following dependency to *pom.xml* file.

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

This dependency is provided by Spring Kafka and it is designed to help you write tests for applications that interact with Kafka.

For example, it provides a way to create a mock Kafka server, so you can test your application's Kafka interactions without needing to connect to a real Kafka server. This makes your tests faster and more reliable.

Now that you're familiar with the dependencies your Spring Boot project needs to support testing, let's move on to the next chapter.



# KAFKA PRODUCER

## SYSTEM UNDER TEST

In this chapter, we will be focusing on writing integration tests for a specific piece of code, known as the “System Under Test” (SUT). The SUT is the particular system or subsystem that we are testing. It’s the component we are interested in, the one we want to ensure is functioning correctly.

Our SUT here is a Spring Boot application that acts as a Kafka Producer. It’s a simple Spring Boot Microservice that uses `KafkaTemplate` to send a message to a Kafka topic.

You can download the source code from the Products Microservice [GitHub repository](#).

## METHOD UNDER TEST

In the context of writing tests, the “Method Under Test” (MUT) refers to the specific method that is being tested. This is the method for which you are writing a test.

The business logic to send a message to a Kafka topic is encapsulated in a method in a Java class annotated with the `@Service` annotation. It is this method for which I want to write an integration test. I aim to verify that the following code does indeed successfully send a message to a Kafka Topic.

```

@Override
public String createProduct(CreateProductRestModel productRestModel) throws
Exception {

    String productId = UUID.randomUUID().toString();

    ProductCreatedEvent productCreatedEvent = new ProductCreatedEvent(productId,
        productRestModel.getTitle(), productRestModel.getPrice(),
        productRestModel.getQuantity());

    ProducerRecord<String, ProductCreatedEvent> record = new ProducerRecord<>(
        "product-created-events-topic",
        productId,
        productCreatedEvent);
    record.headers().add("messageId", UUID.randomUUID().toString().getBytes());

    SendResult<String, ProductCreatedEvent> result =
        kafkaTemplate.send(record).get();

    LOGGER.debug("Partition: " + result.getRecordMetadata().partition());
    LOGGER.debug("Topic: " + result.getRecordMetadata().topic());
    LOGGER.debug("Offset: " + result.getRecordMetadata().offset());

    return productId;
}

```

Let me briefly explain what the above code does.

1. It receives an object of CreateProductRestModel data type as method argument. This object contains product details that were sent in HTTP request to create a new product. We will use these product details, to create and publish a new event.
3. It then generates a unique productId value and uses it to create a ProductCreatedEvent object.

```

String productId = UUID.randomUUID().toString();

ProductCreatedEvent productCreatedEvent = new ProductCreatedEvent(productId,
    productRestModel.getTitle(), productRestModel.getPrice(),
    productRestModel.getQuantity());

```

4. Next, it creates a ProducerRecord object. It will be this object that we will send as Kafka message.

```

ProducerRecord<String, ProductCreatedEvent> record = new ProducerRecord<>(
    "product-created-events-topic",
    productId,
    productCreatedEvent);

```

5. Our Kafka message will also have a unique `messageId` added to the message headers. The value of `messageId` will be used by the idempotent consumer Microservice to make sure that this message is processed only once. You will see how it is used in the following chapters.

```
record.headers().add("messageId", UUID.randomUUID().toString().getBytes());
```

6. And finally, I use the `KafkaTemplate` object to send Kafka message.

```
SendResult<String, ProductCreatedEvent> result =  
    kafkaTemplate.send(record).get();
```

My goal is to write an integration test to verify that this code does successfully send a message to the Kafka topic.

I hope I explained this method clearly enough. If you need further assistance with this code and want to see me create it from scratch, I've got a series of [video lessons](#). These lessons show how to build this Microservice application and set it up as a Kafka Producer right from the start, explaining each configuration step.

# KAFKA PRODUCER CONFIGURATION

I have placed the Kafka producer configuration into a Java class named `KafkaConfig`, which resides in the root package of my Spring Boot application.

```
@Configuration
public class KafkaConfig {

    // Configuration properties are here
    ...

    Map<String, Object> producerConfigs() {
        Map<String, Object> config = new HashMap<>();

        config.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        config.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, keySerializer);
        config.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, valueSerializer);
        config.put(ProducerConfig.ACKS_CONFIG, acks);
        config.put(ProducerConfig.DELIVERY_TIMEOUT_MS_CONFIG, deliveryTimeout);
        config.put(ProducerConfig.LINGER_MS_CONFIG, linger);
        config.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, requestTimeout);
        config.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, idempotence);
        config.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION,
inflightRequests);

        return config;
    }

    @Bean
    ProducerFactory<String, ProductCreatedEvent> producerFactory() {
        return new DefaultKafkaProducerFactory<>(producerConfigs());
    }

    @Bean
    KafkaTemplate<String, ProductCreatedEvent> kafkaTemplate() {
        return new KafkaTemplate<String, ProductCreatedEvent>(producerFactory());
    }

    @Bean
    NewTopic createTopic() {
        return TopicBuilder.name("product-created-events-topic")
            .partitions(3)
            .replicas(3)
            .configs(Map.of("min.insync.replicas", "2"))
            .build();
    }
}
```

Below are the configuration properties that the `producerConfigs()` method uses. These are defined as member variables at the top of the `KafkaConfig` class.

```
@Configuration
public class KafkaConfig {

    @Value("${spring.kafka.producer.bootstrap-servers}")
    private String bootstrapServers;

    @Value("${spring.kafka.producer.key-serializer}")
    private String keySerializer;

    @Value("${spring.kafka.producer.value-serializer}")
    private String valueSerializer;

    @Value("${spring.kafka.producer.acks}")
    private String acks;

    @Value("${spring.kafka.producer.properties.delivery.timeout.ms}")
    private String deliveryTimeout;

    @Value("${spring.kafka.producer.properties.linger.ms}")
    private String linger;

    @Value("${spring.kafka.producer.properties.request.timeout.ms}")
    private String requestTimeout;

    @Value("${spring.kafka.producer.properties.enable.idempotence}")
    private boolean idempotence;

    @Value("${spring.kafka.producer.properties.max.in.flight.requests.per.connection}")
    private Integer inflightRequests;

    ...

    // other methods below
}
```

These configuration properties are loaded from the *application.properties* file of this Spring Boot application.

```
server.port=0
spring.kafka.producer.bootstrap-servers=localhost:9092,localhost:9094
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=org.springframework.kafka.support.serializer.JsonSerializer
spring.kafka.producer.acks=all
spring.kafka.producer.properties.delivery.timeout.ms=120000
spring.kafka.producer.properties.linger.ms=0
spring.kafka.producer.properties.request.timeout.ms=30000
spring.kafka.producer.properties.enable.idempotence=true
spring.kafka.producer.properties.max.in.flight.requests.per.connection=5
```

# TESTING KAFKA PRODUCER

A Kafka Producer is a client or application that produces and sends messages to the Kafka server. These messages are stored in Kafka topics for real-time processing or batch consumption.

In this section of the book, we will write an integration for a method that acts as Kafka Producer. We want to make sure that this method successfully sends message to Kafka topic.

## CREATING A TEST CLASS

Let's start by creating a new test class in the `/src/test/<root package>` folder of our Spring Boot application.

If you downloaded source code that's is provided for this book, then the test class I am creating in this section is located in the folder: *ProductsMicroservice/src/test/java/com/appsdeveloperblog/ws/products*.

Where:

- *ProductsMicroservice* - Is a project name,
- *src/test* - is a folder in your Spring Boot application where you put test classes,
- */java/com/appsdeveloperblog/ws/products* - is a root package of this Spring Boot application.

I will create a new test class with a name: *ProductsServiceIntegrationTest*.

```
public class ProductsServiceIntegrationTest {  
}
```

Next, annotate this test class with the following annotations:

```
@DirtiesContext  
@EmbeddedKafka(partitions = 3, count = 3, controlledShutdown = true)  
@TestInstance(TestInstance.Lifecycle.PER_CLASS)  
@ActiveProfiles("test")  
@SpringBootTest(properties = "spring.kafka.producer.bootstrap-servers=$  
{spring.embedded.kafka.brokers}")  
public class ProductsServiceIntegrationTest {  
  
}
```



Let's discuss these annotations one by one.

## **@SpringBootTest**

The *@SpringBootTest* annotation is a must-have for our integration testing. This Spring Boot annotation is used to automatically start up an application context that includes all of your application's components (like controllers, repositories, services, etc.).

This allows you to test your application in a real-world-like environment, with all of its components interacting together as they would in a running application.

The *@SpringBootTest* annotation is typically used when you want to test the behavior of your application as a whole, or when you want to test the interaction between several components. Which is what we need when writing integration tests.

The property that is configured in *@SpringBootTest* annotation: *spring.kafka.producer.bootstrap-servers=\${spring.embedded.kafka.brokers}* is a configuration property that tells your application where to find the Kafka server.

The *\${spring.embedded.kafka.brokers}* part is a placeholder that gets replaced with the actual address of your embedded Kafka server (which is started by the *@EmbeddedKafka* annotation). This property is necessary because your application needs to know where to send the Kafka messages.

In a real-world application, this would typically be the address of your production Kafka server. But in a test environment, you want to use an embedded server so that you don't accidentally send test messages to your production server.

So, by setting *spring.kafka.producer.bootstrap-servers=\${spring.embedded.kafka.brokers}*, you're telling your application to send messages to your embedded Kafka server, which is exactly what you want in a test environment.

## **@ActiveProfiles("test")**

In Spring Boot, the *@ActiveProfiles* annotation is used to specify which Spring profile(s) should be active when running the tests. This annotation is not required unless you want to test your application with specific configuration properties that are related only to the test environment.

Spring profiles are a way to segregate parts of your application configuration and make it available only in certain environments like for example: development environment, test environment or production environment.

When you annotate a test class with `@ActiveProfiles("test")`, this will make Spring Boot look for a properties file named `application-test.properties` and load the configurations from that file. These configuration properties will be specifically for your testing environment.

The reason for this is to isolate your testing environment from your production environment. You typically don't want your tests to interact with your production resources (like databases). Instead, you'd use test-specific resources, which you can configure in your `application-test.properties` file.

### **`@TestInstance(TestInstance.Lifecycle.PER_CLASS)`**

The `@TestInstance(TestInstance.Lifecycle.PER_CLASS)` annotation is optional but can be very helpful, especially if your test class contains more than one test method.

By default, JUnit creates a new instance of each test class before executing each test method (this is `TestInstance.Lifecycle.PER_METHOD`). This means that if you have five methods in a test class, JUnit will create five instances of your test class, one for each test method.

However, when you use `@TestInstance(TestInstance.Lifecycle.PER_CLASS)`, JUnit creates only one instance of the test class and reuses it for all test methods in the class. This can be useful if you have expensive setup code in your `@BeforeAll` method that you'd prefer to run only once, rather than before each test method.

### **`@EmbeddedKafka(partitions = 3, count = 3, controlledShutdown = true)`**

The `@EmbeddedKafka` annotation is a must-have for the test method that we are going to write. This annotation is used in Spring Boot tests to set up an embedded Kafka server for testing purposes. It allows you to test your Kafka-related code against a real Kafka server, without the need to connect to an external Kafka server.

This is particularly useful for integration tests where you want to test the interaction of your application with Kafka.

Here's what each of the properties in the annotation means:

- **partitions = 3**: This sets the number of partitions for each topic in the embedded Kafka server. By setting this to 3, you're saying that each topic in your embedded Kafka server should have 3 partitions,
- **count = 3**: This sets the number of Kafka servers (also known as brokers) to start. Each broker can handle its own set of topics and partitions. By setting this to 3, you're starting up 3 brokers, which can be useful for testing scenarios involving multiple brokers,
- **controlledShutdown = true**: This makes sure that when the Kafka server is shut down, it will try to migrate all the leaders on the broker to other brokers before shutting down, to ensure a smooth transition and minimize disruption.

So, in summary, the *@EmbeddedKafka* annotation with these properties is setting up an embedded Kafka server with 3 brokers, where each topic has 3 partitions, and where the server will attempt a controlled shutdown when it's stopped. This provides a realistic environment for testing your Kafka-related code.

## **@DirtiesContext**

The *@DirtiesContext* annotation, while optional for the test method we are going to write, is a good-to-have in many other cases. This annotation is a part of the Spring Framework and is specifically used in testing. It indicates that the underlying Spring *ApplicationContext* has been “dirty” or modified during the execution of a test.

When we say a context is “dirty”, it means the test has modified or corrupted it in some manner. For example, by changing the state of a singleton bean. Once a context is marked as dirty, it is removed from the testing framework's cache and closed. As a consequence, the underlying Spring container is rebuilt for any subsequent test that requires a context with the same configuration metadata.

This isolation is particularly important when testing Kafka producers because the state of the producer (e.g., configuration properties, topic partitions, message serialization) can significantly impact the behavior of the producer and the messages it sends to Kafka topics.

So, when you use *@DirtiesContext*, you make sure that each test method tests the Kafka producer in a controlled and isolated environment.

At this moment, our test class is not yet ready to be executed. Let's add one basic test method to it just to see if the test class is working.

## CREATING NEW TEST METHOD

To verify if our test class is correctly configured to start up the embedded Kafka server, let's add one basic test method.

```
@Test
void testCreateProduct_whenGivenValidProductDetails_successfullySendsKafkaMessage()
{

}
```



```
ProductsServiceIntegrationTest.java
1 package com.appsdeveloperblog.ws.products;
2
3 import org.junit.jupiter.api.Test;
4 import org.junit.jupiter.api.TestInstance;
5 import org.springframework.boot.test.context.SpringBootTest;
6 import org.springframework.kafka.test.context.EmbeddedKafka;
7 import org.springframework.test.context.ActiveProfiles;
8
9 @EmbeddedKafka(partitions = 3, count = 3, controlledShutdown = true) new *
10 @TestInstance(TestInstance.Lifecycle.PER_CLASS)
11 @ActiveProfiles("test")
12 @SpringBootTest(properties = "spring.kafka.producer.bootstrap-servers=${spring.embedded.kafka.brokers}")
13 public class ProductsServiceIntegrationTest {
14
15     @Test new *
16     void testCreateProduct_whenGivenValidProductDetails_successfullySendsKafkaMessage() {
17     }
18 }
19
```

There are a couple of things to notice about this method: the `@Test` annotation and an unusually structured method name.

### The `@Test` annotation

The `@Test` annotation is used to indicate that the method below is a test method. This is a signal to JUnit that when it's running tests, it should execute this method as a test. Without the `@Test` annotation, JUnit would not know that this method is meant to be a test, and it would not be executed when you run your tests.

### Method naming convention

The name of the test method might seem strange at first, but it's actually following a common convention for naming test methods.

The convention is to describe what the test does in a way that's easy to understand just from reading the method name. This can make it easier to understand what's being tested and what the expected behavior is, especially when you have many tests in your test class.

It follows this pattern:

**test<System Under Test>\_Condition or State change\_ExpectedResult**

Here's how to interpret this test method name:

- **testCreateProduct**: This indicates that the test is for the `createProduct()` method,
- **whenGivenValidProductDetails**: This describes the conditions under which the test is run. In this case, it's when valid product details are given,
- **successfullySendsKafkaMessage**: This describes the expected outcome of the test. In this case, it's that a Kafka message is successfully sent.

So, put together, the test name is saying: "Test the `createProduct()` method, and when given valid product details, it should successfully send a Kafka message".

### Is it required?

If this method naming convention does not quite fit your or your team's style, then it's perfectly fine to adopt a different one. The key is to ensure that your test methods remain clear and understandable.

However, regardless of the naming convention you choose, always remember to include the `@Test` annotation. This is a non-negotiable aspect as it signals to the testing framework which methods are your test methods.

Now, if I execute this test method, it will pass by default. This is because if the test method does not fail, then it is considered to have successfully passed. And just in case you don't know how to execute a test method, in the following section, I will share with you a couple of ways you can run your test methods.

# WAYS TO RUN A TEST METHOD

## Using your Development Environment

The easiest way to run the test method is to use your development environment. Look for the “Play” icon on the left side of your test method.

If you don’t see it, your Java method might not be annotated with the `@Test` annotation, and it won’t be recognized as a test method.

To execute your test method, simply click the “Play” icon.

After successful execution, you’ll see a green check mark next to the play button. Otherwise, if there are issues, a red icon with an exclamation mark will appear.



```
ProductsServiceIntegrationTest.java x
1  package com.appsdeveloperblog.ws.products;
2
3  import org.junit.jupiter.api.Test;
4  import org.junit.jupiter.api.TestInstance;
5  import org.springframework.boot.test.context.SpringBootTest;
6  import org.springframework.kafka.test.context.EmbeddedKafka;
7  import org.springframework.test.context.ActiveProfiles;
8
9  @EmbeddedKafka(partitions = 3, count = 3, controlledShutdown = true) new *
10 @TestInstance(TestInstance.Lifecycle.PER_CLASS)
11 @ActiveProfiles("test")
12 @SpringBootTest(properties = "spring.kafka.producer.bootstrap-servers=${spring.embedded.kafka.brokers}")
13 >> public class ProductsServiceIntegrationTest {
14
15     @Test new *
16     > void testCreateProduct_whenGivenValidProductDetails_successfullySendsKafkaMessage() {
17
18     }
19 }
```

Execute all test methods in a class

Execute single test method



When you run this test method, check the console in your development environment. There will be a lot of messages. But some of the very first lines will look like these:

```
kafka.server.ControllerServer      : [ControllerServer id=0] Starting controller
kafka.server.ControllerServer      : [ControllerServer id=1] Starting controller
kafka.server.ControllerServer      : [ControllerServer id=2] Starting controller
kafka.server.BrokerServer          : [BrokerServer id=0] Transition from SHUTDOWN to STARTING
kafka.server.BrokerServer          : [BrokerServer id=1] Transition from SHUTDOWN to STARTING
kafka.server.SharedServer          : [SharedServer id=1] Starting SharedServer
kafka.server.SharedServer          : [SharedServer id=0] Starting SharedServer
kafka.server.BrokerServer          : [BrokerServer id=2] Transition from SHUTDOWN to STARTING
kafka.server.SharedServer          : [SharedServer id=2] Starting SharedServer
```

These log messages mean that the embedded Kafka environment is initializing as part of your test setup.

The `@EmbeddedKafka` annotation that we put above the class name is responsible for this process. It starts an in-memory Kafka broker and sets up the necessary Kafka servers, such as `ControllerServers` and `BrokerServers`.

This allows you to test your Kafka producer's functionality in a controlled environment that simulates a real Kafka cluster, without the need for an external Kafka server.

## Execute tests using Maven

You can also execute test methods using Maven commands. To do that, open the terminal window and change directory to the home folder of your Spring Boot project. If you list files in this folder you should see the `pom.xml` file.

To execute test methods, run the following command:

```
mvn test
```

This command will compile your Java code and execute all the test cases..

If you want to run a specific test class or method, you can use the `-Dtest` parameter with the `mvn test` command. For example, to execute all test methods in a class named `ProductsServiceIntegrationTest`, you would use:

```
mvn test -Dtest=ProductsServiceIntegrationTest
```

To execute a specific test method in the test class, you would use the following format:

```
mvn test <test class name>#<test method name>
```

For example:

```
mvn test
```

```
-Dtest=ProductsServiceIntegrationTest#testCreateProduct_whenGivenValidProductDetails_successfully  
SendsKafkaMessage
```

This will execute one specific test method in *ProductsServiceIntegrationTest* class and will print test report.

```
[INFO]  
[INFO] Results:  
[INFO]  
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0  
[INFO]  
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
[INFO] Total time: 20.684 s  
[INFO] Finished at: 2024-04-18T20:44:24-04:00  
[INFO] -----  
sergeykargopolov@Sergeys-MacBook-Pro ProductsMicroservice %
```



## TEST METHOD STRUCTURE. THE AAA PATTERN.

Now that we have a basic test method created, let's add some code to it. I will begin by creating a structure for this method, which involves organizing the code into three distinct sections: **Arrange**, **Act**, and **Assert**.

```
@Test
void testTransferMethod_whenGivenTransferDetails_successfullySendsKafkaMessage() {
    // Arrange

    // Act

    // Assert
}
```

This structure, known as the **Arrange-Act-Assert (AAA) pattern**, is a systematic approach to writing test methods that enhances their clarity and maintainability. Here's how each section functions within the test method:

- **Arrange:** This initial phase involves setting up the necessary conditions for the test. It includes initializing objects, configuring mocks or stubs, and preparing any input data.
- **Act:** In this phase, the actual method under test is executed. This is where you invoke the functionality you're testing with the arranged data.
- **Assert:** The final phase is where you verify that the act phase executed as expected. This is where you will put assertions.

### ARRANGE SECTION

In the arrange section, I will prepare an object that will be sent as as Kafka message payload.

```
// Arrange
String title = "iPhone 11";
BigDecimal price = new BigDecimal(600);
Integer quantity = 1;

CreateProductRestModel createProductRestModel = new CreateProductRestModel();
createProductRestModel.setTitle(title);
createProductRestModel.setQuantity(quantity);
createProductRestModel.setPrice(price);
```

The *CreateProductRestModel* is a custom class that I created in the production code of my Spring Boot application. It is an object of this class that the method under test expects.

## ACT SECTION

In the Act section, I will call the method under test.

```
productService.createProduct(createProductRestModel);
```

But for the *productService* to be available in my test method, I will need to inject it into my test class. So I will define a new member variable.

```
@Autowired
private ProductService productService;
```

Now my test method looks this way:

```
@Test
void testCreateProduct_whenGivenValidProductDetails_successfullySendsKafkaMessage()
throws Exception {
    // Arrange
    String title = "iPhone 11";
    BigDecimal price = new BigDecimal(600);
    Integer quantity = 1;

    CreateProductRestModel createProductRestModel = new CreateProductRestModel();
    createProductRestModel.setTitle(title);
    createProductRestModel.setQuantity(quantity);
    createProductRestModel.setPrice(price);

    // Act
    productService.createProduct(createProductRestModel);

    // Assert
}
```

Notice that I also declared that my test method ***throws Exception***. This is because the *createProduct()* method in the *ProductService* class can throw exception.

Now, when the *createProduct(createProductRestModel)* method is called, it will send a new Kafka message. I can now add test assertions to verify if the message is in Kafka topic.

## CONSUMER CONFIGURATION IN A TEST CLASS

To verify that my Kafka Producer successfully sent a Kafka message, I can check if there is a new message in the Kafka topic. To do this, I'll create a Kafka consumer, consume the new message, and confirm whether it matches the message I just sent. So let's do that.

I will put Kafka Consumer configuration to a private method in the same test class.

```
@Autowired
Environment environment;

@Autowired
private EmbeddedKafkaBroker embeddedKafkaBroker;

private Map<String, Object> getConsumerProperties() {
    return Map.of(
        ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
        embeddedKafkaBroker.getBrokersAsString(),
        ConsumerConfig.GROUP_ID_CONFIG,
        environment.getProperty("spring.kafka.consumer.group-id"),
        ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class,
        ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
        ErrorHandlingDeserializer.class,
        ErrorHandlingDeserializer.VALUE_DESERIALIZER_CLASS,
        JsonSerializer.class,
        JsonSerializer.TRUSTED_PACKAGES,

        environment.getProperty("spring.kafka.consumer.properties.spring.json.trusted.packages"),
        ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
        environment.getProperty("spring.kafka.consumer.auto-offset-reset")
    );
}
```

Notice that for this method to compile, I also needed to inject the *Environment* and *EmbeddedKafkaBroker* objects into the test class.

The **Environment** object is part of the Spring Framework and provides access to configuration properties and environment-related information. In this code, I am using Environment object to read properties defined in *application-test.properties* configuration file.

The *application-test.properties* file is in the */src/test/resources* folder of this Spring Boot project.

```
product-created-events-topic-name = product-created-events
spring.kafka.consumer.group-id=product-created-events
spring.kafka.consumer.properties.spring.json.trusted.packages=com.appsdeveloper
blog.ws.core
spring.kafka.consumer.auto-offset-reset=earliest
```

The **EmbeddedKafkaBroker** is a utility class provided by Spring Kafka for testing purposes. It allows me to set up an embedded Kafka broker (in-memory Kafka server) during testing, and interact with this embedded Kafka broker when testing Kafka-related functionality.

For example, I can use the *EmbeddedKafkaBroker* object to get the list of Kafka servers to connect to.

```
ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, embeddedKafkaBroker.getBrokersAsString()
```

You will use the *embeddedKafkaBroker.getBrokersAsString()* because it will help you connect to the correct brokers even when their port numbers change dynamically during testing.

When you start an embedded Kafka broker, it automatically assigns port numbers to its brokers. These dynamically assigned ports are not known in advance because they can change every time you run your tests. Therefore, you cannot hardcode specific broker addresses and port numbers in your test configuration.

Now that we have Kafka Consumer configuration, we can use it to listen to a specific topic. To do that I will setup Kafka Message Listener in a special method that will execute before the test method executes in this class.

```
private BlockingQueue<ConsumerRecord<String, ProductCreatedEvent>> records;
private KafkaMessageListenerContainer<String, ProductCreatedEvent> container;

@BeforeAll
void setUp() {
    DefaultKafkaConsumerFactory<String, Object> consumerFactory = new
DefaultKafkaConsumerFactory<>(
    getConsumerProperties());
    ContainerProperties containerProperties = new
ContainerProperties(environment.getProperty("product-created-events-topic-name"));
    container = new KafkaMessageListenerContainer<>(consumerFactory,
containerProperties);
    records = new LinkedBlockingQueue<>();
    container.setupMessageListener((MessageListener<String, ProductCreatedEvent>)
records::add);
    container.start();
    ContainerTestUtils.waitForAssignment(container,
embeddedKafkaBroker.getPartitionsPerTopic());
}

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



Let's break it down line by line.

1. Inside the `setUp()` method, I create a *DefaultKafkaConsumerFactory*. This factory is responsible for creating Kafka consumers with the specified properties (retrieved from `getConsumerProperties()`).
2. Then I create a *ContainerProperties* object, specifying the Kafka topic name for which the consumer will listen. The topic name is loaded from the *application-test.properties* file.
3. Then I instantiate a *KafkaMessageListenerContainer* using the consumer factory and container properties. The *KafkaMessageListenerContainer* serves as a bridge between your Spring application and the Kafka broker, and it helps you to consume messages from a topic using the provided Consumer configuration.
4. Then I create a *LinkedBlockingQueue* named `records` to store the consumed Kafka records.
5. Next I set up a message listener for the container using a lambda expression that adds each received record to the *records* queue.
6. And finally, I start the *KafkaMessageListenerContainer*, and the test waits for Kafka partitions to be assigned (using *ContainerTestUtils.waitForAssignment*).

Once *KafkaMessageListenerContainer* is started, it will begin consuming messages from the specified Kafka topic. It will invoke the message listener (the provided lambda expression) whenever a new message arrives. Whenever a new message arrives, it will be added to the *records* queue.

Now that we have *KafkaMessageListenerContainer* started, we can add assertions to see if it receives Kafka message sent by our Kafka Producer.

## ASSERT SECTION

To read Kafka message from the *records* queue I can use its *poll()* method.

```
ConsumerRecord<String, ProductCreatedEvent> message = records.poll(3000,  
TimeUnit.MILLISECONDS);
```

With this line of code I read a *ConsumerRecord* from the records queue, waiting up to 3 seconds for a record to become available.

If a record is available within that time, it will be retrieved and removed from the queue and stored in the *message* variable.

If after 3 seconds the record is not available in the queue, the poll operation will return *null*.

Now, I can add an assertion to check if the message is not null.

```
assertNotNull(message);
```

If the message object is *null*, then this assertion will throw exception and the test method will fail.

If you send Kafka messages as a key:value pair, then you can also check the if the message key is not null.

```
assertNotNull(message.key());
```

To read Java object that was originally sent as a message payload you can use the *value()* method of *ConsumerRecord* object.

```
ProductCreatedEvent productCreatedEvent = message.value();
```

And now, you can validate each property of this *productCreatedEvent* object, to make sure it is the same object that was sent.

```
assertEquals(productCreatedEvent.getQuantity(),  
createProductRestModel.getQuantity());  
assertEquals(productCreatedEvent.getTitle(), createProductRestModel.getTitle());  
assertEquals(productCreatedEvent.getPrice(), createProductRestModel.getPrice());
```

With these assertions added, my test method now appears as follows:

```
@Test
void testCreateProduct_whenGivenValidProductDetails_successfullySendsKafkaMessage()
throws Exception {
    // Arrange
    String title = "iPhone 11";
    BigDecimal price = new BigDecimal(600);
    Integer quantity = 1;

    CreateProductRestModel createProductRestModel = new CreateProductRestModel();
    createProductRestModel.setTitle(title);
    createProductRestModel.setQuantity(quantity);
    createProductRestModel.setPrice(price);

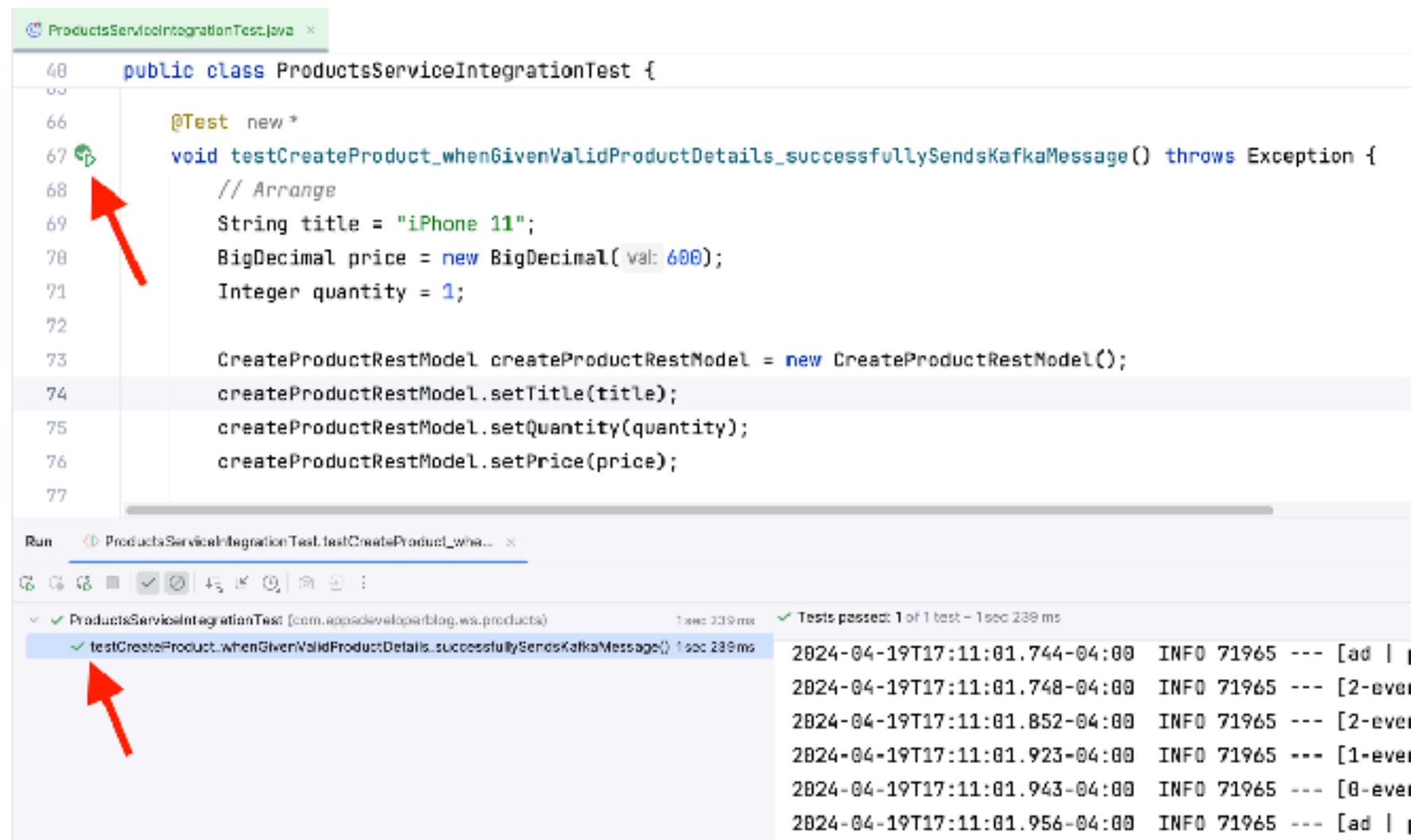
    // Act
    productService.createProduct(createProductRestModel);

    // Assert
    ConsumerRecord<String, ProductCreatedEvent> message = records.poll(3000,
TimeUnit.MILLISECONDS);
    assertNotNull(message);
    assertNotNull(message.key());
    ProductCreatedEvent productCreatedEvent = message.value();
    assertEquals(productCreatedEvent.getQuantity(),
createProductRestModel.getQuantity());
    assertEquals(productCreatedEvent.getTitle(),
createProductRestModel.getTitle());
    assertEquals(productCreatedEvent.getPrice(),
createProductRestModel.getPrice());
}
```

You can download the final version of [this class](#) from the [provided repository](#).

## RUNNING THE TEST METHOD

If you run this test method it should successfully pass.



The screenshot displays an IDE with two panels. The top panel shows the source code of `ProductsServiceIntegrationTest.java`. The bottom panel shows the test results for the method `testCreateProduct_whenGivenValidProductDetails_successfullySendsKafkaMessage()`.

**Source Code:**

```
48 public class ProductsServiceIntegrationTest {  
66     @Test new *  
67     void testCreateProduct_whenGivenValidProductDetails_successfullySendsKafkaMessage() throws Exception {  
68         // Arrange  
69         String title = "iPhone 11";  
70         BigDecimal price = new BigDecimal( val: 600);  
71         Integer quantity = 1;  
72  
73         CreateProductRestModel createProductRestModel = new CreateProductRestModel();  
74         createProductRestModel.setTitle(title);  
75         createProductRestModel.setQuantity(quantity);  
76         createProductRestModel.setPrice(price);  
77     }
```

**Test Results:**

Run ProductsServiceIntegrationTest.testCreateProduct\_whe... x

✓ ProductsServiceIntegrationTest [com.appdeveloperblog.ws.products] 1 sec 239 ms ✓ Tests passed: 1 of 1 test - 1 sec 239 ms

✓ testCreateProduct\_whenGivenValidProductDetails\_successfullySendsKafkaMessage() 1 sec 239 ms

2024-04-19T17:11:01.744-04:00 INFO 71965 --- [ad | |  
2024-04-19T17:11:01.748-04:00 INFO 71965 --- [2-ever  
2024-04-19T17:11:01.852-04:00 INFO 71965 --- [2-ever  
2024-04-19T17:11:01.923-04:00 INFO 71965 --- [1-ever  
2024-04-19T17:11:01.943-04:00 INFO 71965 --- [0-ever  
2024-04-19T17:11:01.956-04:00 INFO 71965 --- [ad | |

# VERIFY PRODUCER CONFIGURATION PROPERTIES

As developers, we often focus on writing and testing the business logic of our applications, but I think it is equally important to ensure that our Kafka producers are correctly configured.

The configuration of a Kafka producer can significantly impact how our application works. Therefore, I think, it is a good practice to write tests that verify producer configuration properties as well.

In this section of the book, we are going to focus on a specific part of the producer configuration which is making our producer idempotent.

## UNDERSTANDING IDEMPOTENT PRODUCERS IN APACHE KAFKA

In Apache Kafka, an idempotent producer is a type of producer that guarantees that messages are delivered exactly once, even in the event of retries or failures. This means that if a producer sends the same message multiple times, the message will only be written to the Kafka topic once, preventing duplicates.

When working with distributed systems like Kafka, it's essential to handle scenarios where messages might be sent multiple times due to network issues, broker failures, or other types of errors. Without idempotence, these retries could lead to duplicate messages being written to the topic, which can cause data inconsistencies and other issues in downstream consumers.

## CONFIGURING IDEMPOTENT PRODUCERS

The good news is that idempotency is enabled in the Apache Kafka producer by default since version 3.0. This is achieved with the following configuration property:

```
spring.kafka.producer.properties.enable.idempotence=true
```

However, there are other configuration properties that, if set to incorrect values, can make your Kafka producer non-idempotent.

To make you Kafka producer idempotent, you typically make sure the following properties are set with correct values:

```
spring.kafka.producer.properties.enable.idempotence=true  
spring.kafka.producer.acks=all  
spring.kafka.producer.properties.max.in.flight.requests.per.connection=5  
spring.kafka.producer.properties.retries=2147483647
```

Let's go through each of these properties and understand what they do:

- **enable.idempotence:** This property must be set to *true* to enable idempotent behavior in the producer. When set to *true*, the producer will ensure that messages are delivered exactly once by maintaining a sequence number for each message and deduplicating messages with the same sequence number.

The good news is that in newer Kafka versions, this configuration property is set to *true* by default. This means that you do not really need to enable it explicitly.

However, it is still recommended to explicitly set this property to *true* to prevent accidental disabling.

If this property is not explicitly set to *true*, you can accidentally disable idempotence in your Kafka Producer by setting other configuration properties to conflicting values.

- **spring.kafka.producer.acks:** This property controls the producer's acknowledgment behavior. For idempotent producers, it should be set to **all**. This means that the producer will wait for acknowledgments from all in-sync replicas before considering a message as successfully written. This ensures that the message is committed to all replicas, reducing the risk of data loss.
- **spring.kafka.producer.properties.max.in.flight.requests.per.connection:** The reason you need to set this configuration property to **5 or less** for idempotence to work is that Kafka uses a technique called sequence numbers to track the order of messages sent by the producer to each partition.

The broker assigns a sequence number to each message in the order it receives them and rejects any message that has a lower sequence number than the last one it processed. This way, the broker can detect and discard any duplicate messages that might have been resent due to retries.



However, this technique only works if the producer does not send more than 5 requests in parallel to the same partition. Otherwise, the broker might receive a message with a higher sequence number before a message with a lower sequence number and reject the latter as a duplicate. This would cause message loss and break the idempotence guarantee.

- **spring.kafka.producer.properties.retries:** This property specifies the number of retries the producer should attempt when an error occurs. For idempotent producers, this value must be greater than zero.

By default, the value of this property is set to the maximum integer value, which is **2147483647**. This default setting allows for numerous retry attempts in case of potentially transient errors.

In production app, I configured idempotent producer using the **@Bean** method.

```
@Configuration
public class KafkaConfig {

    @Value("${spring.kafka.producer.bootstrap-servers}")
    private String bootstrapServers;

    @Value("${spring.kafka.producer.key-serializer}")
    private String keySerializer;

    @Value("${spring.kafka.producer.value-serializer}")
    private String valueSerializer;

    @Value("${spring.kafka.producer.acks}")
    private String acks;

    @Value("${spring.kafka.producer.properties.enable.idempotence}")
    private boolean idempotence;

    @Value("${spring.kafka.producer.properties.max.in.flight.requests.per.connection}")
    private int inflightRequests;

    ...

    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, keySerializer);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, valueSerializer);

        ...

        // Idempotent Producer
        props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, idempotence);
        props.put(ProducerConfig.ACKS_CONFIG, acks);
        props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION,
inflightRequests);

        return props;
    }
}
```

And in the application.properties file, I have:

```
spring.kafka.producer.properties.enable.idempotence=true
spring.kafka.producer.acks=all
spring.kafka.producer.properties.max.in.flight.requests.per.connection=5
. . .
```

These configuration properties are then used in the same `KafkaConfig` class to create *KafkaProducerFactory* and *KafkaTemplate* objects.

```
@Bean
ProducerFactory<String, ProductCreatedEvent> producerFactory() {
    return new DefaultKafkaProducerFactory<>(producerConfigs());
}

@Bean
KafkaTemplate<String, ProductCreatedEvent> kafkaTemplate() {
    return new KafkaTemplate<String, ProductCreatedEvent>(producerFactory());
}
```

Take note that the *KafkaTemplate* object is created using the configuration defined in the *ProducerFactory*. This *KafkaTemplate* object is what we use to send messages to Kafka.

To verify if my Kafka producer is configured to be idempotent, I will inject this *KafkaTemplate* object into my test class. This allows me to access the producer configuration properties directly from the *KafkaTemplate* object.

By doing this, I can confirm that the *KafkaTemplate* object is indeed configured with the correct properties.

## TEST CLASS FOR IDEMPOTENT PRODUCER

To test that Kafka producer is configured to be idempotent, I will create a separate Java class.

```
@SpringBootTest
public class IdempotentProducerIntegrationTest {

}
```

The test method in this class is going to be very simple, so annotating it with *@SpringBootTest* annotation is enough.

## TEST METHOD FOR IDEMPOTENT PRODUCER

Next, I will create a new test method.

```
@SpringBootTest
public class IdempotentProducerIntegrationTest {

    @Test
    void testProducerConfig_whenIdempotenceEnabled_assertsIdempotentProperties() {

    }

}
```

As name of this test method suggests, I am going to test Producer configuration to make sure that it does have configuration properties that enable producer idempotence.

Let's start with the Arrange section.

```
@SpringBootTest
public class IdempotentProducerIntegrationTest {
    @Autowired
    KafkaTemplate<String, ProductCreatedEvent> kafkaTemplate;

    @Test
    void testProducerConfig_whenIdempotenceEnabled_assertsIdempotentProperties() {
        // Arrange
        ProducerFactory<String, ProductCreatedEvent> producerFactory =
kafkaTemplate.getProducerFactory();
    }
}
```

To get configuration properties that Kafka producer was configured with, I can inject into my test class *KafkaTemplate* object and use it to get *ProducerFactory*.

I can now use *ProducerFactory* object to get configuration properties that it uses to create Kafka producer. I can do it in the Act section of my test method.

```

@Test
void testProducerConfig_whenIdempotenceEnabled_assertsIdempotentProperties() {
    // Arrange
    ProducerFactory<String, ProductCreatedEvent> producerFactory =
kafkaTemplate.getProducerFactory();

    // Act
    Map<String, Object> config = producerFactory.getConfigurationProperties();
}

```

Now that I have access to configuration properties, I can verify those that make producer idempotent. I will do it in the Assert section of my test method.

```

@Test
void testProducerConfig_whenIdempotenceEnabled_assertsIdempotentProperties() {
    // Arrange
    ProducerFactory<String, ProductCreatedEvent> producerFactory =
kafkaTemplate.getProducerFactory();

    // Act
    Map<String, Object> config = producerFactory.getConfigurationProperties();

    // Assert
    Assertions.assertTrue((Boolean)
config.get(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG));
    Assertions.assertTrue("all".equalsIgnoreCase((String)
config.get(ProducerConfig.ACKS_CONFIG));
    if (config.containsKey(ProducerConfig.RETRIES_CONFIG)) {
        Assertions.assertTrue(
            Integer.parseInt(config.get(ProducerConfig.RETRIES_CONFIG).toString()) > 0
        );
    }
}

```

You can download the final version of [this class](#) from the [provided repository](#).

If you run this test method, it should successfully pass.

```
13 @SpringBootTest new *
14 public class IdempotentProducerIntegrationTest {
15     @Autowired 1 usage
16     KafkaTemplate<String, ProductCreatedEvent> kafkaTemplate;
17
18     @Test new *
19     void testProducerConfig_whenIdempotenceEnabled_assertsIdempotentProperties() {
20         // Arrange
21         ProducerFactory<String, ProductCreatedEvent> producerFactory = kafkaTemplate.getProducerFactory();
22
23         // Act
24         Map<String, Object> config = producerFactory.getConfigurationProperties();
25
26         // Assert
27         Assertions.assertTrue((Boolean) config.get(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG));
28         Assertions.assertTrue("all".equalsIgnoreCase((String) config.get(ProducerConfig.ACKS_CONFIG)));
29         if (config.containsKey(ProducerConfig.RETRIES_CONFIG)) {
30             Assertions.assertTrue(condition: Integer.parseInt(config.get(ProducerConfig.RETRIES_CONFIG).toString()) > 0)
31         }
32     }
33 }
```

# TESTING KAFKA CONSUMER

In this section of the book, we will write an integration test for a Spring Boot application that acts as a Kafka Consumer. But before we start writing the test code itself, let me quickly explain how the Kafka Consumer is configured in my Spring Boot Microservice.

You can download the source code for this application from the following GitHub repository: [EmailNotificationMicroservice](#).

## KAFKA CONSUMER APPLICATION CODE

To configure Kafka consumer I use *@Bean* method in a class annotated with *@Configuration* annotation.

```
@Configuration
public class KafkaConsumerConfiguration {

    @Autowired
    Environment environment;

    @Bean
    ConsumerFactory<String, Object> consumerFactory() {
        Map<String, Object> config = new HashMap<>();
        config.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
            environment.getProperty("spring.kafka.consumer.bootstrap-
servers"));
        config.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        config.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
ErrorHandlingDeserializer.class);
        config.put(ErrorHandlingDeserializer.VALUE_DESERIALIZER_CLASS,
JsonDeserializer.class);
        config.put(JsonDeserializer.TRUSTED_PACKAGES,

environment.getProperty("spring.kafka.consumer.properties.spring.json.trusted.packa
ges"));
        config.put(ConsumerConfig.GROUP_ID_CONFIG,
environment.getProperty("consumer.group-id"));
        config.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
environment.getProperty("spring.kafka.consumer.auto-offset-reset"));

        return new DefaultKafkaConsumerFactory<>(config);
    }

    . . .
```



Notice that I use *@Autowired* annotation to inject the *Environment* object to this class. I use this *Environment* object to load configuration properties from the application.properties file.

In the application.properties file, these configuration properties are configured as below:

```
spring.kafka.consumer.bootstrap-servers=localhost:9092,localhost:9094
consumer.group-id=product-created-events
spring.kafka.consumer.properties.spring.json.trusted.packages=com.appsdeveloperblog
.ws.core
spring.kafka.consumer.auto-offset-reset=earliest
```

But this is not all. My *KafkaConsumerConfiguration* class has a few other methods.

*@Bean*

```
ConcurrentKafkaListenerContainerFactory<String, Object>
kafkaListenerContainerFactory(
    ConsumerFactory<String, Object> consumerFactory, KafkaTemplate<String,
Object> kafkaTemplate) {
```

```
    DefaultErrorHandler errorHandler = new DefaultErrorHandler(new
DeadLetterPublishingRecoverer(kafkaTemplate),
        new FixedBackOff(5000,3));
    errorHandler.addNotRetryableExceptions(NotRetryableException.class);
    errorHandler.addRetryableExceptions(RetryableException.class);

    ConcurrentKafkaListenerContainerFactory<String, Object> factory = new
ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory);
    factory.setCommonErrorHandler(errorHandler);

    return factory;
}
```

*@Bean*

```
KafkaTemplate<String, Object> kafkaTemplate(ProducerFactory<String, Object>
producerFactory) {
    return new KafkaTemplate<>(producerFactory);
}
```

*@Bean*

```
ProducerFactory<String, Object> producerFactory() {
    Map<String, Object> config = new HashMap<>();
    config.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
environment.getProperty("spring.kafka.consumer.bootstrap-servers"));
    config.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);
    config.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);

    return new DefaultKafkaProducerFactory<>(config);
}
```

I will briefly summarize each of these methods, so that you know their purpose.

1. **kafkaListenerContainerFactory()**: This method is responsible for creating a *ConcurrentKafkaListenerContainerFactory* which is a factory for creating Kafka *MessageListenerContainer* instances. It sets the *ConsumerFactory* and *ErrorHandler*.

The *ConsumerFactory* is used to create Kafka Consumer instances, and the *ErrorHandler* is used to handle any exceptions thrown during the processing of records.

In this case, a *DefaultErrorHandler* is used with a *DeadLetterPublishingRecoverer* which republishes failed records to a dead-letter topic, and a *FixedBackOff* which specifies a fixed delay for retries.

The *addNotRetryableExceptions* and *addRetryableExceptions* methods are used to specify which exceptions should be retried and which should not.

2. **kafkaTemplate()**: This method creates a *KafkaTemplate* which is a template for executing high-level operations. It wraps a Producer instance and provides convenience methods for sending messages to Kafka topics.

The *KafkaTemplate* takes a *ProducerFactory* as a parameter, which is used to create Kafka Producer instances.

3. **producerFactory()**: The *producerFactory()* method is vital as it creates a Kafka *ProducerFactory*. This factory is used to instantiate a *KafkaTemplate*, which is responsible for sending messages to Kafka topics. In the context of error handling, this *KafkaTemplate* is used by the *DeadLetterPublishingRecoverer*.

The *DeadLetterPublishingRecoverer* is part of the error handling strategy. When a message fails to process, instead of losing it, the recoverer sends it to a “dead-letter topic”. This allows you to isolate and investigate failed messages.

Therefore, the *producerFactory()* method indirectly supports this error handling approach by providing the necessary *KafkaTemplate*.

## METHOD UNDER TEST

Next, let's have a look at the method that actually handles received Kafka messages. I intentionally made this method simpler for demonstration purposes.

It will be this method that I will write integration test for.

```
@Component
@KafkaListener(topics = "product-created-events-topic")
public class ProductCreatedEventHandler {

    private final Logger LOGGER = LoggerFactory.getLogger(this.getClass());

    @KafkaHandler
    public void handle(@Payload ProductCreatedEvent productCreatedEvent,
@Header("messageId") String messageId,
        @Header(KafkaHeaders.RECEIVED_KEY) String messageKey) {
        LOGGER.info("Received a new event: " + productCreatedEvent.getTitle());
    }
}
```

The *ProductCreatedEventHandler* class is a Kafka consumer in my Spring Boot application. It listens to the *product-created-events-topic* topic, as specified by the *@KafkaListener* annotation.

The *handle* method is marked with the *@KafkaHandler* annotation, indicating that it's a message handler for the consumer. It processes incoming *ProductCreatedEvent* messages, which are passed as the message payload.

The method also receives two headers: *messageId* and *messageKey*. The *messageId* is a custom header, while *messageKey* is a standard Kafka header, accessed via *KafkaHeaders.RECEIVED\_KEY*.

In the following chapter of this book, we will write a test method that will test the behavior of this Kafka consumer. Specifically, we make sure that when a new message is sent to the *product-created-events-topic*, the *handle* method in the *ProductCreatedEventHandler* class is called.

We'll also verify that it receives all the required method arguments correctly. This includes the *ProductCreatedEvent* object and the headers parameters. This way, we can be confident that our Kafka consumer is working as intended, processing incoming messages and their associated headers accurately.

## CREATING A TEST CLASS

Let's start by creating a new test class in the `/src/test/<root package>` folder of our Spring Boot application.

If you downloaded source code that's is provided for this book, then the test class I am creating in this section is located in the folder: *EmailNotificationMicroservice/src/test/java/com/appsdeveloperblog/ws/emailnotification*.

Where:

- *EmailNotificationMicroservice* - Is a project name,
- *src/test* - is a folder in your Spring Boot application where you put test classes,
- */java/com/appsdeveloperblog/ws/emailnotification* - is a root package of this Spring Boot application.

I will create a new test class with a name: *ProductsCreatedEventHandlerIntegrationTest*.

```
public class ProductsCreatedEventHandlerIntegrationTest {  
  
}
```

Next, annotate this test class with the following annotations:

```
@DirtiesContext  
@EmbeddedKafka  
@SpringBootTest(properties = "spring.kafka.consumer.bootstrap-servers=$  
{spring.embedded.kafka.brokers}")  
@TestInstance(TestInstance.Lifecycle.PER_CLASS)  
@ActiveProfiles({"test"})  
public class ProductsCreatedEventHandlerIntegrationTest {  
  
}
```

You have seen all of these annotations in an earlier in the test class for Kafka producer. Except one little difference. The `@SpringBootTest` annotation is configured with the *spring.kafka.consumer.bootstrap-servers* property, instead of *spring.kafka.producer.bootstrap-servers* property.

## CREATING A TEST METHOD

To verify that the Kafka handler method in my Spring Boot Microservice can successfully consume a Kafka message, I will create a new test method.

```
@Test
public void testProductCreatedEventHandler
    _WhenProductCreatedEventIsPublished_HandleIsInvoked() throws ExecutionException,
    InterruptedException {
    // Arrange

    // Act

    // Assert
}
```

In the arrange section, I will first prepare the *ProductCreatedEvent* object, which will be sent as a message payload.

```
// Arrange
ProductCreatedEvent productCreatedEvent = new ProductCreatedEvent();
productCreatedEvent.setPrice(new BigDecimal(10));
productCreatedEvent.setProductId(UUID.randomUUID().toString());
productCreatedEvent.setQuantity(1);
productCreatedEvent.setTitle("Test product");
```

Then, I will prepare the *messageId* and *messageKey* that will be sent as message headers.

```
String messageId = UUID.randomUUID().toString();
String messageKey = productCreatedEvent.getProductId();
```

And finally, I will create *ProducerRecord* which will be sent as Kafka message.

```
ProducerRecord<String, ProductCreatedEvent> record = new ProducerRecord<>(
    "product-created-events-topic",
    messageKey,
    productCreatedEvent);

record.headers().add("messageId", messageId.getBytes());
record.headers().add(KafkaHeaders.RECEIVED_KEY, messageKey.getBytes());
```

This will conclude the Arrange section.

In the Act section of this test method, I will use *KafkaTemplate* to send this *ProducerRecord* object.

```
@DirtiesContext
@EmbeddedKafka
@SpringBootTest(properties = "spring.kafka.consumer.bootstrap-servers=${spring.embedded.kafka.brokers}")
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
@ActiveProfiles({"test"})
public class ProductsCreatedEventHandlerIntegrationTest {

    @Autowired
    KafkaTemplate<String, ProductCreatedEvent> kafkaTemplate;

    @Test
    public void testProductCreatedEventHandler
    _WhenProductCreatedEventIsPublished_HandleIsInvoked() throws Exception
    {

        // Arrange
        ProductCreatedEvent productCreatedEvent = new ProductCreatedEvent();
        productCreatedEvent.setPrice(new BigDecimal(10));
        productCreatedEvent.setProductId(UUID.randomUUID().toString());
        productCreatedEvent.setQuantity(1);
        productCreatedEvent.setTitle("Test product");

        String messageId = UUID.randomUUID().toString();
        String messageKey = productCreatedEvent.getProductId();

        ProducerRecord<String, ProductCreatedEvent> record = new ProducerRecord<>(
            "product-created-events-topic",
            messageKey,
            productCreatedEvent);

        record.headers().add("messageId", messageId.getBytes());
        record.headers().add(KafkaHeaders.RECEIVED_KEY, messageKey.getBytes());

        // Act
        kafkaTemplate.send(record).get();

    }
}
```

But for this code to compile, I will need to inject *KafkaTemplate* object to my test class.



In the Assert section I will use *ArgumentCaptor* class to verify that the *handle()* method was called with correct arguments.

*ArgumentCaptor* is a class in the Mockito framework used to capture argument values for further assertions. It is used in the verification phase of testing, where you want to check if a method was called with certain arguments.

I will first define the expected arguments.

```
ArgumentCaptor<String> messageIdCaptor = ArgumentCaptor.forClass(String.class);
ArgumentCaptor<String> messageKeyCaptor = ArgumentCaptor.forClass(String.class);
ArgumentCaptor<ProductCreatedEvent> eventCaptor =
ArgumentCaptor.forClass(ProductCreatedEvent.class);
```

Next, I will inject the class under test as a *SpyBean* to my test class.

The *@SpyBean* annotation that I used in this code is used for mocking a Spring bean while still retaining some of its original behavior.

```
@DirtiesContext
@EmbeddedKafka
@SpringBootTest(properties = "spring.kafka.consumer.bootstrap-servers=${spring.embedded.kafka.brokers}")
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
@ActiveProfiles({"test"})
public class ProductsCreatedEventHandlerIntegrationTest {

    . . .

    @SpyBean
    ProductCreatedEventHandler productCreatedEventHandler;

    . . .
}
```

When you annotate a field with the *@SpyBean* annotation, the Spring framework creates a partial mock of the specified bean. This allows you to selectively override or verify certain methods.

The reason I annotated the *productCreatedEventHandler* member variable with the *@SpyBean* annotation is because I want to maintain the original behavior of the *ProductCreatedEventHandler* object while still being able to intercept method calls and perform additional verifications.



For instance, when the `handle()` method is invoked, I aim to capture method arguments and verify their values.

So now that I created a Spy for the *ProductCreatedEventHandler* object, I can use it to verify if the `handle()` method in this class was invoked with the needed arguments.

```
verify(productCreatedEventHandler, timeout(5000).times(1))  
    .handle(eventCaptor.capture(), messageIdCaptor.capture(),  
messageKeyCaptor.capture());
```

Finally, I can add assertions to verify that the received arguments contain expected values.

```
assertEquals(productCreatedEvent.getProductId(),  
eventCaptor.getValue().getProductId());  
assertEquals(messageId, messageIdCaptor.getValue());  
assertEquals(messageKey, messageKeyCaptor.getValue());
```

My complete test method now looks as presented below. You can download the final version of [this class](#) from the [provided repository](#).

```

@Test
public void
testProductCreatedEventHandler_WhenProductCreatedEventIsPublished_HandleIsInvoked()
    throws ExecutionException, InterruptedException {

    // Arrange
    ProductCreatedEvent productCreatedEvent = new ProductCreatedEvent();
    productCreatedEvent.setPrice(new BigDecimal(10));
    productCreatedEvent.setProductId(UUID.randomUUID().toString());
    productCreatedEvent.setQuantity(1);
    productCreatedEvent.setTitle("Test product");

    String messageId = UUID.randomUUID().toString();
    String messageKey = productCreatedEvent.getProductId();

    ProducerRecord<String, ProductCreatedEvent> record = new ProducerRecord<>(
        "product-created-events-topic",
        messageKey,
        productCreatedEvent);

    record.headers().add("messageId", messageId.getBytes());
    record.headers().add(KafkaHeaders.RECEIVED_KEY, messageKey.getBytes());

    // Act
    kafkaTemplate.send(record).get();

    // Assert
    ArgumentCaptor<String> messageIdCaptor = ArgumentCaptor.forClass(String.class);
    ArgumentCaptor<String> messageKeyCaptor =
    ArgumentCaptor.forClass(String.class);
    ArgumentCaptor<ProductCreatedEvent> eventCaptor =
    ArgumentCaptor.forClass(ProductCreatedEvent.class);

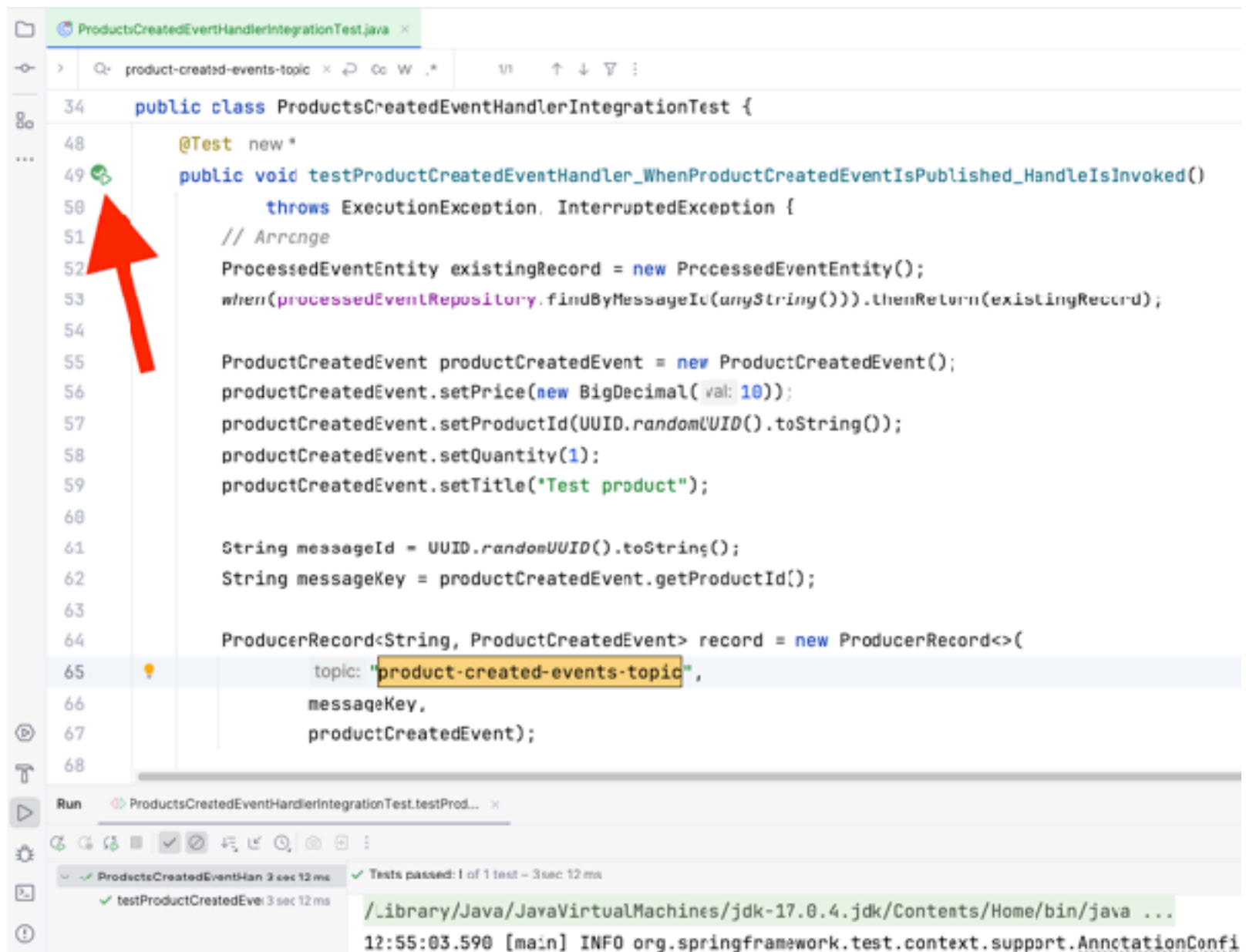
    verify(productCreatedEventHandler, timeout(5000).times(1))
        .handle(eventCaptor.capture(), messageIdCaptor.capture(),
    messageKeyCaptor.capture());

    assertEquals(productCreatedEvent.getProductId(),
    eventCaptor.getValue().getProductId());
    assertEquals(messageId, messageIdCaptor.getValue());
    assertEquals(messageKey, messageKeyCaptor.getValue());
}

```

## RUNNING THE TEST METHOD

If you run this test method it should successfully pass.



The screenshot shows an IDE with a Java file named `ProductsCreatedEventHandlerIntegrationTest.java`. The code defines a public class with a single test method. A red arrow points to the test method signature. Below the code, the 'Run' tab shows the test execution results, indicating that the test passed successfully. The output console shows the Java command used to run the test and a log message from the Spring Framework.

```
34 public class ProductsCreatedEventHandlerIntegrationTest {
48     @Test new *
49     public void testProductCreatedEventHandler_WhenProductCreatedEventIsPublished_HandleIsInvoked()
50         throws ExecutionException, InterruptedException {
51         // Arrange
52         ProcessedEventEntity existingRecord = new ProcessedEventEntity();
53         when(processedEventRepository.findByMessageId(anyString())).thenReturn(existingRecord);
54
55         ProductCreatedEvent productCreatedEvent = new ProductCreatedEvent();
56         productCreatedEvent.setPrice(new BigDecimal("10"));
57         productCreatedEvent.setProductId(UUID.randomUUID().toString());
58         productCreatedEvent.setQuantity(1);
59         productCreatedEvent.setTitle("Test product");
60
61         String messageId = UUID.randomUUID().toString();
62         String messageKey = productCreatedEvent.getProductId();
63
64         ProducerRecord<String, ProductCreatedEvent> record = new ProducerRecord<>(
65             topic: "product-created-events-topic",
66             messageKey,
67             productCreatedEvent);
68 }
```

Run ProductsCreatedEventHandlerIntegrationTest.testProd... x

ProductsCreatedEventHan 3 sec 12 ms Tests passed: 1 of 1 test - 3 sec 12 ms

testProductCreatedEver 3 sec 12 ms

/Library/Java/JavaVirtualMachines/jdk-17.0.4.jdk/Contents/Home/bin/java ...

12:55:03.590 [main] INFO org.springframework.test.context.support.AnnotationConfi

# THANK YOU AND NEXT STEPS

I hope this little ebook was helpful to you, and now you know how to:

- Write integration tests for Kafka Producer to verify it can send messages correctly to Kafka Topics,
- Configure and verify configuration properties for an Idempotent Kafka producer,
- Write integration tests for Kafka Consumers to validate they are receiving correct messages from Kafka topic.

There's much more to learn about Apache Kafka and testing Java code in general. Therefore, I invite you to check out two video courses that I've prepared. Let me briefly explain each of them below.

## Testing Java with JUnit 5 and Mockito



This video course is for beginners; you do not need prior Unit testing knowledge to enrol in this course. The course covers JUnit 5 basics as well as advanced topics.

You will also learn to use another very popular testing framework for Java called Mockito. By the end of this course, you will be able to test even very complex Java code.

## Apache Kafka for Spring Boot Microservices



In this video course, you will learn how to use Apache Kafka to build Event-Driven Spring Boot Microservices.

This course is designed for beginners and will start from the basics of Apache Kafka and Spring Boot Microservices.

If you take this video course, you will learn how to:

- Work with Apache Kafka CLI,
- Run cluster of Apache Kafka servers in Docker containers,
- Implement Idempotent Kafka Producer and Consumer, along with handling retries, errors, and transactions.