How to Work with Apache Kafka in Your Spring Boot Application

NOVEMBER 1, 2018   LAST UPDATED: NOVEMBER 7, 2019

Choosing the right messaging system during your architectural planning is always a challenge, yet one of the most important considerations to nail. As a developer, I write applications daily that need to serve lots of users and process huge amounts of data in real time.

Usually, I use Java with the Spring Framework (Spring Boot, Spring Data, Spring Cloud, Spring Caching, etc.) for this. Spring Boot is a framework that allows me to go through my development process much faster and easier than before. It has come to play a crucial role in my organization. As the number of our users quickly grew, we realized our apparent need for something that could process as many as 1,000,000 events per second.

When we found Apache Kafka®, we saw that it met our needs and could handle millions of messages quickly. That's why we decided to try it. And since that moment, Kafka has been a vital tool in my pocket. Why did I choose it, you ask?

Apache Kafka is:

- Scalable
- Fault tolerant
- A great publish-subscribe messaging system
- Capable of higher throughput compared with most messaging systems

- Highly durable
- Highly reliable
- High performant

That's why I decided to use it in my projects. Based on my experience, I provide here a step-by-step guide on how to include Apache Kafka in your Spring Boot application so that you can start leveraging its benefits too.

Prerequisites

- This article requires you to have Confluent Platform
- Manual install using ZIP and TAR archives
  - Download
  - Unzip it
  - Follow the step-by-step instructions, and you'll get Kafka up and running in your local environment

I recommend using the Confluent CLI for your development to have Apache Kafka and other components of a streaming platform up and running.

What you'll get out of this guide

After reading this guide, you will have a Spring Boot application with a Kafka producer to publish messages to your Kafka topic, as well as with a Kafka consumer to read those messages.

And with that, let's get started!

Table of contents

Step 1: Generate our project

First, let's go to Spring Initializr to generate our project. Our project will have Spring MVC/web support and Apache Kafka support.

Once you have unzipped the project, you'll have a very simple structure. I'll show you how the project will look like at the end of this article so you can easily follow the same structure. I'm going to use Intellij IDEA, but you can use any Java IDE.

Step 2: Publish/read messages from the Kafka topic
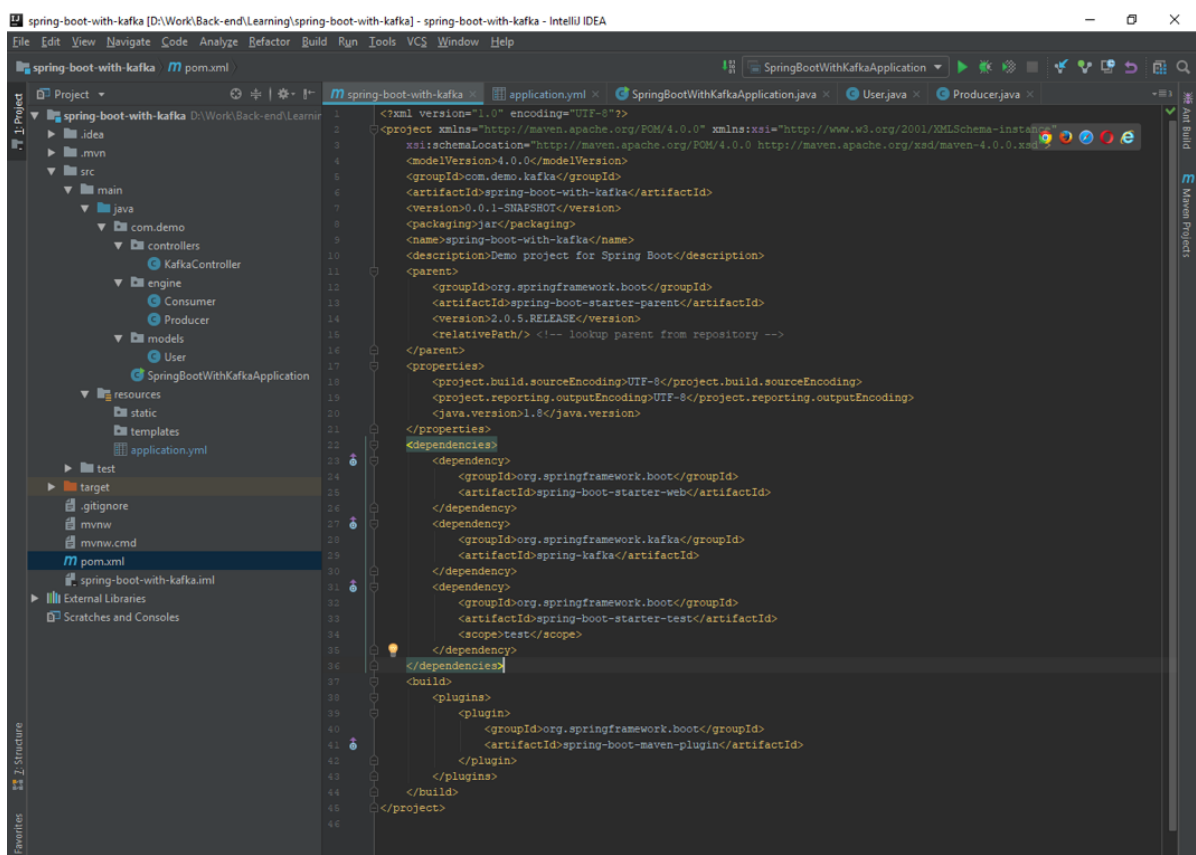
Now, you can see what it looks like. Let's move on to publishing/reading messages from the Kafka topic.
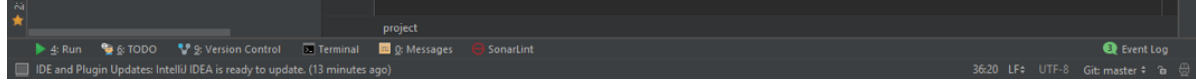
Start by creating a simple Java class, which we will use for our example: `package com.demo.models;`

```java
public class User {

    private String name;
    private int age;

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

project

▶ ⬧ Run   ☰ 6: TODO   ⑂ 9: Version Control   ⊡ Terminal   ☰ 0: Messages   ● SonarLint   ⦿ Event Log

☐ IDE and Plugin Updates: IntelliJ IDEA is ready to update. (13 minutes ago)   36:20   LF: UTF-8   Git: master ⬩   ⬩ ⬗

Step 3: Configure Kafka through `application.yml` configuration file

Next, we need to create the configuration file. We need to somehow configure our Kafka producer and consumer to be able to publish and read messages to and from the topic. Instead of creating a Java class, marking it with `@Configuration` annotation, we can use either

`application.properties` file or `application.yml`. Spring Boot allows us to avoid all the boilerplate code we used to write in the past, and provide us with much more intelligent way of configuring our application, like this:

```
server: port: 9000
spring:
   kafka:
      consumer:
         bootstrap-servers: localhost:9092
         group-id: group_id
         auto-offset-reset: earliest
         key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
         value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      producer:
         bootstrap-servers: localhost:9092
         key-serializer: org.apache.kafka.common.serialization.StringSerializer
         value-serializer: org.apache.kafka.common.serialization.StringSerializer
```

If you want to get more about Spring Boot auto-configuration, you can read this short and useful article. For a full list of available configuration properties, you can refer to the official documentation.

Step 4: Create a producer

Creating a producer will write our messages to the topic.

```
@Service
public class Producer {

    private static final Logger logger = LoggerFactory.getLogger(Producer.class);
    private static final String TOPIC = "users";

    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    public void sendMessage(String message) {
        logger.info(String.format("#### -> Producing message -> %s", message));
        this.kafkaTemplate.send(TOPIC, message);
    }
}
```

We just auto-wired `KafkaTemplate` and will use this instance to publish messages to the topic— that's it for producer!

Step 5: Create a consumer

Consumer is the service that will be responsible for reading messages processing them according to the needs of your own business logic. To set it up, enter the following:

```java
@Service
public class Consumer {

    private final Logger logger = LoggerFactory.getLogger(Producer.class);

    @KafkaListener(topics = "users", groupId = "group_id")
    public void consume(String message) throws IOException {
        logger.info(String.format("#### -> Consumed message -> %s", message));
    }
}
```

Here, we told our method `void consume` (String message) to subscribe to the user's topic and just emit every message to the application log. In your real application, you can handle messages the way your business requires you to.

Step 6: Create a REST controller

If we already have a consumer, then we already have all we need to be able to consume Kafka messages.

To fully show how everything that we created works, we need to create a controller with single endpoint. The message will be published to this endpoint, and then handled by our producer.

Then, our consumer will catch and handle it the way we set it up by logging to the console.

```java
@RestController
@RequestMapping(value = "/kafka")
public class KafkaController {

    private final Producer producer;

    @Autowired
    KafkaController(Producer producer) {
        this.producer = producer;
    }

    @PostMapping(value = "/publish")
    public void sendMessageToKafkaTopic(@RequestParam("message") String message) {
        this.producer.sendMessage(message);
    }
}
```
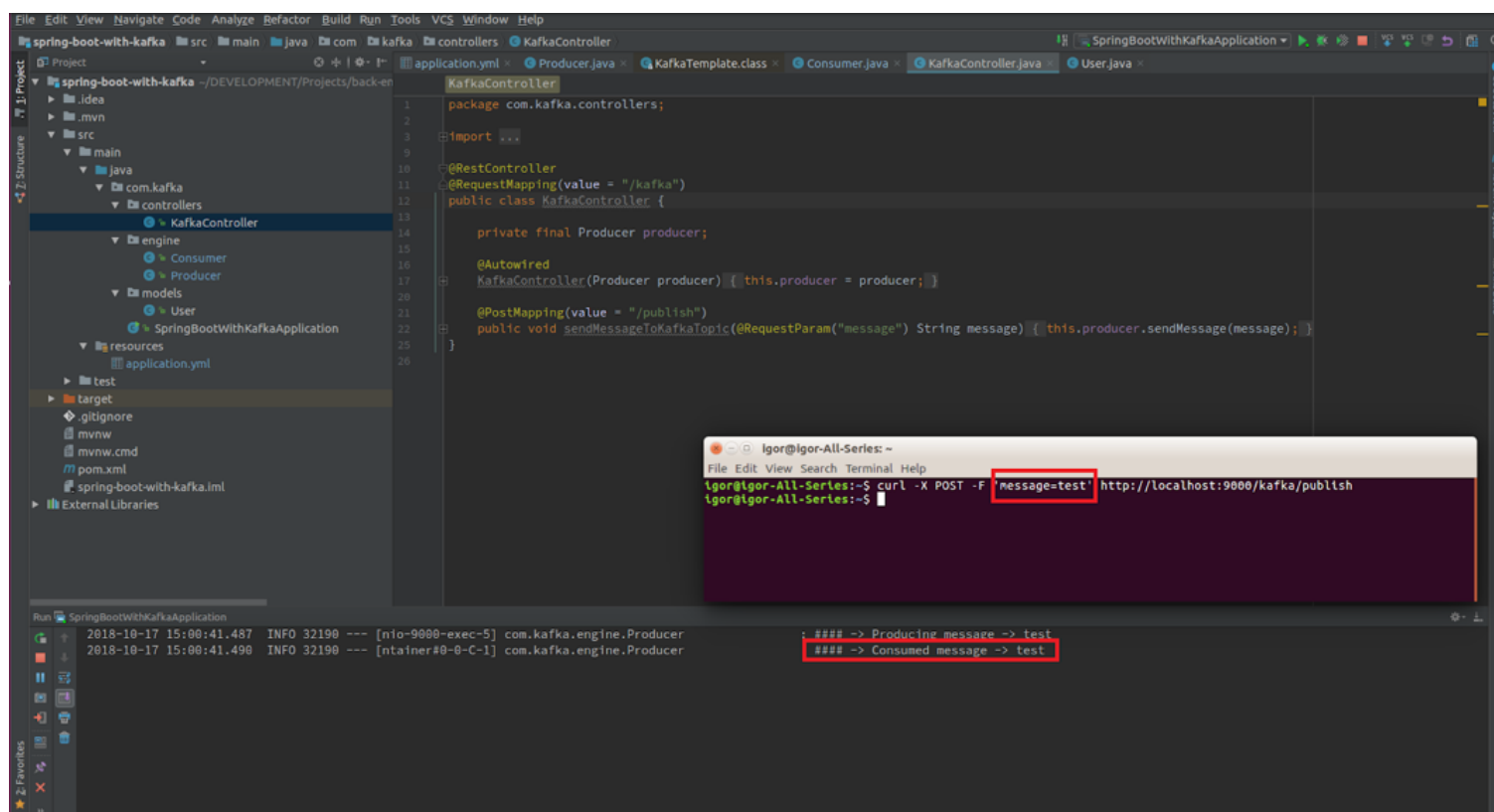
Let's send our message to Kafka using cURL:

```
curl -X POST -F 'message=test' http://localhost:9000/kafka/publish
```


Basically, that's it! In fewer than 10 steps, you learned how easy it is to add Apache Kafka to your Spring Boot project. If you followed this guide, you now know how to integrate Kafka into your Spring Boot project, and you are ready to go with this super tool!

Interested in more?

If you'd like to know more, you can download the Confluent Platform, the leading distribution of Apache Kafka. You can also find all the code in this article on GitHub.

Further reading

- Spring for Apache Kafka Deep Dive – Part 1: Error Handling, Message Conversion and Transaction Support
- Spring for Apache Kafka Deep Dive – Part 2: Apache Kafka and Spring Cloud Stream
- Spring for Apache Kafka Deep Dive – Part 3: Apache Kafka and Spring Cloud Data Flow
- Spring for Apache Kafka Deep Dive – Part 4: Continuous Delivery of Event Streaming Pipelines

This is a guest post by Igor Kosandyak, a Java software engineer at Oril, with extensive experience in various development areas.