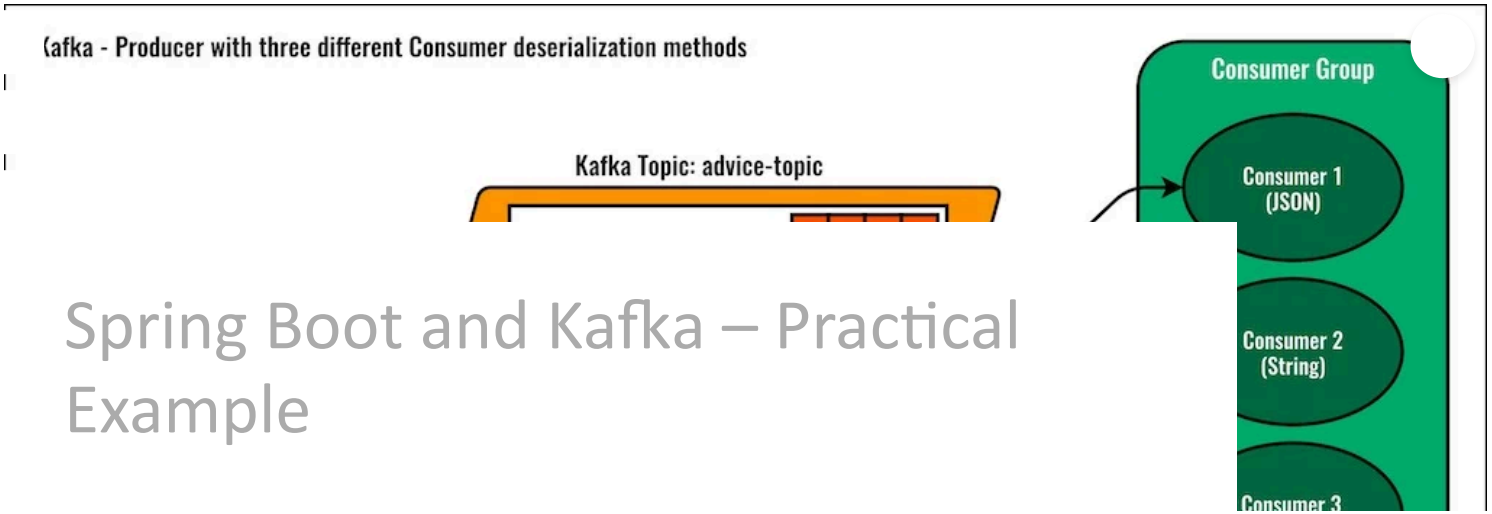
THE PRACTICAL DEVELOPER

SOFTWARE DEVELOPMENT IS EASY WHEN YOU UNDERSTAND WHAT YOU'RE DOING



Spring Boot and Kafka – Practical Example

BY MOISÉS MACERO ON OCTOBER 8, 2021

This blog post shows you how to configure Spring Kafka and Spring Boot to send messages using JSON and receive them in multiple formats: JSON, plain Strings or byte arrays.

— ABOUT MOISÉS MACERO

Software Developer, Architect, and Author.

Are you interested in my workshops?

This sample application also demonstrates how to use multiple Kafka consumers within the same consumer group with the **@KafkaListener** annotation, so the messages are load-balanced. Each consumer implements a different deserialization approach.

Besides, at the end of this post, you will find some practical exercises in case you want to grasp some Kafka concepts like the Consumer Group and Topic partitions.

Table of Contents

— LATEST POSTS

Book's Upgrade: Migrating from Spring Boot 2.6 to 2.7
AUGUST 5, 2022

Book's Upgrade: Migrating from Spring Boot 2.5 to 2.6
JANUARY 21, 2022

Book's Upgrade: Migrating from Spring Boot 2.4 to 2.5
OCTOBER 15, 2021

How to test a controller in Spring Boot - a practical guide
OCTOBER 9, 2021

Spring Boot and Kafka – Practical Example
OCTOBER 8, 2021

— TAGS

JSON REST AGILE

ANGULAR ARCHITECTURE

ASYNC BDD BOOK

- Try some Kafka Exercises
  - Request /hello multiple times
  - Reduce the number of partitions
  - Change one Consumer's Group Identifier

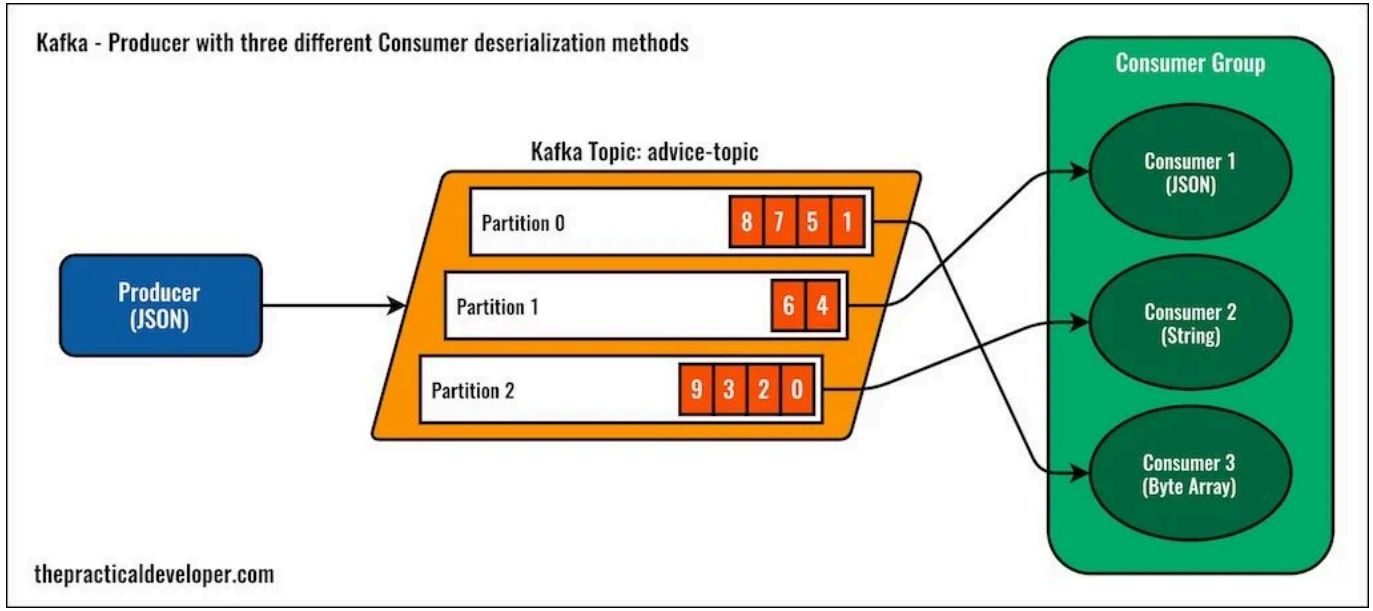
Updated (Oct 8th, 2021): This post uses now Spring Boot 2.5 and Java 17. If you're curious about the changes check the Pull Request with the changes.

Multiple consumers in a consumer group

Logical View

To better understand the configuration, have a look at the diagram below. As you can see, we create a Kafka topic with three partitions. On the consumer side, there is only one application, but it implements three Kafka consumers with the same `group.id` property. This is the configuration needed for having them in the same Kafka Consumer Group.

BOOK-2ND CONFERENCES CUCUMBER DOCKER EUREKA FEATURE-TOGGLE GAMIFICATION HYSTRIX JAVA JBOSS JUNIT KAFKA MICROSERVICES MONGODB RABBITMQ REACTIVE RIBBON SPRING-BOOT SWAGGER TEST WEBFLUX WILDFLY ZUUL



Kafka deserialization examples

When we start the application, Kafka assigns each consumer a different partition. This consumer group will receive the messages in a load-balanced manner. Later in this post, you'll see what is the difference if we make them have different group identifiers (you probably know the result if you are familiar with Kafka).

The Example Use Case

The logic we are going to build is simple. Each time we call a given REST endpoint, `hello`, the app will produce a configurable number of messages and send them to the same topic, using a sequence number as the Kafka key. It will wait (using a `CountDownLatch`) for all messages to be consumed before returning a message, `Hello Kafka!`. There will be three consumers,

each using a different deserialization mechanism, that will decrement the latch count when they receive a new message.

Easy, right? Let's see how to build it.

Setting up Spring Boot and Kafka

All the code in this post is available on GitHub: Kafka and Spring Boot Example. If you find it useful, please give it a star!

Starting up Kafka

First, you need to have a running Kafka cluster to connect to. For this application, I will use docker-compose and Kafka running in a single node. This is clearly far from being a production configuration, but it is good enough for the goal of this post.

docker-compose.yml

```
version: '2'
services:
  zookeeper:
    image: wurstmeister/zookeeper
    ports:
    - "2181:2181"
  kafka:
    image: wurstmeister/kafka
    ports:
```

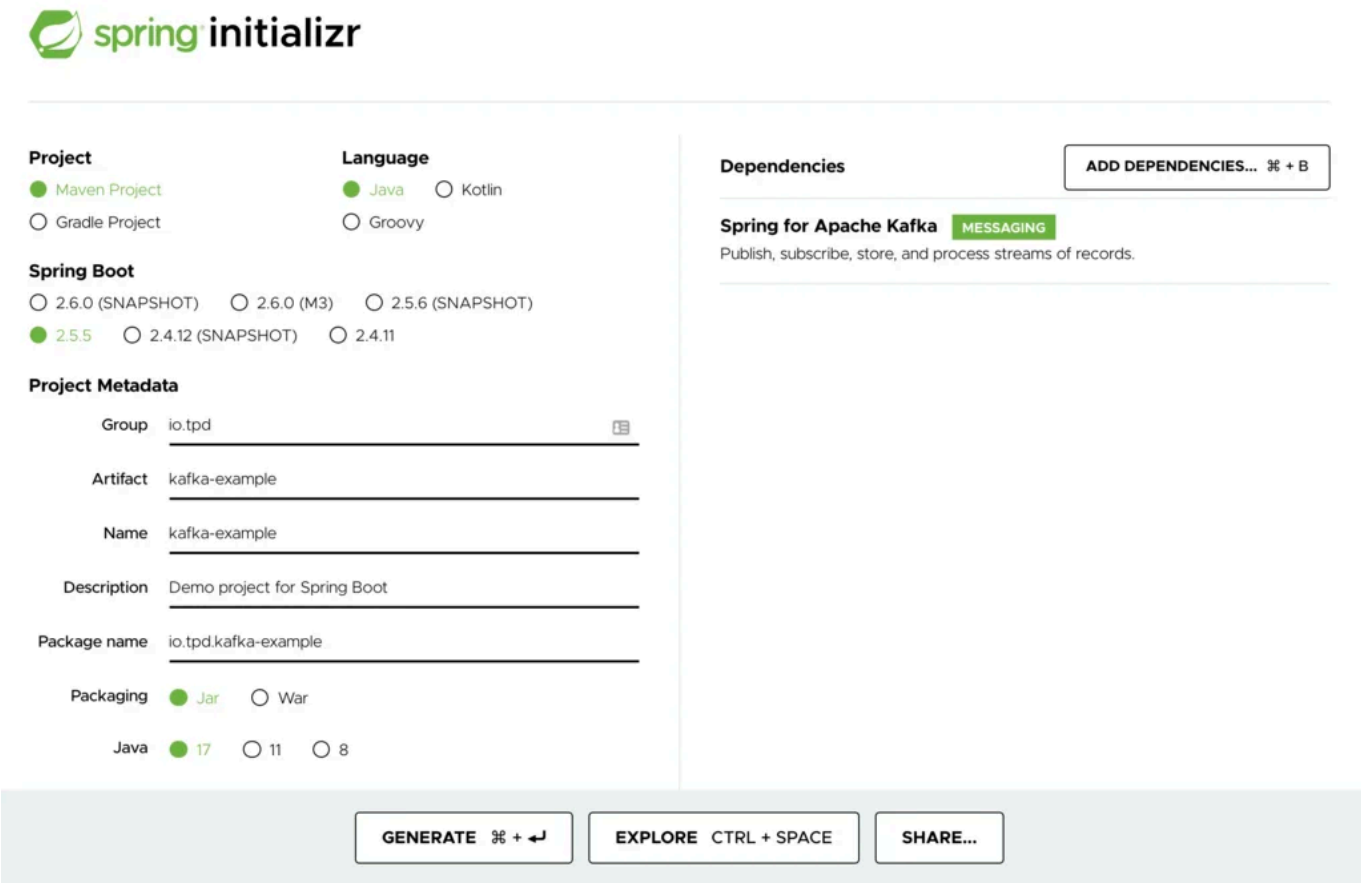
```
    - "9092:9092"
    environment:
      KAFKA_ADVERTISED_HOST_NAME: 127.0.0.1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_AUTO_CREATE_TOPICS_ENABLE: 'false'

```

Note that I configured Kafka to **not** create topics automatically. We will create our topic from the Spring Boot application since we want to pass some custom configuration anyway. If you want to play around with these Docker images (e.g. to use multiple nodes), have a look at the `wurstmeister/zookeeper` image docs.

To start up Kafka and Zookeeper containers, just run `docker-compose up` from the folder where this file lives.

Basic Spring Boot and Kafka application



Spring Initializer Kafka

The easiest way to get a skeleton for our app is to navigate to start.spring.io, fill in the basic details for our project and select Kafka as a dependency. Then, download the zip file and use your favorite IDE to load the sources.

Let's use YAML for our configuration. You may need to rename the `application.properties` file inside `src/main/java/resources` to `application.yml`. These are the configuration values we are going to use for this sample application:

```yaml
spring:
  kafka:
    consumer:
      group-id: tpd-loggers
      auto-offset-reset: earliest
    # change this property if you are using your own
    # Kafka cluster or your Docker IP is different
    bootstrap-servers: localhost:9092

tpd:
  topic-name: advice-topic
  messages-per-request: 10
```

The first block of properties is Spring Kafka configuration:

- The `group-id` that will be used by default by our consumers.
- The `auto-offset-reset` property is set to `earliest`, which means that the consumers will start reading messages from the earliest one available when there is no existing offset for that consumer.
- The server to use to connect to Kafka, in this case, the only one available if you use the single-node configuration. Note that this property is redundant if you use the default value, `localhost:9092`.

The second block is application-specific. We define the Kafka topic name and the number of messages to send every time we do an HTTP REST request.

The Message class

This is the Java record that we will use as Kafka message. We need to use the `@JsonProperty` annotations for the record fields so Jackson can deserialize it properly. If you're using a version of Java that doesn't support records, you can use the old version of this file available on GitHub.

PracticalAdvice class

```
1  package io.tpd.kafkaexample;
2
3  import com.fasterxml.jackson.annotation.JsonProperty;
4
5  record PracticalAdvice(@JsonProperty("message") String message,
6                         @JsonProperty("identifier") int identifier) {
7  }
```

Kafka Producer configuration in Spring Boot

To keep the application simple, we will add the configuration in the main Spring Boot class. Eventually, we want to include here both producer and consumer configuration, and use three different variations for deserialization. Remember that you can find the complete source code in the GitHub repository.

First, let's focus on the Producer configuration.

Spring Boot Kafka Producer

```java
@SpringBootApplication
public class KafkaExampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(KafkaExampleApplication.class, args);
    }

    @Autowired
    private KafkaProperties kafkaProperties;

    @Value("${tpd.topic-name}")
    private String topicName;

    // Producer configuration

    @Bean
    public Map<String, Object> producerConfigs() {
        Map<String, Object> props =
                new HashMap<>(kafkaProperties.buildProducerProperties());




        return props;
    }

    @Bean
    public ProducerFactory<String, Object> producerFactory() {

    }

    @Bean
    public KafkaTemplate<String, Object> kafkaTemplate() {

    }


```

```
38      @Bean
40          return new NewTopic(topicName, 3, (short) 1);
41      }
    }
```

In this configuration, we are setting up two parts of the application:

- The **KafkaTemplate** instance. This is the object we employ to send messages to Kafka. We don't want to use the default one because we want to play with different settings, so we need to inject our custom version in the Spring's application context.
  - We type (with generics) the KafkaTemplate to have a plain String key, and an Object as value. The reason to have Object as a value is that we want to send multiple object types with the same template.
  - The KafkaTemplate accepts a ProducerFactory as a parameter. We create a default factory with our custom producer configuration values.
  - The Producer Configuration is a simple key-value map. We inject the default properties using `@Autowired` to obtain the `KafkaProperties` bean. Then, we build our map passing the default values for the producer and overriding the default Kafka key and value serializers. The producer will serialize keys as Strings using the Kafka library's `StringSerializer` and will do the same for values but this time using JSON, with a `JsonSerializer`, in this case provided by Spring Kafka.
- The **Kafka topic**. When we inject a `NewTopic` bean, we're instructing the Kafka's `AdminClient` bean (already in the context) to create a topic with the given configuration. The first parameter is the name (`advice-topic`, from the app configuration), the second

is the number of partitions (`3`) and the third one is the replication factor (`1`, since we're using a single node anyway).

About Kafka Serializers and Deserializers for Java

There are a few basic Serializers available in the core Kafka library (javadoc) for Strings, all kind of number classes and byte arrays, plus the JSON ones provided by Spring Kafka (javadoc).

On top of that, you can create your own Serializers and Deserializers just by implementing `Serializer` or `ExtendedSerializer`, or their corresponding versions for deserialization. That gives you a lot of flexibility to optimize the amount of data traveling through Kafka, in case you need to do so. As you can see in those interfaces, Kafka works with plain byte arrays so, eventually, no matter what complex type you're working with, it needs to be transformed to a `byte[]`.

Knowing that, you may wonder why someone would want to use JSON with Kafka. It's quite inefficient since you're transforming your objects to JSON and then to a byte array. But you have to consider two main advantages of doing this:

- JSON is more readable by a human than an array of bytes. If you want to debug or analyze the contents of your Kafka topics, it's going to be way simpler than looking at bare bytes.

- JSON is a standard, whereas default byte array serializers depend on the programming language implementation. Thus, if you want to consume messages from multiple programming languages, you would need to replicate the (de)serializer logic in all those languages.

On the other hand, if you are concerned about the traffic load in Kafka, storage, or speed in (de)serialization, you may want to choose byte arrays and even go for your own serializer/deserializer implementation.

Sending messages with Spring Boot and Kafka

Following the plan, we create a Rest Controller and use the injected `KafkaTemplate` to produce some JSON messages when the endpoint is requested.

This is the first implementation of the controller, containing only the logic producing the messages.

HelloKafkaController

```
@RestController
public class HelloKafkaController {

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

    private final KafkaTemplate<String, Object> template;
    private final String topicName;
    private final int messagesPerRequest;
    private CountDownLatch latch;

    public HelloKafkaController(
            final KafkaTemplate<String, Object> template,
            @Value("${tpd.topic-name}") final String topicName,
            @Value("${tpd.messages-per-request}") final int messagesPerRequest) {
        this.template = template;
        this.topicName = topicName;
        this.messagesPerRequest = messagesPerRequest;
    }

    @GetMapping("/hello")
    public String hello() throws Exception {
        latch = new CountDownLatch(messagesPerRequest);
        IntStream.range(0, messagesPerRequest)
                .forEach(i -> this.template.send(topicName, String.valueOf(i),
                        new PracticalAdvice("A Practical Advice", i))
                );
        latch.await(60, TimeUnit.SECONDS);
        logger.info("All messages received");
        return "Hello Kafka!";
    }
}

```

In the constructor, we pass some configuration parameters and the KafkaTemplate that we customized to send String keys and JSON values. Then, when the API client requests the `/hello` endpoint, we send 10 messages (that's the configuration value) and then we block the thread for a maximum of 60 seconds. As you can see, there is no implementation yet for the Kafka consumers to decrease the latch count. After the latch gets unlocked, we return the message `Hello Kafka!` to our client.

This entire lock idea is not a pattern that would see in a real application, but it's good for the sake of this example. That way, you can check the number of messages received. If you prefer, you can remove the latch and return the "Hello Kafka!" message before receiving the messages.

Kafka Consumer configuration

As mentioned previously on this post, we want to demonstrate different ways of deserialization with Spring Boot and Spring Kafka and, at the same time, see how multiple consumers can work in a load-balanced manner when they are part of the same consumer-group.

Spring Boot Kafka configuration - Consumer

```
@SpringBootApplication
public class KafkaExampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(KafkaExampleApplication.class, args);
```

```
    }

    @Autowired
    private KafkaProperties kafkaProperties;

    @Value("${tpd.topic-name}")
    private String topicName;

    // Producer configuration
    // omitted...

    // Consumer configuration

    // If you only need one kind of deserialization, you only need to set the
    // Consumer configuration properties. Uncomment this and remove all others below.
//    @Bean
//    public Map<String, Object> consumerConfigs() {
//        Map<String, Object> props = new HashMap<>(
//                kafkaProperties.buildConsumerProperties()
//        );
//        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
//                StringDeserializer.class);
//        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
//                JsonDeserializer.class);
//        props.put(ConsumerConfig.GROUP_ID_CONFIG,
//                "tpd-loggers");
//
//        return props;
//    }

    @Bean
    public ConsumerFactory<String, Object> consumerFactory() {
        final JsonDeserializer<Object> jsonDeserializer = new JsonDeserializer<>();
        jsonDeserializer.addTrustedPackages("*");
        return new DefaultKafkaConsumerFactory<>(
                kafkaProperties.buildConsumerProperties(), new StringDeserializer(), jsonD
        );
```

```
}

@Bean
public ConcurrentKafkaListenerContainerFactory<String, Object> kafkaListenerContainerF
    ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory());

    return factory;
}

// String Consumer Configuration

@Bean
public ConsumerFactory<String, String> stringConsumerFactory() {
    return new DefaultKafkaConsumerFactory<>(
            kafkaProperties.buildConsumerProperties(), new StringDeserializer(), new S
    );
}

@Bean
public ConcurrentKafkaListenerContainerFactory<String, String> kafkaListenerStringCont
    ConcurrentKafkaListenerContainerFactory<String, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(stringConsumerFactory());

    return factory;
}

// Byte Array Consumer Configuration

@Bean
public ConsumerFactory<String, byte[]> byteArrayConsumerFactory() {
    return new DefaultKafkaConsumerFactory<>(
            kafkaProperties.buildConsumerProperties(), new StringDeserializer(), new B
    );
}
```

```
81
82      @Bean
83      public ConcurrentKafkaListenerContainerFactory<String, byte[]> kafkaListenerByteArrayC
84          ConcurrentKafkaListenerContainerFactory<String, byte[]> factory =
85                  new ConcurrentKafkaListenerContainerFactory<>();
86          factory.setConsumerFactory(byteArrayConsumerFactory());
87          return factory;
88      }
    }
```

This configuration may look extense but take into account that, to demonstrate these three types of deserialization, we have repeated three times the creation of the ConsumerFactory and the KafkaListenerContainerFactory instances so we can switch between them in our consumers.

The basic steps to configure a consumer are:

1. [Omitted] Set up the Consumer properties in a similar way as we did for the Producer. We can skip this step since the only configuration we need is the Group ID, specified in the Spring Boot properties file, and the key and value deserializers, which we will override while creating the customized consumer and KafkaListener factories. **If you only need one configuration**, meaning always the same type of Key and Value deserializers, this commented code block **is the only thing you need**. Adjust the deserializer types to the ones you want to use.
2. Create the `ConsumerFactory` to be used by the `KafkaListenerContainerFactory`. We create three, switching the value deserializer in each case to 1) a JSON deserializer, 2) a

String deserializer and 3) a Byte Array deserializer.

1. Note that, after creating the JSON Deserializer, we're including an extra step to specify that we trust all packages. You can fine-tune this in your application if you want. If we don't do this, we will get an error message saying something like: `java.lang.IllegalArgumentException: The class [] is not in the trusted packages`.

3. Construct the Kafka Listener container factory (a concurrent one) using the previously configured Consumer Factory. Again, we do this three times to use a different one per instance.

Receiving messages with Spring Boot and Kafka in JSON, String and byte[] formats

It’s time to show how the Kafka consumers look like. We will use the `@KafkaListener` annotation since it simplifies the process and takes care of the deserialization to the passed Java type.

Kafka listeners

```
@RestController
public class HelloKafkaController {

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

    private final KafkaTemplate<String, Object> template;
    private final String topicName;
```

```
private final int messagesPerRequest;
private CountDownLatch latch;

public HelloKafkaController(
        final KafkaTemplate<String, Object> template,
        @Value("${tpd.topic-name}") final String topicName,
        @Value("${tpd.messages-per-request}") final int messagesPerRequest) {
    this.template = template;
    this.topicName = topicName;
    this.messagesPerRequest = messagesPerRequest;
}

@GetMapping("/hello")
public String hello() throws Exception {
    latch = new CountDownLatch(messagesPerRequest);
    IntStream.range(0, messagesPerRequest)
            .forEach(i -> this.template.send(topicName, String.valueOf(i),
                    new PracticalAdvice("A Practical Advice", i))
            );
    latch.await(60, TimeUnit.SECONDS);
    logger.info("All messages received");
    return "Hello Kafka!";
}

@KafkaListener(topics = "advice-topic", clientIdPrefix = "json",
        containerFactory = "kafkaListenerContainerFactory")
public void listenAsObject(ConsumerRecord<String, PracticalAdvice> cr,
                           @Payload PracticalAdvice payload) {
    logger.info("Logger 1 [JSON] received key {}: Type [{}] | Payload: {} | Record: {}
            typeIdHeader(cr.headers()), payload, cr.toString());
    latch.countDown();
}

@KafkaListener(topics = "advice-topic", clientIdPrefix = "string",
        containerFactory = "kafkaListenerStringContainerFactory")
public void listenasString(ConsumerRecord<String, String> cr,
                           @Payload String payload) {
```

```
46
47          logger.info("Logger 2 [String] received key {}: Type [{}] | Payload: {} | Record:
48                  typeIdHeader(cr.headers()), payload, cr.toString());
49          latch.countDown();
50      }
55                                  @PayloadByte[] payload) {
56          logger.info("Logger 3 [ByteArray] received key {}: Type [{}] | Payload: {} | Recor
57                  typeIdHeader(cr.headers()), payload, cr.toString());
58          latch.countDown();
59      }
60
61      private static String typeIdHeader(Headers headers) {
63                  .filter(header -> header.key().equals("__TypeId__"))
64                  .findFirst().map(header -> new String(header.value())).orElse("N/A");
65      }
66  }
```

There are three listeners in this class. First, let’s describe the `@KafkaListener` annotation’s parameters:

- All listeners are consuming from the same topic, `advice-topic`. This parameter is mandatory.
- The parameter `clientIdPrefix` is optional. I'm using it here so the logs are more human-friendly. You will know which consumer does what by its name prefix. Kafka will append a number this prefix.
- The `containerFactory` parameter is optional, you can also rely on naming convention. If you don't specify it, it will look for a bean with the name `kafkaListenerContainerFactory`, which is also the default name used by Spring Boot when autoconfiguring Kafka. You can also override it by using the same name (although it looks like magic for someone who doesn't know about the convention). We need to set it explicitly because we want to use a different one for each listener, to be able to use different deserializers.

Note that the first argument passed to all listeners is the same, a `ConsumerRecord`. The second one, annotated with `@Payload` is redundant if we use the first. We can access the payload using the method `value()` in `ConsumerRecord`, but I included it so you see how simple it’s to get directly the message payload by inferred deserialization.

The TypeId Header in Kafka

The `__TypeId__` header is automatically set by the Kafka library by default. The utility method `typeIdHeader` that I use here is just to get the string representation since you will only see a byte array in the output of `ConsumerRecord`'s `toString()` method. This TypeId header can be useful for deserialization, so you can find the type to map the data to. It's not needed for JSON deserialization because that specific deserializer is made by the Spring team and they infer the type from the method's argument.

Running the application

All the code in this post is available on GitHub: Kafka and Spring Boot Example. If you find it useful, please give it a star!

Now that we finished the Kafka producer and consumers, we can run Kafka and the Spring Boot app:

```
$ docker-compose up -d
Starting kafka-example_zookeeper_1 ... done
Starting kafka-example_kafka_1     ... done
$ mvn spring-boot:run
...
```

The Spring Boot app starts and the consumers are registered in Kafka, which assigns a partition to them. We configured the topic with three partitions, so each consumer gets one of them assigned.

Output - Kafka Topic partitions

```
[Consumer clientId=bytearray-0, groupId=tpd-loggers] Subscribed to topic(s): advice-topic
[Consumer clientId=json-0, groupId=tpd-loggers] Subscribed to topic(s): advice-topic
[Consumer clientId=string-0, groupId=tpd-loggers] Subscribed to topic(s): advice-topic
[...]
[Consumer clientId=bytearray-0, groupId=tpd-loggers] Notifying assignor about the new Assignmer
[Consumer clientId=string-0, groupId=tpd-loggers] Notifying assignor about the new Assignment(p
[Consumer clientId=json-0, groupId=tpd-loggers] Notifying assignor about the new Assignment(par
```

We can try now an HTTP call to the service. You can use your browser or a command-line tool like `curl`, for example:

Invoking the endpoint

```
$ curl localhost:8080/hello
```

The output in the logs should look like this:

Kafka Listeners output

```
[ntainer#1-0-C-1] [...]: Logger 2 [String] received key 0: Type [io.tpd.kafkaexample.Pract
[ntainer#2-0-C-1] [...]: Logger 3 [ByteArray] received key 1: Type [io.tpd.kafkaexample.Pr
[ntainer#2-0-C-1] [...]: Logger 3 [ByteArray] received key 5: Type [io.tpd.kafkaexample.Pr
[ntainer#1-0-C-1] [...]: Logger 2 [String] received key 2: Type [io.tpd.kafkaexample.Pract
[ntainer#2-0-C-1] [...]: Logger 3 [ByteArray] received key 7: Type [io.tpd.kafkaexample.Pr
[ntainer#1-0-C-1] [...]: Logger 2 [String] received key 3: Type [io.tpd.kafkaexample.Pract
[ntainer#2-0-C-1] [...]: Logger 3 [ByteArray] received key 8: Type [io.tpd.kafkaexample.Pr
[ntainer#1-0-C-1] [...]: Logger 2 [String] received key 9: Type [io.tpd.kafkaexample.Pract
[ntainer#0-0-C-1] [...]: Logger 1 [JSON] received key 4: Type [N/A] | Payload: PracticalAc
[ntainer#0-0-C-1] [...]: Logger 1 [JSON] received key 6: Type [N/A] | Payload: PracticalAc
```

```
11 | [nio-8080-exec-1] [...]: All messages received
```

Explanation

Kafka is hashing the message key (a simple string identifier) and, based on that, placing messages into different partitions. Each consumer gets the messages in its assigned partition and uses its deserializer to convert it to a Java object. Remember, our producer always sends

As you can see in the logs, each deserializer manages to do its task so the String consumer prints the raw JSON message, the Byte Array shows the byte representation of that JSON String, and the JSON deserializer is using the Java Type Mapper to convert it to the original class, PracticalAdvice. You can have a look at the logged ConsumerRecord and you'll see the headers, the assigned partition, the offset, etc.

And that's how you can Send and Receive JSON messages with Spring Boot and Kafka. I hope that you found this guide useful, below you have some code variations so you can explore a bit more how Kafka works.

Should you have any feedback, let me know via Twitter or comments.

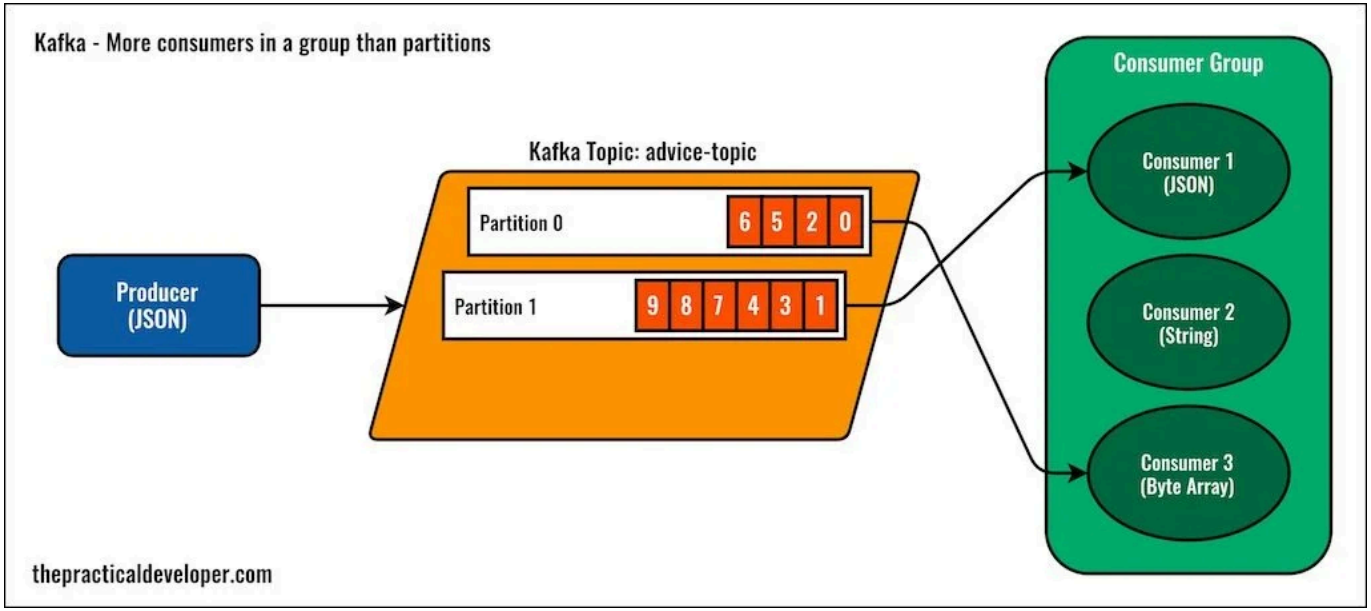
Try some Kafka Exercises

If you are new to Kafka, you may want to try some code changes to better understand how Kafka works.

Request /hello multiple times

Make a few requests and then look at how the messages are distributed across partitions. **Kafka messages with the same key are always placed in the same partitions**. This feature is very useful when you want to make sure that all messages for a given user, or process, or whatever logic you're working on, are received by the same consumer in the same order as they were produced, no matter how much load balancing you're doing.

Reduce the number of partitions



Kafka - more consumers in a group than partitions

First, make sure to restart Kafka so you just discard the previous configuration.

Then, redefine the topic in the application to have only 2 partitions:

Redefine Kafka Topic

```
@Bean
public NewTopic adviceTopic() {
    return new NewTopic(topicName, 2, (short) 1);
}
```

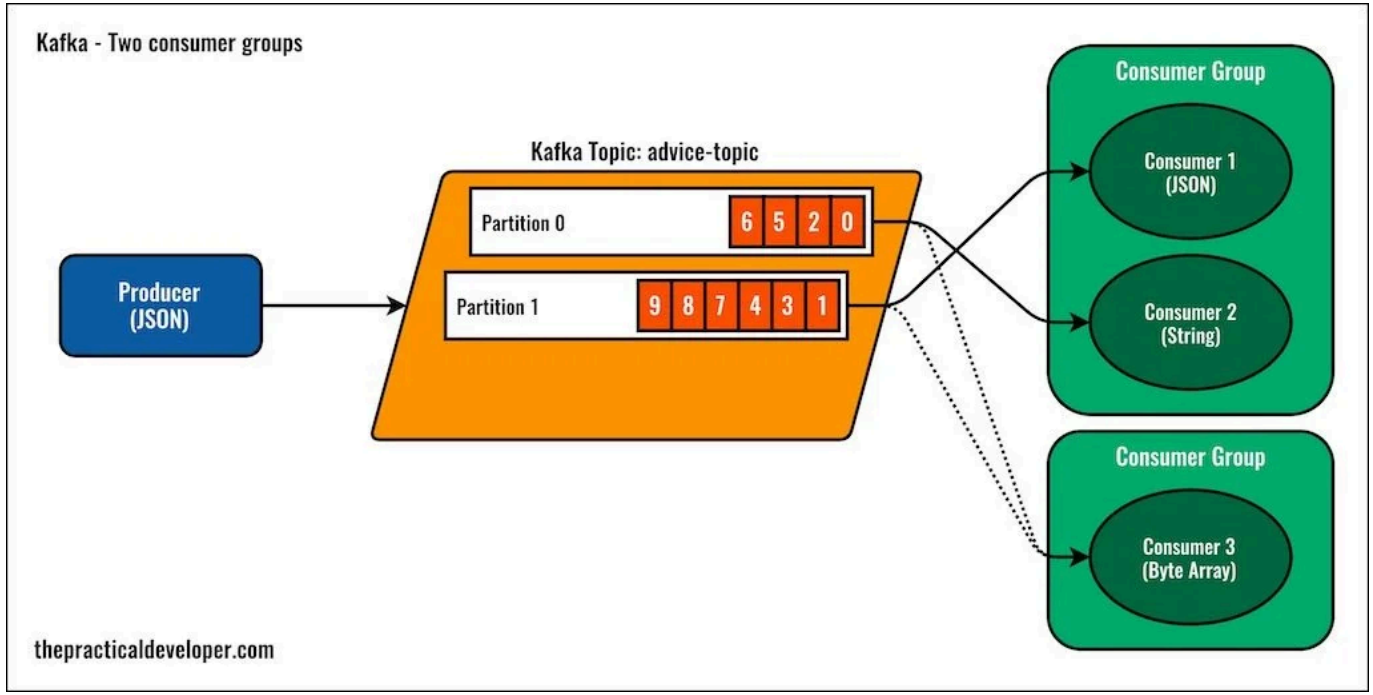
Now, run the app again and do a request to the `/hello` endpoint.

Kafka consumer with no assigned partition

```
Logger 3 [ByteArray] received key 0: Type [io.tpd.kafkaexample.PracticalAdvice] | Payload:
        Logger 3 [ByteArray] received key 2: Type [io.tpd.kafkaexample.PracticalAdvice] |
        Logger 3 [ByteArray] received key 5: Type [io.tpd.kafkaexample.PracticalAdvice] |
        Logger 3 [ByteArray] received key 6: Type [io.tpd.kafkaexample.PracticalAdvice] |
        Logger 1 [JSON] received key 1: Type [N/A] | Payload: PracticalAdvice[message=A Pr
        Logger 1 [JSON] received key 3: Type [N/A] | Payload: PracticalAdvice[message=A Pr
        Logger 1 [JSON] received key 4: Type [N/A] | Payload: PracticalAdvice[message=A Pr
        Logger 1 [JSON] received key 7: Type [N/A] | Payload: PracticalAdvice[message=A Pr
        Logger 1 [JSON] received key 8: Type [N/A] | Payload: PracticalAdvice[message=A Pr
        Logger 1 [JSON] received key 9: Type [N/A] | Payload: PracticalAdvice[message=A Pr
        All messages received
```

One of the consumers (in this case the String version) doesn't receive any messages. This is the expected behavior since there are no more partitions available for it within the same consumer group. To reproduce this behavior, you may need to destroy the kafka cluster in docker with `docker-compose down -v` and start it back again, so the topic is re-created again from Java.

Change one Consumer's Group Identifier



Kafka - Two consumer groups

Keep the changes from the previous case, the topic has now only 2 partitions. We are now **changing the group id** of one of our consumers, so it's working independently.

Changing Kafka group id

```
@KafkaListener(topics = "advice-topic", clientIdPrefix = "bytearray",
        containerFactory = "kafkaListenerByteArrayContainerFactory",
        groupId = "tpd-loggers-2")
public void listenAsByteArray(ConsumerRecord<String, byte[]> cr,
                              @Payload byte[] payload) {
        logger.info("Logger 3 [ByteArray] received key {}", cr.key());
    latch.countDown();
}
```

We also changed the logged message to easily differentiate it from the rest. It only prints the received key. Now, we also need to change the `CountDownLatch` so it expects twice the number of messages since there are two different consumer groups. Keep reading for a detailed explanation.

Latch sets to twice the number

```
1 | latch = new CountDownLatch(messagesPerRequest * 2);
```

Why do we need this? As I described at the beginning of this post, **when consumers belong to the same Consumer Group, they're (conceptually) working on the same task**. We're implementing a load-balanced mechanism in which concurrent workers get messages from different partitions without needing to process each other's messages. They're splitting the job across workers.

In this example, I also changed the "task" of the last consumer to better understand this: it's printing something different. Since we changed the group id, this consumer works separate from the others. Kafka will assign both partitions to it since it's the only one in its group. The Byte Array consumer receives all messages now. See the logs below.

Two Kafka Consumers

```
Logger 3 [ByteArray] received key 0
Logger 3 [ByteArray] received key 2
Logger 3 [ByteArray] received key 5
Logger 3 [ByteArray] received key 6
Logger 2 [String] received key 1: Type [io.tpd.kafkaexample.PracticalAdvice] | Payload: {"
```

```
Logger 2 [String] received key 3: Type [io.tpd.kafkaexample.PracticalAdvice] | Payload: {'
Logger 2 [String] received key 4: Type [io.tpd.kafkaexample.PracticalAdvice] | Payload: {'
Logger 2 [String] received key 7: Type [io.tpd.kafkaexample.PracticalAdvice] | Payload: {'
Logger 2 [String] received key 8: Type [io.tpd.kafkaexample.PracticalAdvice] | Payload: {'
Logger 2 [String] received key 9: Type [io.tpd.kafkaexample.PracticalAdvice] | Payload: {'
Logger 1 [JSON] received key 0: Type [N/A] | Payload: PracticalAdvice[message=A Practical
Logger 1 [JSON] received key 2: Type [N/A] | Payload: PracticalAdvice[message=A Practical
Logger 1 [JSON] received key 5: Type [N/A] | Payload: PracticalAdvice[message=A Practical
Logger 1 [JSON] received key 6: Type [N/A] | Payload: PracticalAdvice[message=A Practical
Logger 3 [ByteArray] received key 1
Logger 3 [ByteArray] received key 3
Logger 3 [ByteArray] received key 4
Logger 3 [ByteArray] received key 7
Logger 3 [ByteArray] received key 8
Logger 3 [ByteArray] received key 9
All messages received
```

With these exercises, changing parameters here and there, I think you should have understood the different basic concepts. Remember: if you liked this post please share it or let me know your feedback via Twitter.