Integrating Kafka in a Spring Boot Java application

Introduction

Apache Kafka is an open-source distributed event streaming platform used for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.

In this tutorial, we will cover all the necessary steps for integrating Kafka into a Spring Boot application:

- Setting up the docker Kafka container
- Configuring Spring Boot to work with Kafka
- Publishing a message to a Kafka topic
- Consuming a message from a Kafka topic
- Testing the Kafka integration

Prerequisites

- Java 17 or later installed (with associated JDK)
- An IDE of your choice (Eclipse, IntelliJ IDEA, etc.)
- Spring Boot configured in your project
- Maven/Gradle configured in your project (Maven will be used in this tutorial)
- Docker Desktop installed

Setting up the docker Kafka container

Before we begin the integration, we must install our Kafka instance in a docker container. We can use a publicly available Kafka image and configure it before it is deployed. Open your Spring Boot project in your desired IDE and create a new text file named “docker-compose.yml”. This file will contain our Kafka image configuration for deployment within a container.

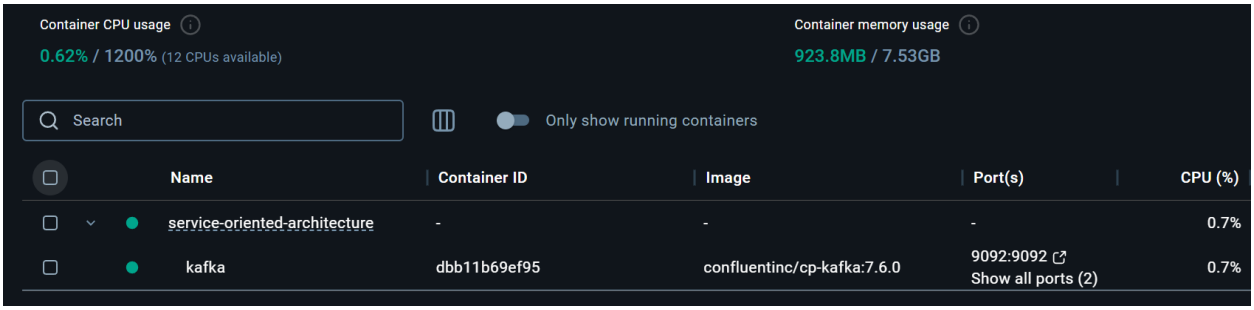
Add the following in “docker-compose.yml”:

```
version: '3'
services:
  kafka:
    image: 'confluentinc/cp-kafka:7.6.0'
    hostname: kafka
```

```
    container_name: kafka
    ports:
      - "9092:9092"
      - "9101:9101"
    environment:
      KAFKA_NODE_ID: 1
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: CONTROLLER:PLAINTEXT,
PLAINTEXT:PLAINTEXT, PLAINTEXT_HOST:PLAINTEXT
      KAFKA_ADVERTISED_LISTENERS: "PLAINTEXT://kafka:29092,
PLAINTEXT_HOST://localhost:9092"
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
      KAFKA_JMX_PORT: 9101
      KAFKA_JMX_HOSTNAME: localhost
      KAFKA_PROCESS_ROLES: "broker, controller"
      KAFKA_CONTROLLER_QUORUM_VOTERS: "1@kafka:29093"
      KAFKA_LISTENERS: "PLAINTEXT://kafka:29092, CONTROLLER://kafka:29093,
PLAINTEXT_HOST://0.0.0.0:9092"
      KAFKA_INTER_BROKER_LISTENER_NAME: "PLAINTEXT"
      KAFKA_CONTROLLER_LISTENER_NAMES: "CONTROLLER"
      CLUSTER_ID: "MkU3OEVBNTcwNTJENDM2Qk"
```

This will set up all the configurations needed for the Kafka container. To create and deploy the container, we must do the following:

- Open Docker Desktop.
- From your IDE's CMD/PowerShell window, navigate to the folder where your "docker-compose.yml" file is located.
- Run the following command: docker-compose up -d.
- Inside your Docker Desktop interface, we should see that a new container has been added and it's up and running:



Configuring Spring Boot to work with Kafka

Now that our Kafka instance is up and running, we need to add some required dependencies to our Spring Boot application. Add the following dependencies in pom.xml:

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

This will enable us to use Kafka related classes and methods. To finish up the configuration, we will set up the KafkaConfig class, which will declare the necessary factories for the Kafka consumers and producers:

```
@EnableKafka
@Configuration
public class KafkaConfig {

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, String>
kafkaListenerContainerFactory(
            ConsumerFactory<String, String> consumerFactory
    ) {
        ConcurrentKafkaListenerContainerFactory<String, String> factory =
                new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory);

        return factory;
    }

    @Bean
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "message-group");
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        return props;
    }

    @Bean
    public ConsumerFactory<String, String> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(
                consumerConfigs(),
                new StringDeserializer(),
                new StringDeserializer()
        );
    }
}
```

Publishing a message to a Kafka topic

For simplicity, our Spring Boot application will act both as a producer and as a consumer of a Kafka topic. Let us implement the KafkaMessageProducer class:

```
@Service
public class KafkaMessageProducer {

    private final KafkaTemplate<String, String> kafkaTemplate;
    private static final long startTime = System.currentTimeMillis();

    public KafkaMessageProducer(KafkaTemplate<String, String> kafkaTemplate)
{
        this.kafkaTemplate = kafkaTemplate;
    }

    @Scheduled(fixedRate = 10000) // Send message every 10 seconds
    public void sendMessage() {
        long secondsElapsed = (System.currentTimeMillis() - startTime) /
1000;
        String message = "It's been " + secondsElapsed + " seconds since
Kafka instance is running!";

        kafkaTemplate.send("message-topic", message);
        System.out.println("Sent: " + message);
    }
}
```

This class will use a Kafka Template object to send our message to a Kafka topic. The sendMessage() function will produce and publish a message on the “message-topic” Kafka topic every 10 seconds. We declared an initial startTime of our application and then calculated the elapsed seconds in this function. The message produced will tell how many seconds have passed since the application has started. The same message will also be printed in the console.

Consuming a message from a Kafka topic

In order to consume a given message in real-time, we must implement the KafkaMessageConsumer class:

```
@Service
public class KafkaMessageConsumer {

    @KafkaListener(topics = "message-topic", groupId = "message-group")
    public void listen(String message) {
        System.out.println("Received from Kafka: " + message);
    }
}
```

This class defines a simple Kafka Listener method that receives a message from the “message-topic” Kafka topic and prints it in the console.

Testing the Kafka integration

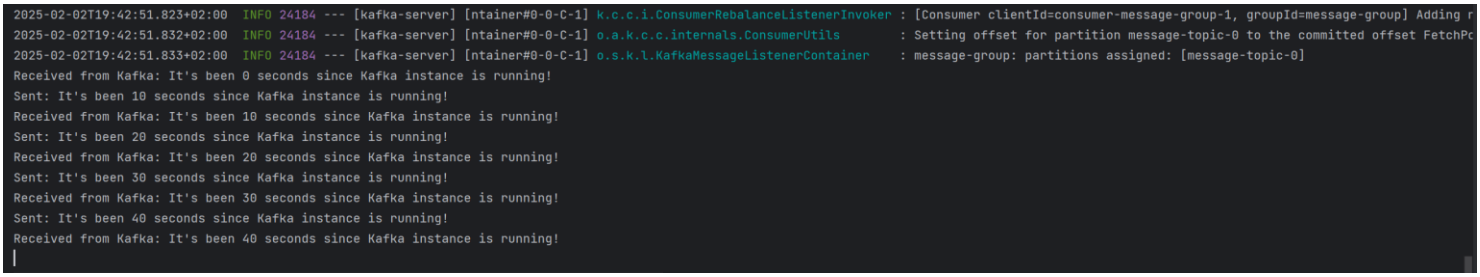
Before running the application, make sure the “@EnableScheduling” annotation is added to your main class:

```
@EnableScheduling
@SpringBootApplication
public class KafkaServerApplication {

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

}
```

Then, simply run your application. You should see messages arriving in real-time in the console every 10 seconds, displaying how much time has elapsed since the Kafka instance is running:



Conclusion

You have now integrated Kafka into a simple Spring Boot Java application that acts as both consumer and producer. This setup demonstrated how to properly set up and configure the Kafka docker image, add the necessary dependencies in your project, set up the Kafka configuration class and implement a producer that streams real-time data alongside a consumer that receives said data and displays it.