Robotic Factory Exercise 4
Event-Driven Microservices

Dominique Blouin, Télécom Paris, Institut Polytechnique de Paris
dominique.blouin@telecom-paris.fr

For this fourth exercise, you will modify your distributed simulator so that it can run in an event-driven way besides the synchronous approach that you developed during the previous exercise. For this, you will use Apache Kafka, which is an open-source distributed event streaming platform.

After setting up a Kafka broker server on your computer, you will modify your simulation microservice so that it creates a topic in the broker server for each factory model that must be simulated. Each model change that occurs as the factory model data evolves during simulation will then be converted into an event that will be published to this topic.

On the client side, your simulator viewer standalone Java application will register to this topic so that it can be notified when the simulation model data has changed to refresh the viewer data.

This exercise is based on the excellent Apache Kafka Tutorial with Spring Boot Reactive & WebFlux by Ali Bouali, which will be used throughout this project exercise.

Introduction to Apache Kafka

Watch the first 23 minutes of the above tutorial to learn the basics of Apache Kafka.

Installing and Testing Apache Kafka

First download Kafka from here and unzip its content into a directory of your choice.

Installing Kafka

Kafka only runs on Linux so if you are using Windows, you will need to either install Kafka on your WSL (Windows Subsystem for Linux) or in a virtual machine such as Docker or Virtual Box. In addition, ensure that Java version 8 or above is installed in the virtual machine.

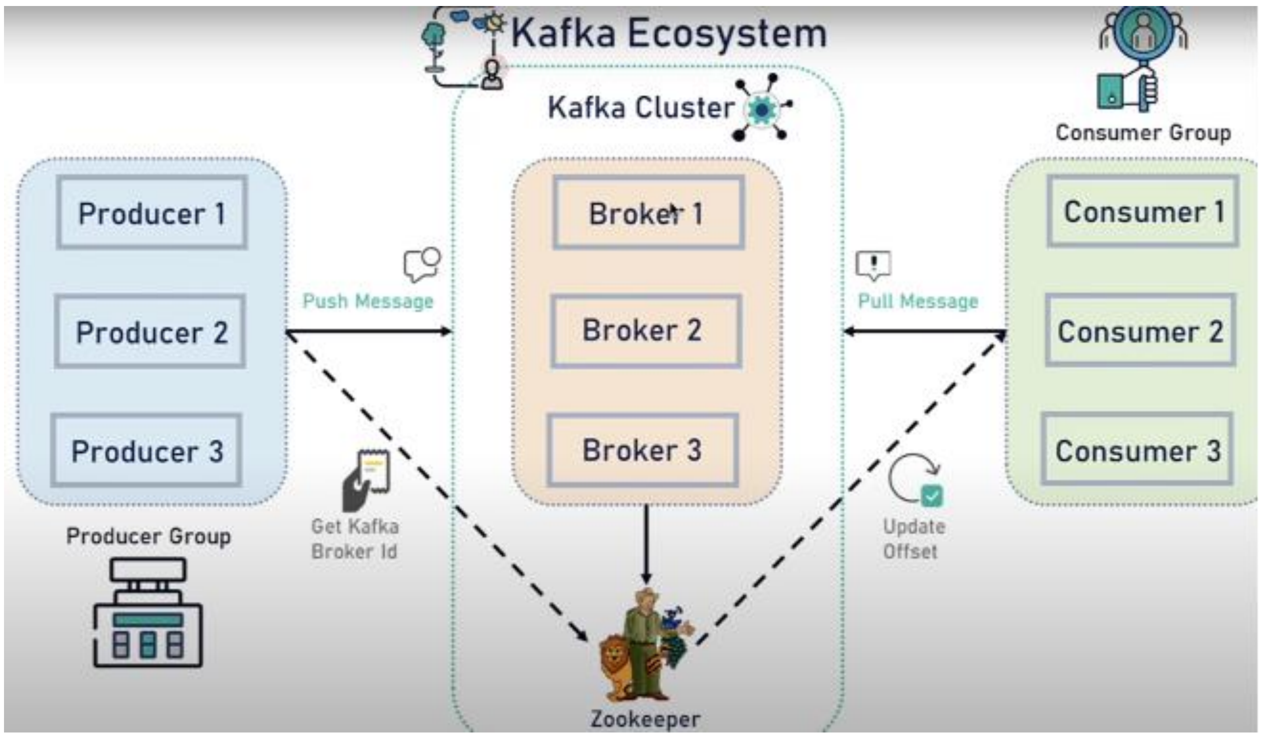
If Kafka is installed on a virtual machine and the microservice runs on another machine (e.g., the host machine such as Windows), you will also need to modify the default configuration of Kafka. The purpose is to specify the name of the host of the Kafka listeners so that they can be accessed by the microservice running on a different machine.

To do so, edit the configuration file named *server.properties* located under the *config* directory of the Kafka root folder (e.g., *kafka_2.13-3.8.1/config/ server.properties*). Make sure the following line is present in the configuration file:

```
listeners=PLAINTEXT://localhost:9092
```

Starting Kafka

As seen in the figure below, Kafka typically runs in a cluster so that it can contain several message brokers working together. In this way, several data streams can be processed by several brokers in parallel to increase speed. In addition, the data can be replicated across several brokers to improve reliability; if one broker fails, the data is still available from another broker. Kafka can also balance the loads across multiple brokers to improve the overall scalability.



To manage these brokers, another tool called Zookeeper is used. Therefore, the first thing to start before Kafka itself is Zookeeper. To do so, open a terminal, navigate to the installation directory of Kafka and run the following command:

```
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```

Check that no error message is shown on the console meaning that Zookeeper started without problems.

Next, start the Kafka server by opening another terminal, navigating to the installation directory of Kafka, and running the following command:

```
$ bin/kafka-server-start.sh config/server.properties
```

Again, check that the server started correctly.

Creating a Test Topic

To test that Kafka is working properly, create a simple topic. To do so, open another terminal, navigate to the installation directory of Kafka and run the following command:

```
bin/kafka-topics.sh --create --topic my-test-topic --bootstrap-server localhost:9092
```

Ensure that the topic was properly created by looking at the message in the console.

Consuming the Topic Messages

In another terminal, launch a message consumer application for the topic you just created by navigating to the installation directory of Kafka and running the following command:

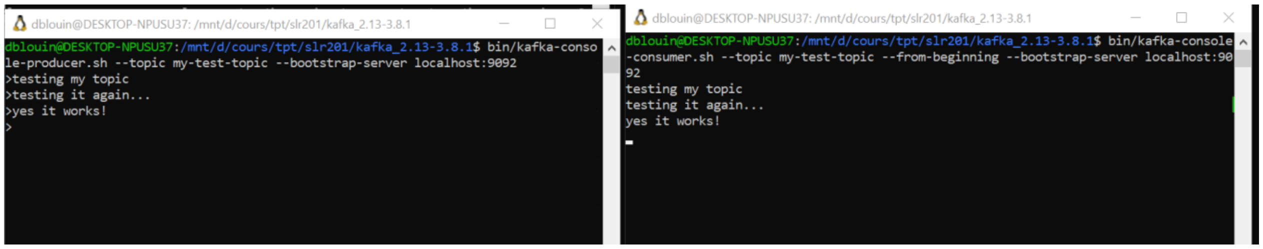
```
bin/kafka-console-consumer.sh --topic my-test-topic --from-beginning --bootstrap-
server localhost:9092
```

Publishing Messages with the Console Message Producer Application

Now open another terminal, navigate to the installation directory of Kafka and run a message producer application for the topic previously created using the following command:

```
bin/kafka-console-producer.sh --topic my-test-topic --bootstrap-server localhost:9092
```

Move the producer and consumer terminal windows side by side and check that messages that you enter in the producer terminal are displayed in the producer console as shown below.


dblouin@DESKTOP-NPUSU37: /mnt/d/cours/tpt/slr201/kafka_2.13-3.8.1

```
dblouin@DESKTOP-NPUSU37:/mnt/d/cours/tpt/slr201/kafka_2.13-3.8.1$ bin/kafka-conso
le-producer.sh --topic my-test-topic --bootstrap-server localhost:9092
>testing my topic
>testing it again...
>yes it works!
>
```

dblouin@DESKTOP-NPUSU37: /mnt/d/cours/tpt/slr201/kafka_2.13-3.8.1

```
dblouin@DESKTOP-NPUSU37:/mnt/d/cours/tpt/slr201/kafka_2.13-3.8.1$ bin/kafka-console
-consumer.sh --topic my-test-topic --from-beginning --bootstrap-server localhost:90
92
testing my topic
testing it again...
yes it works!
```

Errors that can Occur when Starting Kafka

It may happen that the Kafka server no longer starts due to a problem accessing the *tmp/kafka-logs* directory that stores the topic events. To solve this problem, navigate to the root directory of the Linux system and delete the *tmp/kafka-logs* directory.

Modifying the Simulation Microservice so that it Publishes Simulation Events

Is this section, you will create classes for your simulation microservice so that it can publish simulation events to Kafka topics.

Configuring the Microservice Project for Kafka

Spring Boot provides a library to ease the development of event-based microservices with Kafka. To use this library, open the *pom.xml* file in your microservice project and add the following dependencies under the <*dependencies*> XML tag.

```
<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>
```

Save the file and observe that Maven really downloaded the Spring Kafka library by opening the *Maven Dependencies* folder under the microservice project.

In a standard event-based microservice, you would also need to add different Kafka properties to the *application.properties* file to configure the Kafka producer. However, because we need to use a custom Jackson object mapper, this configuration can only be set using Java code and will be explained later.

Refactoring the Simulator Factory Model by Creating and Using Notifier Classes

As described during the first robotic factory project exercise, the simulator is based on a Model-View-Controller (MVC) architectural design pattern. The factory model class maintains a list of observers (canvas viewers, for instance) that are notified whenever any factory model data is changed. One problem with the actual factory model class is that it handles both the responsibility of carrying the factory data and notifying the model observers. As we have already seen, this breaks the Single Responsibility Principle (SRP).

Our objective in this project exercise is to change the way the model notifies its observers so that Kafka is used. Therefore, the first task is to refactor the factory model so that it uses a separate class to notify its observers.

For this, first create an interface named *FactoryModelChangedNotifier* that will provide the *notifyObservers()*, *addObserver()* and *removeObserver()* method signatures. Those methods are currently provided by the *Factory* class. Then, add a notifier attribute to the *Factory* class to be used to notify the observers. Modify the *notifyObservers()* method to use the notifier object instead of the observers themselves. Your code should look like the one below:

```
public void notifyObservers() {
    if (notifier != null) {
        notifier.notifyObservers();
    }
//  for (final Observer observer : getObservers()) {
//      observer.modelChanged();
//  }
}
```

Also modify the *addObserver()* and *removeObserver()* methods accordingly.

Creating a Local Notifier Class for the Notifier Interface

Create a class that implements the notifier interface created above. This class will be used when the factory model is being simulated on the same computer as the user interface. The class will hold the list of observers that was previously directly held by the Factory class so that it can notify them when its *notifyObservers()* method is called.

Set the factory notifier to be an instance of this default notifier implementation, run the simulator and check that it still works as before.

Creating a Remote Notifier Class

Next, create a *KafkaFactoryModelChangeNotifier* class implementing the notifier interface. This class will send modification events to a Kafka broker instead of notifying the observers directly. It will store a factory model object as an attribute whose value will be passed to the constructor. In the constructor, it will create a Kafka topic for this factory to which the data modification events will be published. Creating a topic can be achieved using the following code snippet, naming the topic using the factory model id as suffix to ensure topic names uniqueness.

```
TopicBuilder.name("simulation-" + factoryModel.getId()).build();
```

Also create a Kafka template attribute in the notifier class whose value will be passed to the constructor as shown in the code below:

```
private KafkaTemplate<String, Factory> simulationEventTemplate;
```

That object will be used later to send the events.

Next, implement the *notifyObservers()* method that will publish messages to the Kafka topic. For this, as shown in the code below, first construct a Spring *Message* object and set its payload as the factory object. Also set the topic name in the message header as shown in the code and use the Kafka template object to send the message. Check for any exception that could have occurred.

```
final Message<Factory> factoryMessage = MessageBuilder.withPayload(factoryModel)
    .setHeader(KafkaHeaders.TOPIC, "simulation-" + factoryModel.getId())
    .build();

final CompletableFuture<SendResult<String, Factory>> sendResult =
    simulationEventTemplate.send(factoryMessage);
sendResult.whenComplete((result, ex) -> {
    if (ex != null) {
        throw new RuntimeException(ex);
    }
});
```

Instantiating the Kafka Template Object

To obtain an instance of the *simulationEventTemplate* object used to send the messages to the different factory topics, we will use Spring IoC (Inversion of Control) design principle. However, this object internally uses a *JsonSerializer* object to serialize the data into JSON text. The class of such *JsonSerializer* object would normally be specified in the *application.properties* microservice configuration file, but as already explained, the *JsonSerializer* object needs to use a specific Jackson object mapper, custommized for the *Factory* model class. Hence, we cannot simply let Spring instantiate the serializer from a class given in the *application.properties* file.

Instead, you can use Java code in a configuration bean to instantiate the objects as it was done for the specific Jackson object mapper. The code below achieves this, where a default producer factory object is instantiated providing properties such as the Kafka server name and port, the class to be used to serialize the key parts of the key-value pair events and a *JsonSerializer* object to serialize factories giving it the proper object mapper to use.

```
@Configuration
public class SimulationServiceConfig {

    @Bean
    @Primary
    ObjectMapper objectMapper() {
        ...
        return objectMapper;
    }

    @Bean
    ProducerFactory<String, Factory> producerFactory(){
        final Map<String, Object> config = new HashMap<>();
        config.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
                   SimulationServiceUtils.BOOTSTRAP_SERVERS);

        final JsonSerializer<Factory> factorySerializer = new
            JsonSerializer<>(objectMapper());

        return new DefaultKafkaProducerFactory<>(config,
                                                  new StringSerializer(),
                                                  factorySerializer);
    }

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

Finally, to obtain the Kafka template object created by the service configuration bean above, add an attribute of this type to the Spring controller class of your microservice and as shown in the code below, annotate it with *@Autowired* to tell Spring to set a value for the attribute via dependency injection.

```
@Autowired
private KafkaTemplate<String, Factory> simulationEventTemplate;
```

Finally, modify the *startSimulation()* method of your Spring controller class so that for every factory model that is simulated, its notifier is set as an instance of the *KafkaFactoryModelChangeNotifier* class as shown in the code below:

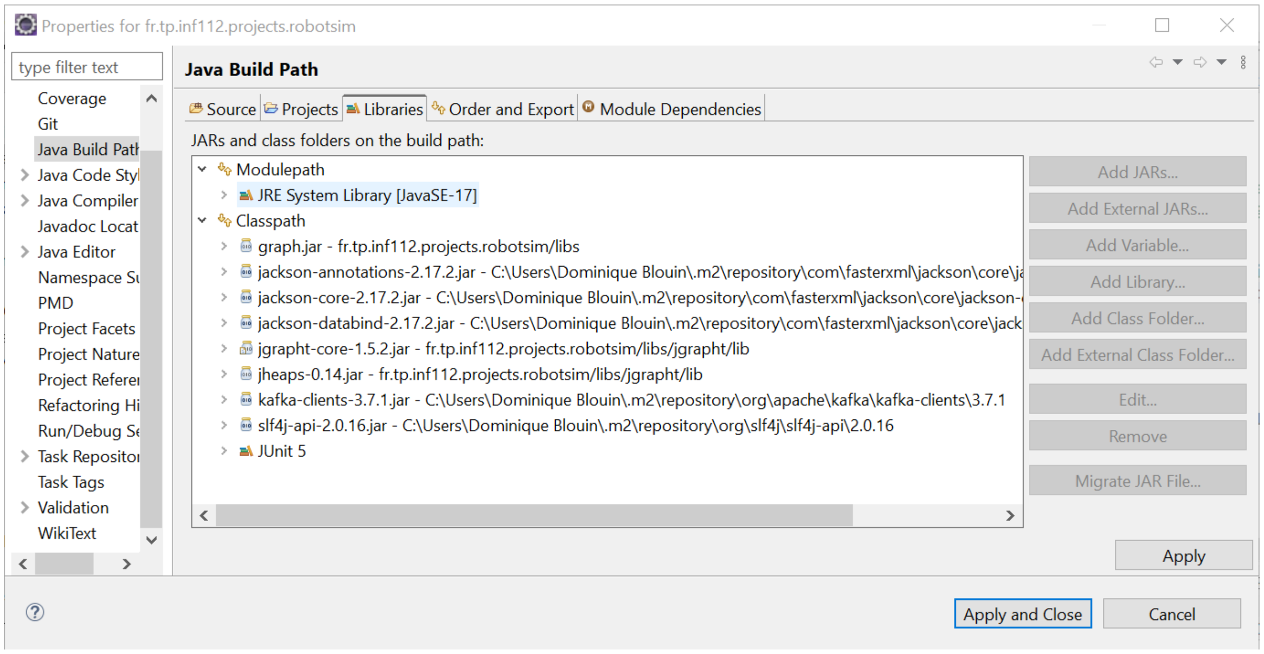
```
final FactoryModelChangedNotifier notifier = new
    KafkaFactoryModelChangeNotifier(factory, simulationEventTemplate);
factory.setNotifier(notifier);
```

Modifying the Remote Simulator Controller to Consume Simulation Events

Kafka events can be consumed from both an application deployed on a server or from a plain Java application such as the simulator viewer. The last part of this exercise consists of modifying the remote simulator controller to consume Kafka events instead of pooling the simulation microservice to obtain a fresh factory model and update the view.

Configuring the Simulator Project to Use Kafka

The first step consists of adding the Kafka client library to your simulator project, which is required to consume Kafka events. For this, ensure that the *kafka-clients* library has been added to the simulator project classpath as shown below:


Properties for fr.tp.inf112.projects.robotsim

Java Build Path

Source | Projects | Libraries | Order and Export | Module Dependencies

JARs and class folders on the build path:

```
Modulepath
    JRE System Library [JavaSE-17]
Classpath
    graph.jar - fr.tp.inf112.projects.robotsim/libs
    jackson-annotations-2.17.2.jar - C:\Users\Dominique Blouin\.m2\repository\com\fasterxml\jackson\core\ja
    jackson-core-2.17.2.jar - C:\Users\Dominique Blouin\.m2\repository\com\fasterxml\jackson\core\jackson-
    jackson-databind-2.17.2.jar - C:\Users\Dominique Blouin\.m2\repository\com\fasterxml\jackson\core\jack
    jgrapht-core-1.5.2.jar - fr.tp.inf112.projects.robotsim/libs/jgraph/lib
    jheaps-0.14.jar - fr.tp.inf112.projects.robotsim/libs/jgraph/lib
    kafka-clients-3.7.1.jar - C:\Users\Dominique Blouin\.m2\repository\org\apache\kafka\kafka-clients\3.7.1
    slf4j-api-2.0.16.jar - C:\Users\Dominique Blouin\.m2\repository\org\slf4j\slf4j-api\2.0.16
    JUnit 5
```

Creating a Simulation Event Consumer Class

The remote simulator controller will be modified to consume the Kafka simulation events that are produced by the simulation microservice. For this, first create a *FactorySimulationEventConsumer* class that will receive the events, extract the JSON text out of them, and send this text to the remote simulator controller who will use its Jackson object mapper to parse the text into a factory object. The factory attribute of the remote simulator controller will then be set to this new factory and the viewer will be notified that the data has changed so that it can refresh itself.

The *FactorySimulationEventConsumer* class will first initialize a Kafka consumer object used to read the events. As shown in the code below, it will provide to the consumer object a properties object specifying the Kafka broker host and the classes of the objects that will be used to deserialize both the keys and values of the received Kafka events. Finally, the consumer will be subscribed to the topic.

```java
private final KafkaConsumer<String, String> consumer;
private final RemoteSimulatorController controller;

public FactorySimulationEventConsumer(final RemoteSimulatorController controller) {
    this.controller = controller;

    final Properties props = SimulationServiceUtils.getDefaultConsumerProperties();
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
        StringDeserializer.class);
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
        StringDeserializer.class);

    this.consumer = new KafkaConsumer<>(props);
    final String topicName =
        SimulationTopicsUtils.getTopicName(controller.getCanvas());
    this.consumer.subscribe(Collections.singletonList(topicName));
}

public class SimulationServiceUtils {

    public static final String BOOTSTRAP_SERVERS = "localhost:9092";
    private static final String GROUP_ID = "Factory-Simulation-Group";
    private static final String AUTO_OFFSET_RESET = "earliest";
    private static final String TOPIC = "simulation-topic-";

    public static String getTopicName(final Factory factoryModel) {
        return TOPIC + factoryModel.getId();
    }

    public static Properties getDefaultConsumerProperties() {
        final Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        props.put(ConsumerConfig.GROUP_ID_CONFIG, GROUP_ID);
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, AUTO_OFFSET_RESET);

        return props;
    }
}
```

To read the event, the following method can be added to the event consumer class. The Kafka consumer takes care of pooling the Kafka broker to read the events. The value of the received consumer records will simply consist of the JSON text of the serialized factory object. This data is passed to the remote simulation controller who uses its Jackson object mapper to parse the factory object.

```
public void consumeMessages() {
    try {
        while (controller.isAnimationRunning()) {
            final ConsumerRecords<String, String> records =
                consumer.poll(Duration.ofMillis(100));

            for (final ConsumerRecord<String, String> record : records) {
                LOGGER.fine("Received JSON Factory text '" + record.value() + "'.");
                controller.setCanvas(record.value());
            }
        }
    }
    finally {
        consumer.close();
    }
}
```

Using the Event Consumer in the Remote Simulator Controller

Finally, update the remote simulator controller class so that it uses the event consumer class that you just developed to consume simulation events. Instead of pooling the microservice to read the simulating factory model at a given period, simply instantiate the *FactorySimulationEventConsumer* class and call its *consumeMessages()* method.

As already mentioned, the *setCanvas()* method of the controller called by the event consumer will take care of parsing the JSON text into a factory model object. It will then set this new object to the controller factory attribute and notify the views that the model data has changed. To implement such notification, an easy way consists of adding a local notifier attribute of the *LocalFactoryModelChangedNotifier* class to the controller class. Then, redefine the *addObserver()* and *removeObserver()* methods so that view is added to the list of observers of the notifier.

Testing that the Simulator still Works as Before

Finally, start Zookeeper, the Kafka broker, the persistence server, the simulation microservice and the simulator viewer and check that your event-based simulator works normally.