Kafka configuration for communication between 2 microservices

local environment, one kafka cluster and one broker

We are gonna focus on the feasibility and fees modules for this explanation

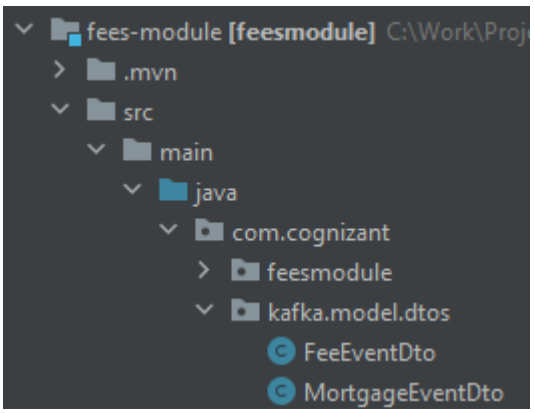
In this case, the feasibility module will act as a Producer and the fees module will be the Consumer.

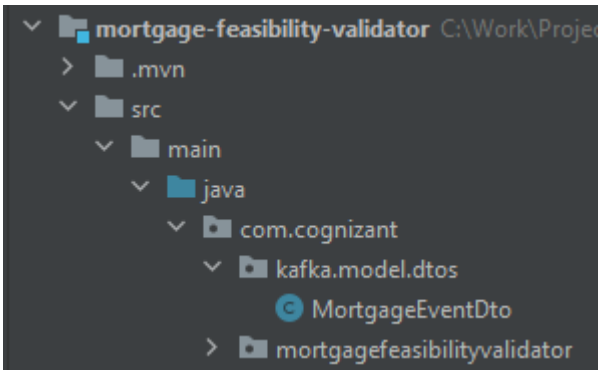
There are several **key issues** to take into account when trying to communicate two modules with a kafka topic:

- **Firstly**, we need to **specify the data** that is being sent through the topic. Both modules must agree on this data, and keep the same format and types at both ends. For example, the data we agreed on **before** implementing the producer and the consumer is called *MortgageEventDto*. Also, **both classes must have the same name and same parameters**:

```
public class MortgageEventDto {
    private String company;
    private UUID mortgageId;
    private BigDecimal homePrice;
    private MortgageStatus mortgageStatus;
}
```

- **Secondly**, the data sent through the kafka topic **must have the same package path** on both sides. As we can see in this example, both classes are in the package *src.main.java.com.cognizant.kafka.model.dtos*





- **Thirdly**, both modules must agree on the topic name where they will produce the messages to and consume from. In this case, we have agreed on the name *MORTGAGE_EVENTS* for the topic between feasibility and fees modules.

Docker-compose.yml

```
version: '2'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    ports:
      - 22181:2181
  kafka:
    image: confluentinc/cp-kafka:latest
    depends_on:
      - zookeeper
    ports:
      - 29092:29092
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092,PLAINTEXT_HOST://localhost:29092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

- To start an Apache Kafka server, first, we'd need to start a Zookeeper server. We can configure this dependency in the docker-compose.yml file, which will ensure that the Zookeeper server always starts before the Kafka server and stops after it.

- In this setup, our Zookeeper server is listening on port 2181 for the kafka service, which is defined within the same container setup. However, for any client running on the host, it'll be exposed on port 22181.

- Similarly, the Kafka service is exposed to the host applications through port 29092, but it is actually advertised on port 9092 within the container environment configured by the KAFKA_ADVERTISED_LISTENERS property, which is a comma-separated list of listeners with their host/ip and port. This is the metadata that’s passed back to clients.

- *KAFKA_LISTENER_SECURITY_PROTOCOL_MAP* defines key/value pairs for the security protocol to use, per listener name.

- Kafka listeners: https://rmoff.net/2018/08/02/kafka-listeners-explained/

Application.properties

Feasibility module:

```
spring.kafka.bootstrap-server=192.168.0.13:29092

kafka.topic.producer.mortgage.events=MORTGAGE_EVENTS
kafka.producer.group-id=MortgageModule
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=org.springframework.kafka.support.serializer.JsonSerializer
```

- These are the **basic configuration properties** that we will need to specify in our spring application in order for it to connect to our kafka topic, **as a producer**:
    - *spring.kafka.bootstrap-server* indicates where our kafka broker is. In this case, running a local configuration, we will have to specify **our IPv4 address** and the **same port as exposed in the docker-compose.yml** file. In this case, we are exposing port *29092*. If we would like to further detail this, we would specify this is a producer: *spring.kafka.producer.bootstrap-server.*
    - *kafka.topic.producer.mortgage.events* this is just a name we give for the topic name. This property will be the one we will use within our application to **configure the producer method**. In this case, our feasibility module acts as a **producer**, hence the *.producer* name within the property.

```
@Component
public class MortgageEventPublisherServiceImpl implements MortgageEventPublisherService {

    private KafkaTemplate<String, MortgageEventDto> kafkaTemplate;

    @Value("${kafka.topic.producer.mortgage.events}")
    private String MORTGAGE_EVENTS_TOPIC;

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

    @Override
    public void handleMortgageValidatedEvent(MortgageEventDto mortgageValidatedEvent) {
        kafkaTemplate.send(
                new ProducerRecord<>(MORTGAGE_EVENTS_TOPIC,  key: null, mortgageValidatedEvent)
        );
    }
}
```

- *kafka.producer.group-id* unique string that identifies the **producer group** to which this producer belongs.

- **Serializer properties** - These properties are mandatory when we need to send complex objects through the topic (not basic types), like Dtos. They tell spring how to serialize the message, with a key-value approach. It is **very important** to be aware if you are configuring a producer or a consumer at this point, since the properties for the producer are “SERIALIZERS” and the properties for consumers are “DESERIALIZERS”. In this case, we want the key to be serialized as a string, and the value as a Json object.

Fees module:

```
spring.kafka.consumer.bootstrap-servers=192.168.0.13:29092
kafka.topic.consumer.mortgage.events=MORTGAGE_EVENTS
kafka.consumer.group-id=FeesModule
spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.value-deserializer=org.springframework.kafka.support.serializer.JsonDeserializer
spring.kafka.consumer.properties.spring.json.trusted.packages=*
spring.kafka.consumer.properties.spring.json.value.default.type=com.cognizant.kafka.model.dtos.MortgageEventDto
spring.kafka.consumer.auto-offset-reset=latest
```

- These are the **basic configuration properties** that we will need to specify in our spring application in order for it to connect to our kafka topic, **as a consumer**:
    - *spring.kafka.consumer.bootstrap-server* indicates where our kafka broker is, and we are specifying we are working as a consumer. Just as in the producer, we are running a local configuration, so we are specifying **our IPv4 address** and the **same port as exposed in the docker-compose.yml** file, port *29092*.
    - *kafka.topic.consumer.mortgage.events* again, this is the topic name property. This property will be the one we will use within our application to **configure the consumer method**. In this case, our fees module acts as a **consumer**, hence the *.consumer* name within the property.

```
@Log4j2
@Component
public class MortgageEventsConsumer {

    private final FeesService feesService;

    public MortgageEventsConsumer(FeesService feesService) { this.feesService = feesService; }

    @KafkaListener(topics = "${kafka.topic.consumer.mortgage.events}", groupId = "${kafka.consumer.group-id}")
    public void receive(MortgageEventDto mortgageDto) throws NotAValidMortgageException, FeeRepositoryException {
        log.info("#### Receiving and processing mortgageDto: " + mortgageDto.toString());
        feesService.processFees(mortgageDto);
    }

}
```

    - *kafka.consumer.group-id* unique string that identifies the consumer group to which this consumer belongs.
    - **Deserializer properties** - These properties are mandatory when we are receiving complex objects through the topic (not basic types), like Dtos. They tell spring how to deserialize the message, with a key-value approach. Like I mentioned in the producer, It is **very important** to be aware if you are configuring a producer or a consumer at this point, since the properties for the

producer are “SERIALIZERS” and the properties for consumers are “DESERIALIZERS”. In this case, we want the key to be serialized as a string, and the value as a Json object, and since we are in a consumer, we are DEserializing the object.

- *spring.kafka.consumer.properties.spring.json.trusted.packages* - We need to specify to spring that whatever **objects we are receiving through the topic come from trusted packages** / sources. Otherwise, an error will be thrown. In this case we decided to trust all packages, but you can restrict this option.
- *spring.kafka.consumer.properties.spring.json.default.type* - where we specify what is the **default message object we are expecting from the topic**
- *spring.kafka.consumer.auto-offset-reset* - here we are specifying what **type of offset** we want: **earliest or latest**. *Earliest* is when we want to read all the messages from the queue, *latest* is when we only want new messages. **WARNING:** we noted this property is not working 100% all the time, so our recommendation while developing is to always start your tests with a fresh topic. To do so, you must **delete the docker containers related to kafka and zookeeper and start them again**.