Example: Topics & Partitions

In this lesson, we'll study how our example is divided into topics and partitions.

We'll cover the following

Technical parameters of the partitions and topics

The topic `order` contains the order records. Docker Compose configures the Kafka Docker container based on the environment variable `KAFKA_CREATE_TOPICS` in the file `docker-compose.yml` in such a way as to create the topic `order`.

The topic `order` is divided into **five partitions**, as a greater number of partitions allows for more concurrency. In the example scenario, it is not important to have a high degree of concurrency. More partitions require more file handles on the server and more memory on the client. When a Kafka node fails, it might be necessary to choose a new leader for each partition. This also takes longer when more partitions exist.

This **argues for a lower number of partitions** as used in the example in order to save resources. The number of partitions in a topic can still be changed after creating a topic.

However, in that case, the mapping of records to partitions will change. This can cause problems because then the assignment of records to consumers is not unambiguous anymore. Therefore, **the number of partitions should be chosen sufficiently high from the start**.

No replication in the example

For a production environment, a replication across multiple servers is necessary to compensate for the failure of individual servers. For a demo, the level of complexity required is not needed, so that only one Kafka node is running.

Producers

The order microservice has to send the information about the order to the other microservices. To do so, the microservice uses the `KafkaTemplate`. This class from the Spring Kafka framework encapsulates the producer API and facilitates the sending of records. Only the method `send()` has to be called. This is shown in the code piece from the class `OrderService` in the listing.

```
public Order order(Order order) {
  if (order.getNumberOfLines() == 0) {
    throw new IllegalArgumentException("No order lines!");
  }
  order.setUpdated(new Date());
  Order result = orderRepository.save(order);
  fireOrderCreatedEvent(order);
  return result;
}

private void fireOrderCreatedEvent(Order order) {
  kafkaTemplate.send("order", order.getId() + "created", order);
}
```

Behind the scenes, Spring Kafka converts the Java objects to JSON data with the help of the Jackson library. Additional configurations such as the configuration of the JSON serialization can be found in the file `application.properties` in the Java project. In `docker-compose.yml`, environment variables for Docker compose are defined, which are evaluated by Spring Kafka; these are the Kafka host and the port. Thus, with a change to `docker-compose.yml`, the configuration of the Docker container with the Kafka server can be changed and the producers can be adapted in such a way that they use the new Kafka host.

Consumers

The consumers are also configured in `docker-compose.yml` and with the `application.properties` in the Java project. Spring Boot and Spring Kafka automatically build an infrastructure with multiple threads that read and process records. In the code, only a method is annotated with `@KafkaListener(topics = "order")` in the class `OrderKafkaListener`.

```
@KafkaListener(topics = "order")
public void order(Invoice invoice, Acknowledgment acknowledgment) {
  log.info("Received invoice " + invoice.getId());
  invoiceService.generateInvoice(invoice);
  acknowledgment.acknowledge();
```

```
    acknowledgment.acknowledge();
}
```

One parameter of the method is a Java object that contains the data from the JSON in the Kafka record. During deserialization the data conversion takes place.

Invoicing and shipping read only the data they need; the remaining information is ignored. Of course, in a real system, it is possible to implement more complex logic rather than just filtering the relevant fields.

The other parameter of the method is of the type `Acknowledgement`. This allows the consumer to commit the record. When an error occurs, the code can prevent the acknowledgement. In this case, the record would be processed again.

The **data processing in the Kafka example is idempotent**. When a record is supposed to be processed, first the database is queried for data stemming from a previous processing of the same record. If the microservice finds such data, the record is obviously a duplicate and is not processed a second time.

```
version: '3'
services:
  zookeeper:
    image: wurstmeister/zookeeper:3.4.6
  kafka:
    image: wurstmeister/kafka:2.12-2.5.0
    links:
     - zookeeper
    environment:
      KAFKA_ADVERTISED_HOST_NAME: kafka
      KAFKA_ADVERTISED_PORT: 9092
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_CREATE_TOPICS: "order:5:1"
  apache:
    image: educative1/mapi_mskafka_apache
    links:
     - order
     - shipping
     - invoicing
    ports:
     - "8080:80"
  postgres:
    image: educative1/mapi_mskafka_postgres
    environment:
      POSTGRES_PASSWORD: dbpass
      POSTGRES_USER: dbuser
  order:
    image: educative1/mapi_mskafka_order
    links:
     - kafka
     - postgres
    environment:
      SPRING_KAFKA_BOOTSTRAP_SERVERS: kafka:9092
  shipping:
    image: educative1/mapi_mskafka_shipping
    links:
     - kafka
     - postgres
    environment:
      SPRING_KAFKA_BOOTSTRAP_SERVERS: kafka:9092
  invoicing:
    image: educative1/mapi_mskafka_invoicing
    links:
     - kafka
     - postgres
    environment:
      SPRING_KAFKA_BOOTSTRAP_SERVERS: kafka:9092
```

Consumer groups

The setting `spring.kafka.consumer.group-id` in the file `application.properties` in the projects `microservice-kafka-invoicing` and `microservice-kafka-shipping` defines the consumer group to which the microservices belong. All instances of shipping or invoicing each form a consumer group. Exactly one instance of the shipping or invoicing microservice receives a record. This ensures that an order is not processed in parallel by multiple instances.

Using `docker-compose up --scale shipping=2`, more instances of the shipping microservice can be started. If you look at the log output of an instance with `docker logs -f mskafka_shipping_1`, you will see which partitions are assigned to this instance and that the assignment changes when additional instances are started. Similarly, you can see which instance processes a record when a new order is generated.

It is also possible to have a look at the content of the topic. To do so, you have to start a shell on the Kafka container with `docker exec -it mskafka_kafka_1 /bin/sh`. The command `kafka-console-consumer.sh --bootstrap-server kafka:9092 --topic order --from-beginning` shows the complete content of the topic. Because all the microservices belong to a consumer group and commit the processed records, they receive only the new records. However, a new consumer group would indeed process all records again.

Try these in the Kafka coding environment above!

QUI
Z

1

What are the arguments for a **lower** number of partitions?

☐ A) Lesser memory consumed on the client.

☐ B) It takes less time to find a leader for each partition if the node fails.

☐ C) More opportunity for concurrency.

Submit Answer

Question 1 of 3
0 attempted

Reset Quiz

In the next lesson, we'll discuss Kafka-based testing and other data formats.

← Back
Example: Technical Structure & Live A...

Next →
Example: Testing & Other Data Formats

✓ Mark as Completed

Report an Issue