Example: Introduction

In this lesson, we'll introduce a Kafka based coding example.

WE'LL COVER THE FOLLOWING

- Introduction
- Data model for the communication
- Domain-driven design and strategic design
- Implementation of the communication

Introduction

The example in this section is based on the example for events from Events (see the drawing below).

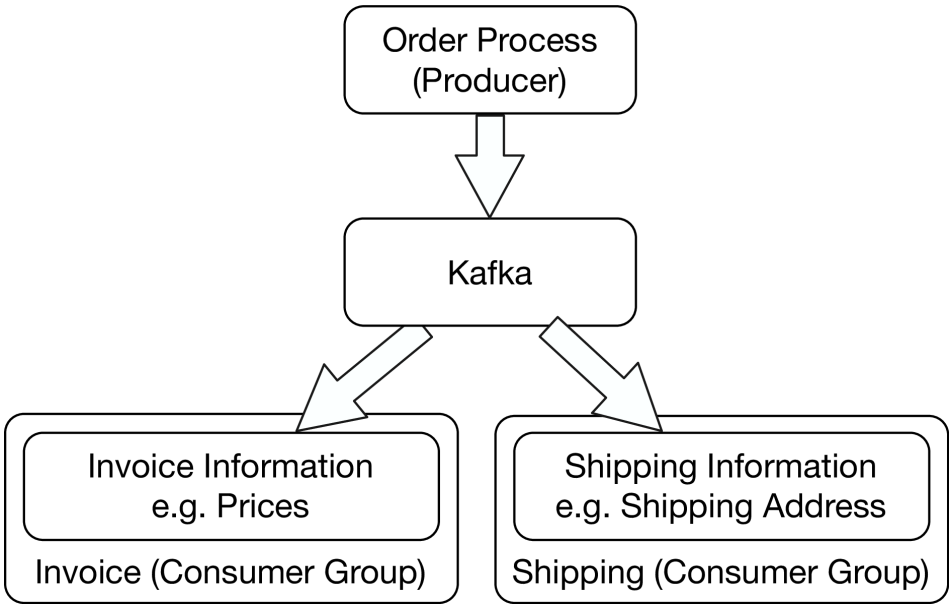
The microservice **microservice-kafka-order** is responsible for creating the order as it sends the orders to a Kafka topic. Therefore, the **microservice-kafka-order** is the **producer**.

Two microservices read the orders; the microservice **microservice-kafka-invoicing** issues an invoice for an order, and the microservice **microservice-kafka-shipping** delivers the ordered goods.

The two microservices are organized in **two consumer groups**. Each record is just consumed and processed by:

- one instance of the microservice **microservice-kafka-invoicing**
- one instance of **microservice-kafka-shipping**.

Data model for the communication



Example for Kafka

The two microservices **microservice-kafka-invoicing** and **microservice-kafka-shipping** require different information:

- The invoicing microservice requires the billing address and information about the prices of the ordered goods.
- The shipping microservice needs the delivery address but does not require prices.

Both microservices read the necessary information from the same Kafka topic and records. The only difference is what data they read from the records. Technically, this is easily done because the data about the orders is delivered as JSON. Thus, the two microservices can just ignore unneeded fields.

Domain-driven design and strategic design

In the example, the communication and conversion of the data are deliberately kept simple.

They implement the DDD pattern **published language**. There is a standardized data format from which all systems read the necessary data. With a large number of communication partners, the data model can become confusingly large.

In such a case, **customer/supplier** could be used. The teams responsible for shipping and invoicing dictate to the order team what data an order must

contain to allow shipping and invoicing. The order team then provides the necessary data. The interfaces can even be separated, but this seems to be a step backwards.

After all, **published language** offers a common data structure that all microservices can use. In reality, however, it is a mixture of the two data sets that are needed by shipping, invoicing, and ordering.

Separating this one model into two models for the communication between invoicing and order or delivery and order makes it obvious which data is relevant for which microservice and makes it easier to assess the impact of changes. The two data models can be further developed independently of each other. This serves the goal of microservices to **make software easier to modify and to limit the effects of a change**.

The patterns **customer/supplier** and **published language** originate from the strategic design part of the domain-driven design (DDD). The lesson, Events, also discusses what data should be contained in an event.

Implementation of the communication

Technically, communication is implemented as follows. The Java class `Order` from the project *microservice-kafka-order* is serialized in JSON. The classes `Invoice` from the project *microservice-kafka-invoicing* and `Shipping` from the project *microservice-kafka-shipping* get their data from this JSON. They ignore fields unrequired in the systems. The only exceptions are the `orderLines` from `Order`, which in `shipping` are called `shippingLines` and in `Invoice` are called `invoiceLine`. For the conversion, there is a `setOrderLine()` method in the two classes to deserialize the data from JSON.

QUIZ

1 What pattern for communication is used in the given example?

In the next lesson, we'll look at the data aspects of this example.