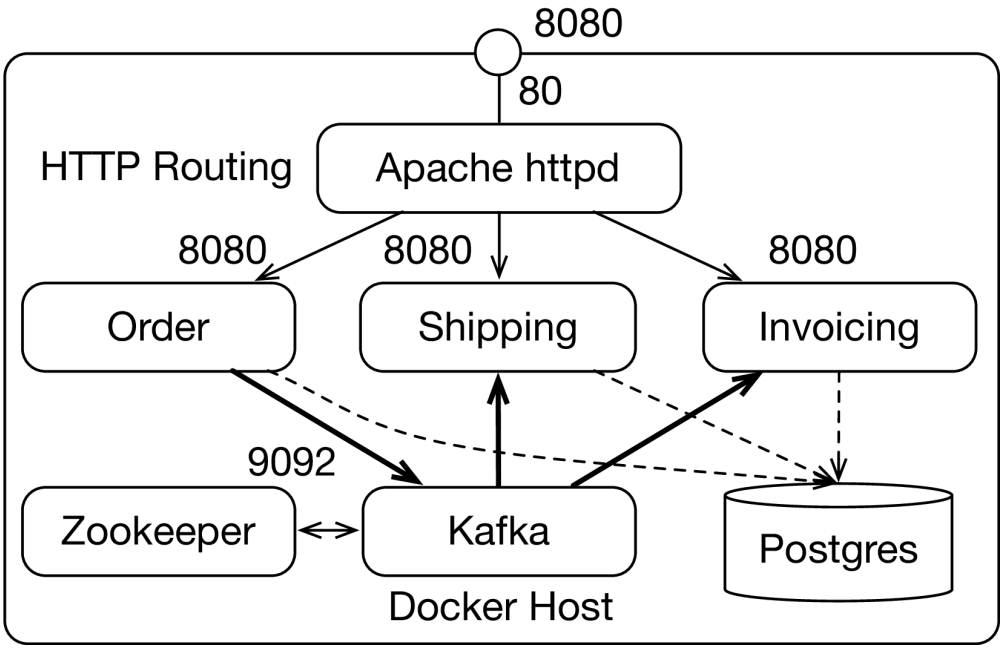
Example: Technical Structure & Live App

In this lesson, we'll look at a real live example of Kafka.

WE'LL COVER THE FOLLOWING

Technical structure



Overview of the Kafka System

The drawing above shows how the example is structured technically.

Apache httpd

The **Apache httpd** distributes incoming HTTP requests. Thus, there can be multiple instances of each microservice. This is useful for showing the distribution of records to multiple consumers.

In addition, only the Apache httpd server is accessible from the outside. The other microservices can be contacted only from inside the Docker network.

Zookeeper

Zookeeper serves to coordinate the Kafka instances and stores, information about the distribution of topics and partitions. The example uses the image at https://hub.docker.com/r/wurstmeister/zookeeper/.

Kafka instance

The **Kafka instance** ensures the communication between the microservices. The order microservice sends the orders to the shipping and invoicing microservices. The example uses the Kafka image at https://hub.docker.com/r/wurstmeister/kafka/.

Postgres database

Finally, the order, shipping, and invoicing microservices use the same *Postgres database*. Within the database instance, each microservice has its own separate database schema.

Thus, the microservices are **completely independent** in regard to their database schemas. At the same time, one database instance can be enough to run all the microservices.

The alternative would be to **give each microservice its own database instance**. However, this would increase the number of Docker containers and would make the demo more complex.

Running the example locally

The example can be found at https://github.com/ewolff/microservice-kafka. To

start the example, you have to first download the code with `git clone https://github.com/ewolff/microservice-kafka.git`. Afterwards, the command

`./mvnw clean package` (macOS, Linux) or `mvnw.cmd clean package` (Windows) has to be executed in the directory `microservice-kafka` to compile the code. See the appendix for more details on Maven and how to troubleshoot the build. Then `docker-compose build` has to be executed in the directory `docker` to generate the Docker images and `docker-compose up -d` for starting the environment. See the appendix for more details on Docker, Docker Compose and how to troubleshoot them. The Apache httpd load balancer is available at port 8080. If Docker runs locally, it can be found at http://localhost:8080/. From there, you can use the order microservice to create an order. The microservices shipping and invoicing should display the order data after some time.

At https://github.com/ewolff/microservice-kafka/blob/master/HOW-TO-RUN.md extensive documentation can be found that explains installation and instructions for starting the example step by step.

Key for the records

Kafka transfers the data in records where each record contains an order. The key of the record is the order ID with the extension `created`, for example, `1created`. Just the order ID would not be enough. In case of a log compaction, all records with an identical key are deleted except for the last record. There can be different records for one order.

One record can be a result of the generation of a new order, and other records might be results of the different updates. Thus, the key should contain more than the order ID to keep all records belonging to an order during log compaction. When the key corresponds to the order ID, only the last record would be left after a log compaction.

However, this approach has the **disadvantage that records belonging to one order can end up in different partitions** and with different consumers because they have different keys. This means that, for example, records for the same order can be processed in parallel, which can cause errors.

Implementing a custom partitioner

To solve this problem, a function has to be implemented which assigns **all**

To solve this problem, a function has to be implemented which assigns **all records for one order to one partition**. A partition is processed by a single consumer, and the sequence of the messages within a partition is guaranteed. Thus, it is ensured that all messages located in the same partition for one order are processed by the same consumer in the correct sequence.

This function is called a partitioner. Therefore, it is possible to write custom code for the distribution of records onto the partitions. This allows a producer to write all records that belong together from a domain perspective into the same partition and to have them processed by the same consumer although they have different keys.

Sending all information about the order in the record

A possible alternative would be to **use only the order ID as the key**. To avoid the problem with log compaction, it is possible to send the complete state of the order along with each record so that a consumer can reconstruct its state from the data in Kafka, although only the last record for an order remains after log compaction.

However, this requires a data model that contains all the data all consumers need. It takes a lot of effort to design such a data model, besides being complicated and difficult to maintain. It also contradicts the bounded context pattern somewhat, even though it can be considered a published language.

QUIZ

1 Why did we not create separate database instances for each microservice in the example above?

In the next lesson, we'll look at topics and partitions.