Microservices

A Practical Guide
2nd Edition

Principles, Concepts, and Recipes

Eberhard Wolff

Microservices - A Practical Guide

Principles, Concepts, and Recipes

Eberhard Wolff

This book is for sale at http://leanpub.com/practical-microservices

This version was published on 2021-04-21

Contents

Introduction to the Sample

Thank you for your interest in the book "Microservices - A Practical Guide" and for downloading this sample!

The sample contains the introduction to show you the goal of the book and the approach the book takes.

The books is divided into three parts. The sample contains the introductions to each part and the full table of content to provide an overview of all the subjects that the book covers.

Chapter 11 is an example of a chapter of the book. It covers Kafka and shows the approach the book takes with regards to explaining the technologies.

The book is a translation from German[1].

You can find a list of the demos for the book at http://ewolff.com/microservices-demos.html . The free Microservices Recipes[2] eBook also provides an overview of these demos.

[1]http://microservices-praxisbuch.de/
[2]https://leanpub.com/microservices-recipes/overview

0 Introduction

Microservices are one of the most important software architecture trends. A number of detailed guides to microservices are already currently available, among them the microservices-book[3] written by the author of this volume. So why do we need still another book on microservices?

It is one thing to define an architecture, quite another to implement it. This book presents technologies for the implementation of microservices and highlights the associated benefits and disadvantages.

The focus rests specifically on technologies for entire microservices systems. Each individual microservice can be implemented using different technologies. Therefore, the technological decisions for frameworks for individual microservices are not as important as the decisions at the level of the overall system. For individual microservices, the decision for a framework can be quite easily revised. However, technologies chosen for the overall system are difficult to change.

This means, compared to the microservices-book,[4] this book talks primarily about technologies. This does discuss architecture and reasons for or against microservices, but only briefly.

Basic Principles

To become familiar with microservices, an introduction into microservices-based architectures and their benefits, disadvantages, and variations is essential. However, this book explains the basic principles only to the extent required for understanding the practical implementations. A more complete discussion is part of the microservices-book[5].

Concepts

Microservices require solutions for different challenges. Among those are concepts for integration (*frontend integration, synchronous and asynchronous microservices*) and for operation (*monitoring, log analysis, tracing*). Microservices platforms such as *PaaS* or *Kubernetes* represent exhaustive solutions for the operation of microservices.

Recipes

The book uses recipes as a metaphor for the technologies, which can be used to implement the different concepts. Each approach shares a number of features with a recipe.

[3] http://microservices-book.com
[4] http://microservices-book.com
[5] http://microservices-book.com

- Each recipe is described in *practical* terms, including an exemplary technical implementation. The most important aspect of the examples is their *simplicity*. Each example can be easily followed, extended, and modified.
- The book provides the reader with a *plethora of recipes*. The readers have to *select* a specific recipe from this collection for their projects, akin to a cook who has to select a recipe for her or his menu. The book shows different options. In practice, nearly every project has to be dealt with differently. The recipes build the basis for this.
- *Variations* exit for each recipe. After all, a recipe can be cooked in many different ways. This is also true for the technologies described in this book. Sometimes the variations are very simple, so that they can be immediately implemented as *experiments* in an executable example.

Each recipe includes an associated *executable example* based on a concrete technology. The examples can be run individually; they are not based on each other. This allows the readers to concentrate on the recipes that are interesting and useful for them and to skip the examples that are less relevant for their work.

In this manner, the book provides an easy *access* for obtaining an *overview* of the relevant technologies, and thereby enables the readers to select a suitable technology stack. Subsequently, the readers can use the links supplied in the book to acquire in depth knowledge about the relevant technologies.

Source Code

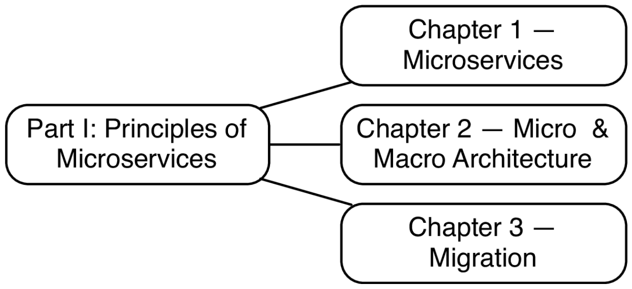
Sample code is provided for almost all the technologies presented in this book. If the reader wants to really understand the technologies, it makes sense to browse the code. It also makes sense to look at the code to understand how the concepts are actually implemented. The reader might even want to have the source right next to the book to read both code and this book.

0.1 Structure of the Book

This book consists of three parts.

Part I — Architecture Basics

Part I introduces the basic principles of a microservices-based architecture.

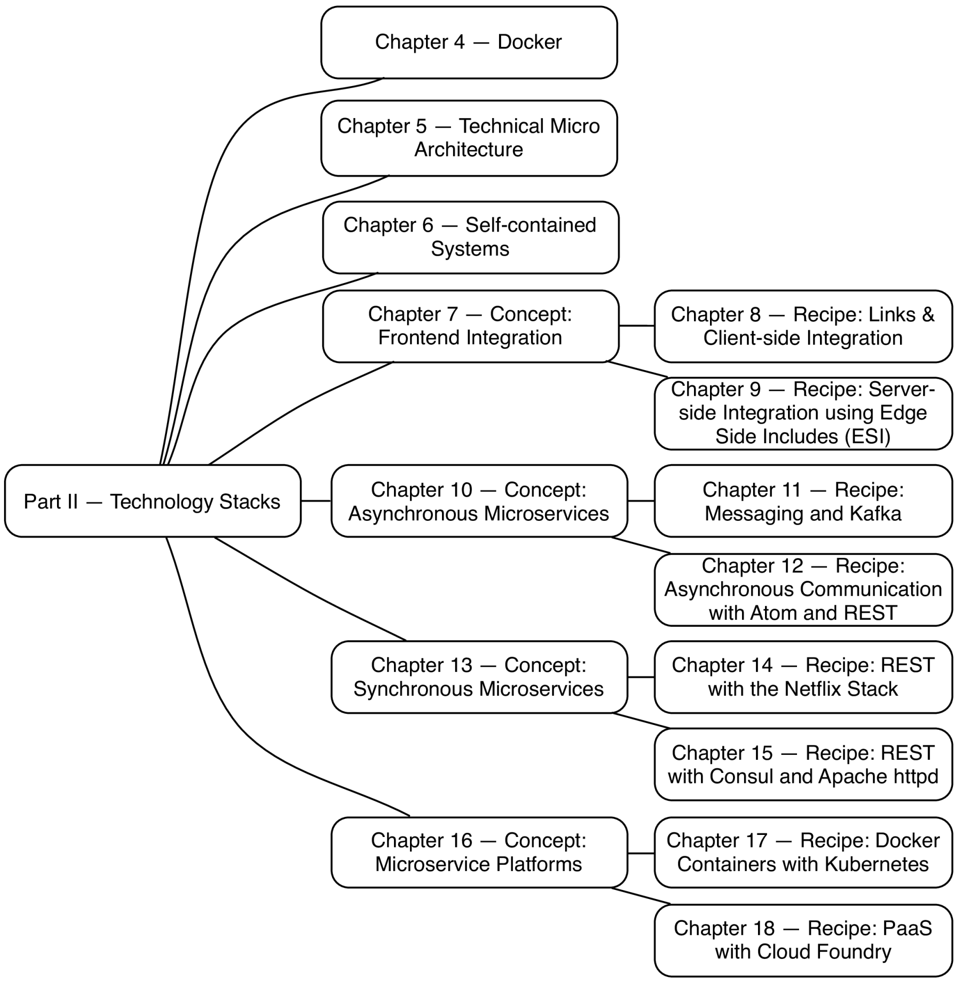

Part I: Principles of Microservices
Chapter 1 — Microservices
Chapter 2 — Micro & Macro Architecture
Chapter 3 — Migration

Overview of Part I

- Chapter 1 defines the term *microservice*.
- A microservices architecture has two levels: micro and macro architecture. They represent global and local decisions as explained in chapter 2.
- Often, old systems are migrated into microservices, a topic covered in chapter 3.

Part II — Technology Stacks

Technology stacks are the focus of part II.


Chapter 4 — Docker
Chapter 5 — Technical Micro Architecture
Chapter 6 — Self-contained Systems
Chapter 7 — Concept: Frontend Integration
Chapter 8 — Recipe: Links & Client-side Integration
Chapter 9 — Recipe: Server-side Integration using Edge Side Includes (ESI)
Part II — Technology Stacks
Chapter 10 — Concept: Asynchronous Microservices
Chapter 11 — Recipe: Messaging and Kafka
Chapter 12 — Recipe: Asynchronous Communication with Atom and REST
Chapter 13 — Concept: Synchronous Microservices
Chapter 14 — Recipe: REST with the Netflix Stack
Chapter 15 — Recipe: REST with Consul and Apache httpd
Chapter 16 — Concept: Microservice Platforms
Chapter 17 — Recipe: Docker Containers with Kubernetes
Chapter 18 — Recipe: PaaS with Cloud Foundry

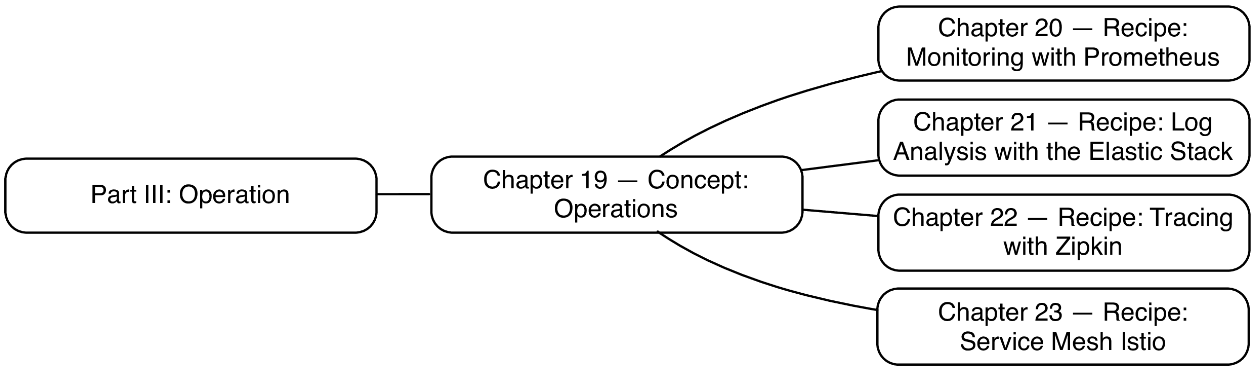
Overview of Part II

- *Docker* serves as basis for many microservices architectures (chapter 4). It facilitates the roll-out of software and the operation of the services.
- The *technical micro architecture* (chapter 5) describes technologies for implementing microservices.
- Chapter 6 explains *Self-contained Systems (SCS)* as an especially useful approach for microservices. It focuses on microservices that include a UI as well as logic.
- One possibility for integration in particular for SCS is *integrating at the web frontend* (chapter 7). Frontend integration results in a loose coupling between the microservices and a high degree of flexibility.

- The recipe for web frontend integration presented in chapter 8 capitalizes on *links* and *JavaScript* for dynamic content loading. This approach is easy to implement and utilizes well-established web technologies.
- On the server, integration can be achieved with *ESI (Edge Side Includes)* (chapter 9). ESI is implemented in web caches so that the system can attain high performance and reliability.
- The concept of *asynchronous communication* is the focus of chapter 10. Asynchronous communication improves reliability and decouples the systems.
- *Apache Kafka* is an example for an asynchronous technology (chapter 11) for sending messages. Kafka can save messages permanently and thereby enables a different approach to asynchronous processing.
- An alternative for asynchronous communication is *Atom* (chapter 12). Atom uses a REST infrastructure and thus is very easy to implement and operate.
- Chapter 13 illustrates how to implement *synchronous microservices*. Synchronous communication between microservices is often used in practice although this approach can pose challenges in regards to response times and reliability.
- The *Netflix Stack* (chapter 14) offers Eureka for service discovery, Ribbon for load balancing, Hystrix for resilience, and Zuul for routing. The Netflix Stack is especially widely used in the Java community.
- *Consul* (chapter 15) is an alternative option for service discovery. Consul contains numerous features and can be used with a broad spectrum of technologies.
- Chapter 16 explains the concept of *microservices platforms*, which support operation and communication of microservices.
- The *Kubernetes* (chapter 17) infrastructure can be used as a microservices platform and is able to execute Docker containers, as well as having solutions for service discovery and load balancing. The microservice remains independent of this infrastructure.
- *PaaS (Platform as a Service)* is another infrastructure that can be used as a microservices platform (chapter 18). Cloud Foundry is used as an example. Cloud Foundry is very flexible and can be run in your own computing center as well as in the public cloud.

Part III — Operation

It is a huge challenge to ensure the *operation* of a plethora of microservices. Part III discusses possible recipes that address this challenge.


Part III: Operation
Chapter 19 — Concept: Operations
Chapter 20 — Recipe: Monitoring with Prometheus
Chapter 21 — Recipe: Log Analysis with the Elastic Stack
Chapter 22 — Recipe: Tracing with Zipkin
Chapter 23 — Recipe: Service Mesh Istio

Overview of Part III

- Chapter 19 explains *basic principles* of operation, and why it is so hard to operate microservices.
- Chapter 20 deals with *monitoring* and introduces the tool Prometheus. Prometheus supports multi-dimensional data structures and can analyze metrics even of numerous microservice instances.
- Chapter 21 concentrates on the *analysis of log data*. The Elastic Stack is presented as a concrete tool. This stack is very popular and represents a good basis for analyzing large amounts of log data.
- *Tracing* allows one to trace calls between microservices (chapter 22), often done with the help of Zipkin. Zipkin supports different platforms and represents a de facto standard for tracing.
- *Service meshes* add proxies to the network traffic between microservices. This enables them to support monitoring, log analysis, tracing and other features such as resilience or security. Chapter 23 explains Istio as an example of a service mesh.

Conclusion and Appendices

At the end of the book, chapter 24 offers an *outlook*.

The appendices explain the software installation (appendix A), how to use the build tool Maven (appendix B), and also Docker and Docker Compose (appendix C), which you can use to run the environments for the examples.

0.2 Target Group

This book explains basic principles and technical aspects of microservices. Thus, it is interesting for different audiences.

- For *developers*, part II offers a guideline for selecting a suitable technology stack. The example projects serve as basis for learning the foundations of the technologies. The microservices contained in the example projects are written in Java using the Spring Framework. However, the technologies used in the examples serve to integrate microservices. So additional microservices

can be written in different languages. Part III completes the book by including the topic of operation that becomes more and more important for developers. Part I explains the basic principles of architecture concepts.
- For *architects*, part I contains fundamental knowledge about microservices. Part II and III present practical recipes and technologies for implementing microservices architectures. With these topics, this book goes much more into depth than other books that just focus on architecture, but do not cover technologies.
- For experts in *DevOps* and *operations*, the recipes in part III represent a sound basis for a technological evaluation of operational aspects such as log analysis, monitoring, and tracing of microservices. Part II introduces technologies for deployment such as Docker, Kubernetes, or Cloud Foundry that also solve some operational challenges. Part I provides background about the concepts behind the microservices architecture approach.
- *Managers* are presented with an overview of the advantages and specific challenges of the microservices architecture approach in part I. If they are interested in technical details, they will benefit from reading part II and III .

0.3 Prior Knowledge

The book assumes the reader to have basic knowledge of software architecture and software development. All practical examples are documented in such a way that they can be executed with very little prior knowledge. This book focuses on technologies that can be employed for microservices using different programming languages. However, the examples are written in Java using the Spring Boot and Spring Cloud frameworks so that changes to the code require knowledge of Java.

0.4 Quick Start

This book focuses first of all on introducing technologies. An example system is presented for each technology in each chapter. In addition, the quick start allows the reader to rapidly gain practical experience with the different technologies and to understand how they work with the help of the examples.

- First, the necessary software has to be *installed* on the computer. The installation is described in appendix A.
- *Maven* handles the build of the examples. Appendix B explains how to use Maven.
- All the examples use *Docker* and *Docker Compose*. Appendix C describes the most important commands for Docker and Docker Compose.

For the Maven-based build and also for Docker and Docker Compose, the chapters contain basic instructions and advice on troubleshooting.

The examples are explained in the following sections:

Concept	Recipe	Section
Frontend Integration	Links and Client-side Integration	8.2
Frontend Integration	Edge Side Includes (ESI)	9.2
Asynchronous Microservices	Kafka	11.4
Asynchronous Microservices	REST and Atom	12.2
Synchronous Microservices	Netflix Stack	14.1
Synchronous Microservices	Consul and Apache httpd	15.1
Microservices Platform	Kubernetes	17.3
Microservices Platform	Cloud Foundry	18.3
Operation	Monitoring with Prometheus	20.4
Operation	Log Analysis with Elastic Stack	21.3
Operation	Tracing with Zipkin	22.2
Operation	Service Mesh with Istio	23.2

All projects are available on GitHub. The projects always contain a `HOW-TO-RUN.md` file showing step-by-step instructions on how the demos can be installed and started.

The examples are independent of each other, so you can start with any one of them.

0.5 Acknowledgements

I would like to thank everybody who discussed microservices with me, who asked me about them, or worked with me. Unfortunately, these are far too numerous to name them all individually. The exchange of ideas is enormously helpful and also fun!

Many of the ideas and also their implementation would not have been possible without my colleagues at INNOQ. I would especially like to thank Alexander Heusingfeld, Christian Stettler, Christine Koppelt, Daniel Westheide, Gerald Preissler, Hanna Prinz, Jörg Müller, Lucas Dohmen, Marc Giersch, Michael Simons, Michael Vitz, Philipp Neugebauer, Simon Kölsch, Sophie Kuna, Stefan Lauer, and Tammo van Lessen.

Also Merten Driemeyer and Olcay Tümce provided important feedback.

Finally, I would like to thank my friends and family, whom I often neglected while writing this book – especially my wife. She also did the translation into English.

Of course, my thanks go also to the people who developed the technologies which I introduce in this book and thereby created the foundation for microservices.

I also would like to thank the developers of the tools of https://www.softcover.io/ and Leanpub.

Last but not least, I would like to thank my publisher dpunkt.verlag and René Schönfeldt, who professionally supported me during the creation of the German version of this book.

0.6 Website

You can find the website accompanying this book at http://practical-microservices.com/. It contains links to the examples and also errata.

Part I: Principles of Microservices

The first part of this book introduces the fundamental ideas underlying microservice-based architectures.

Microservices

Chapter 1 explains the basics about *microservices*. What are microservices? What benefits and disadvantages do microservices architectures have?

Micro and Macro Architecture

Microservices offer a lot of freedom. Still, some decisions have to be made that affect all microservices of a system. Chapter 2 introduces the concept of *micro and macro architecture*. Micro architecture comprises all decisions that can be made individually for each microservice. Macro architecture, on the other hand, comprises the decisions that concern all microservices. In addition to the components of micro and macro architecture, this book also explains who designs macro architecture.

Migration

Most microservice projects serve to migrate an existing system into a microservices architecture. Therefore, chapter 3 presents possible objectives for a *migration* and introduces different strategies for migrations.

Part II: Technology Stacks

The second part of this book deals with recipes for technologies that can be used to implement microservices.

Docker

First, chapter 4 provides an introduction to *Docker*. Docker offers a good foundation for the implementation of microservices and is the basis for the examples in this book. Reading this chapter is therefore critical for understanding the examples in the later chapters.

Technical Micro Architecture

Chapter 2 introduced the concepts of micro and macro architecture. Micro architecture comprises the decisions that can be made differently for each microservice. Macro architecture represents the decisions that have to be uniform for all microservices. Chapter 5 discusses technical possibilities for the implementation of the micro architecture of a microservice.

Self-contained Systems

Chapter 6 describes *Self-contained Systems*. They are a collection of best practices for microservices architectures with a focus on independence and web applications. In addition to benefits and disadvantages, this chapter discusses possible variations of this idea.

SCS always include a web UI and rely on frontend integration. Therefore, discussing this approach right before frontend integration is explained in more detail makes sense to motivate this approach for integration.

Frontend Integration

One possibility for the integration of microservices is *frontend integration*, which chapter 7 explains. A concrete technical implementation with *links* and *client-side integration with JavaScript* is shown in chapter 8. Chapter 9 describes *Edge Side Includes (ESI)* that provide UI integration on the server.

Asynchronous Microservices

Chapter 10 presents *asynchronous microservices*. Chapter 11 introduces *Apache Kafka* as an example of a middleware that can be used to implement asynchronous microservices. *Atom*, the topic of chapter 12, is a data format that can be useful for asynchronous communication via REST.

Synchronous Microservices

Chapter 13 explains synchronous microservices. The *Netflix stack* discussed in chapter 14 is a way to implement synchronous microservices. The stack includes solutions for load balancing, service discovery, and resilience. Chapter 15 shows *Consul* as an alternative for service discovery and introduces *Apache httpd* for load balancing.

Microservices Platforms

Chapter 16 discusses *microservice platforms*, which provide support for synchronous communication and also a runtime environment for deployment and operation. Chapter 17 demonstrates how synchronous microservices can be implemented with *Kubernetes*. Kubernetes serves as a runtime environment for Docker containers and has, among others, features for load balancing and service discovery.

Chapter 18 describes *PaaS* (Platform as a Service). A PaaS makes it possible to leave the operation and deployment of the microservices to the infrastructure for the most part. *Cloud Foundry* is discussed as an example for a PaaS.

11 Recipe: Messaging and Kafka

This chapter shows the integration of microservices using a message-oriented middleware (MOM). A MOM sends messages and ensures that they reach the recipient. MOMs are asynchronous. This means that they do not implement request/reply as is done with synchronous communication protocols, but only send messages.

MOMs have different characteristics such as high reliability, low latency, or high throughput. MOMs also have a long history; they form the basis of numerous business-critical systems.

This chapter covers the following points:

- First, it gives an overview of the various MOMs and their differences. This allows readers to form an opinion on which MOM is most suitable for supporting their application.
- The introduction into Kafka shows why Kafka is especially well suited for a microservices system and how event sourcing (see section 10.2) can be implemented with Kafka.
- Finally, the example in this chapter illustrates at the code level how an event sourcing system with Kafka can be built in practice.

11.1 Message-oriented Middleware (MOM)

Microservices are decoupled by a MOM. A microservice sends a message to or receives it from the MOM. This means that sender and recipient do not know each other, but only the communication channel. Service discovery is therefore not necessary. Sender and recipient find each other via the topic or queue through which they exchange messages. Load balancing is also easy. If several recipients have registered for the same communication channel, a message can be processed by one of the recipients and the load can be distributed, thereby eliminating need for a specific infrastructure for load balancing.

However, a MOM is a complex software that handles all communication. Therefore, the MOM must be highly available and has to offer a high throughput. MOMs are generally very mature products, but ensuring adequate performance under all conditions requires a lot of know-how, for example, concerning the configuration.

Variants of MOMs

In the area of MOMs, the following products are popular:

- JMS[6] (Java Messaging Service) is a standardized API for the programming language Java and part of the Java EE standard. Well known implementations are Apache ActiveMQ[7] or IBM MQ[8], which was previously known as IBM MQSeries. However, many more JMS products[9] are available. JMS implementations might also provide APIs for other programming languages than Java. Those would not be covered by the JMS specification as it is specific for Java. Java application servers that support the entire Java EE profile – not just the web profile – have to contain a JMS implementation, so that JMS is often anyway available.
- AMQP[10] (Advanced Message Queuing Protocol) does not standardize an API, but a network protocol at the level of TCP/IP. This allows for a simpler exchange of the implementation. RabbitMQ[11], Apache ActiveMQ[12], and Apache Qpid[13] are the best known implementations of the AMQP standard. There are also a lot more implementations[14].

In addition, there is ZeroMQ[15], which does not comply with any of the standards. ZeroMQ is a library i.e. there is no need to install a message broker to use it. And MQTT[16] is a messaging protocol that plays a prominent role for the Internet of Things (IoT).

All of these MOM technologies can be used to build a microservices system. If a certain technology is already in use and thus knowledge about its use it readily available, a decision for an already known technology can make a lot of sense. It takes a lot of effort to run a microservices system. The use of a well-known technology reduces risk and effort. The requirements for availability and scalability of MOMs are high. A well-known MOM can help to meet these requirements in a simple way.

11.2 The Architecture of Kafka

In the area of microservices, Kafka[17] is an interesting option. In addition to typical features such as high throughput and low latency, Kafka can compensate for the failure of individual servers via replication and can scale with a larger number of servers.

Kafka Stores the Message History

Above all, Kafka is able to store an extensive message history. Usually, MOMs aim only to deliver messages to recipients. The MOM then deletes the message because it has left the MOM's area of responsibility. This saves resources. However, it also means that approaches such as event sourcing

[6]https://jcp.org/aboutJava/communityprocess/final/jsr914/index.html
[7]http://activemq.apache.org/
[8]http://www-03.ibm.com/software/products/en/ibm-mq
[9]https://en.wikipedia.org/wiki/Java_Message_Service#Provider_implementations
[10]https://www.amqp.org/
[11]https://www.rabbitmq.com/
[12]http://activemq.apache.org/
[13]https://qpid.apache.org/
[14]https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol#Implementations
[15]http://zeromq.org/
[16]http://mqtt.org/
[17]https://kafka.apache.org/

(see section 10.2) are possible only if every microservice stores the event history itself. Kafka, on the other hand, can save records permanently. Kafka can also handle large amounts of data and can be distributed across multiple servers.

Kafka also has stream-processing capabilities. For this, applications receive the data records from Kafka, transform them, and send them back to Kafka.

Kafka: Licence and Committers

Kafka is licensed under Apache 2.0. This license grants users extensive freedom. The project is run by the Apache Software Foundation, which manages several open source projects. Many committers work for the company Confluent, which also offers commercial support, a Kafka Enterprise solution, and a solution in the cloud.

APIs

Kafka offers a separate API for each of the three different tasks of a MOM:

- The *producer API* serves to send data.
- The *consumer API* to receive data.
- Finally, the *streams API* to transform the data.

Kafka is written in Java. The APIs can be used with a language-neutral protocol. Clients[18] for many programming languages are available.

Records

Kafka organizes data in *records*. This is what other MOMs call "messages". Records contain the transported data as a *value*. Kafka treats the value as a black box and does not interpret the data. In addition, records have a *key* and a *timestamp*.

A record could contain information about a new order or an update to an order. The key can then be composed of the identity of the record and information about whether the record is an update or a new order – for example "update42" or "neworder42."

Topics

A *topic* is a set of records. Consumers send records to a topic and consumers receive them from a topic. Topics have names.

If microservices in an e-commerce system are interested in new orders or want to inform other microservices about new orders, they could use a topic called "order." New customers would be another topic; that topic could be called "customer."

[18]https://cwiki.apache.org/confluence/display/KAFKA/Clients

Partitions

Topics are divided into *partitions*. Partitions allow strong guarantees concerning the order of records, but also parallel processing.

When a producer creates a new record, it is appended to a partition. Therefore, each record is stored in only one single partition. Records are usually assigned to partitions by calculating the hash of the key of the record. However, a producer can also implement its own algorithm to assign records to a partition.

For each partition, the order of the records is preserved. That means the order in which the records are written to the partition is also the order in which consumers read the records. There is no guarantee of order across partitions. Therefore, partitions are also a concept for parallel processing: Reading in a partition is linear. A consumer has to process each record in order. Across partitions, processing can be parallel.

More partitions have different effects[19]. They allow more parallelism, but at a cost of higher overhead and resource consumption. So it makes sense to have a considerable number of partitions, but not too many. Hundreds of partitions is typical.

Basically, a partition is just a file to which records are appended. Appending data is one of the most efficient operations on a mass storage device. Moreover, such operations are very reliable and easy to implement. This makes the implementation of Kafka not too complex.

To continue the example with the "order" topic: There might be a record with the key "neworder42" that contains an event about the order 42 that was just created and "updated42" which contains an update to the order 42. With the default key algorithm, the keys would be hashed. The two records might therefore end up in different partitions and no order might be preserved. This is not ideal because the two events obviously need to be processed in the correct order. It makes no sense to process "updated42" before "neworder42." However, it is perfectly fine to process "updated42" and "updated21" because the orders probably do not depend on each other. So the producer would need to implement an algorithm that sends the records with the keys "updated42" and "neworder42" to the same partition.

Commit

For each consumer, Kafka stores the offset for each partition. This offset indicates which record in the partition the consumer read and processed last. It helps Kafka to ensure that each record is eventually handled.

When consumers have processed a record, they commit a new offset. In this way, Kafka knows at all times which records have been processed by which consumer and which records still have to be processed. Of course, the consumers can commit records before they are actually processed. As a result, records never getting processed is a possibility.

[19]http://www.confluent.io/blog/how-to-choose-the-number-of-topicspartitions-in-a-kafka-cluster/

The commit is on an offset – for example, "all records up to record 10 in this partition have been processed." So a consumer can commit a batch of records, which results in better performance because fewer commits are required. But then duplicates can occur. This happens when the consumer fails after processing a part of a batch but has not yet committed the batch. At restart, the application would read the complete batch again, because Kafka restarts at the last committed record and thus at the beginning of the batch.

Kafka also supports *exactly once semantics*[20] – that is, a guaranteed one-time delivery.

Polling

The consumers poll the data. In other words, the consumers fetch new data and process it. With the push model, on the other hand, the producer would send the data to the consumer. Polling doesn't seem to be very elegant. However, in absence of a push, the consumers are protected from too much load when a large number of records are being sent and have to be processed. Consumers can decide for themselves when they process the records. Libraries like Spring Kafka for Java, which is used in the example, poll new records in the background. The developer implement methods to handle new records. Spring Kafka then calls them The polling is hidden from the developer.

Records, Topics, Partitions, and Commits

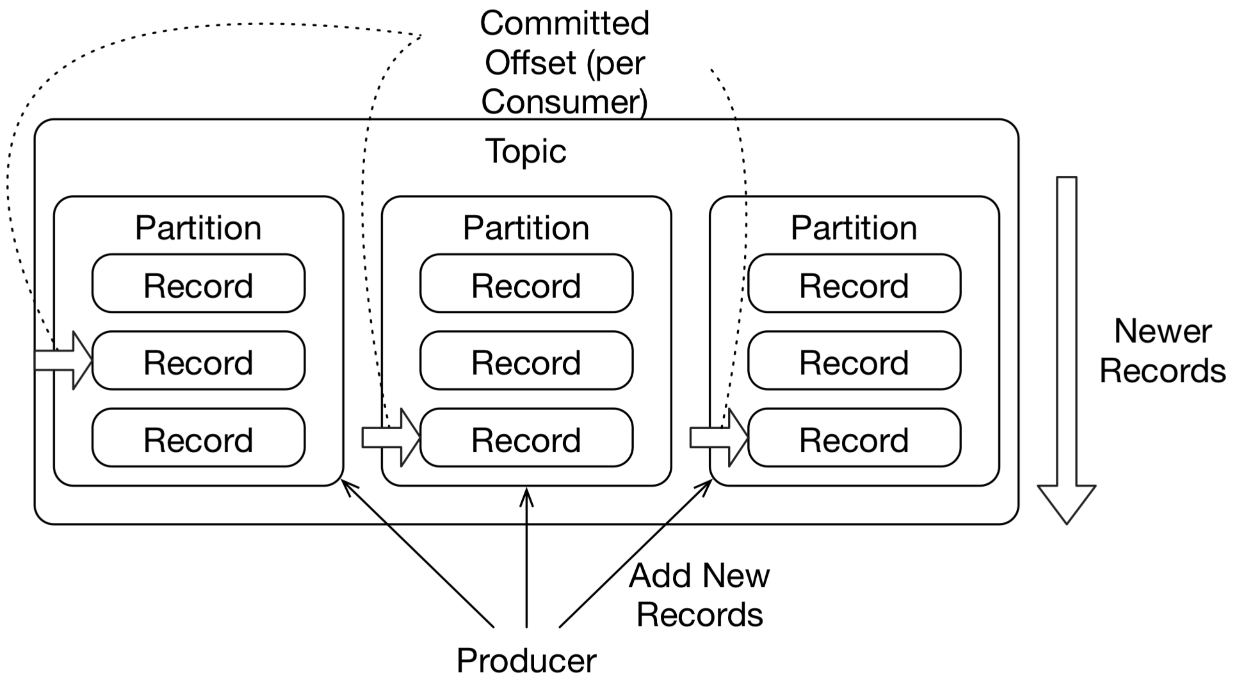


Fig. 11-1: Partitions and Topics in Kafka

[20] https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/

Figure 11-1 shows an example. The topic is divided into three partitions, each containing three records. In the lower part of the figure are the newer records. The producer creates new records at the bottom. The consumer has not yet committed the latest record for the first partition, but has for all other partitions.

Replication

Partitions store the data. Because data in one partition is independent from data in the other partitions, partitions can be distributed over servers. Each server then processes some partitions. This allows load balancing. To handle a larger load, new servers just need to be added and some partitions must be moved to the new server.

The partitions can also be replicated, so that the data is stored on several servers. This way Kafka can be made fail-safe. If one server crashes or loses its data, other replicas still exist.

The number “N” of replicas can be configured. When writing, you can determine how many in-sync replicas must commit changes. With N=3 replicas and two in-sync replicas, the cluster remains available even if one of the three replicas fails. Even if one server fails, new records can still be written to two replicas. If a replica fails, no data is lost because every write operation must have been successful on at least two replicas. Even if a replica is lost, the data must still be stored on at least one additional replica. Kafka thus supports some fine tuning regarding the CAP theorem (see section 10.2) by changing the number of replicas and in-sync replicas.

Leader and Follower

The replication is implemented in such a way that one leader writes and the remaining replicas write as followers. The producer writes directly to the leader. Several write operations can be combined in a batch. On the one hand, it then takes longer before a batch is complete and for the changes to be actually saved. On the other hand, throughput increases because it is more efficient to store multiple records at once. The overhead of coordinating the writes just happens once for the full batch and not for each record.

Retry Sending

If the transfer to the consumer was not successful, the producer can use the API to specify that the transfer is attempted again. The default setting is that sending a record is not repeated. This can cause records to be lost. If the transfer is configured to occur more than once, the record may already have been successfully transferred despite the error. In this case, there would be a duplicate, which the consumer would have to be able to deal with. One possibility is to develop the consumer in such a way that it offers idempotent processing. This means that the consumer is in the same state, no matter how often the consumer processes a record (see section 10.3). For example, if a duplicate is received, the consumer can determine that it has already modified the record accordingly and ignore it.

Consumer Groups

An event like "neworder42" should probably be processed only once by one instance of the invoicing microservice. Therefore, just one instance of a microservice should receive it. That ensures that only one invoice is written for this order. However, another instance of a microservice might work on "neworder21" in parallel.

Consumers are organized in consumer groups. Each partition sends records to exactly one consumer in the consumer group. One consumer can be responsible for several partitions.

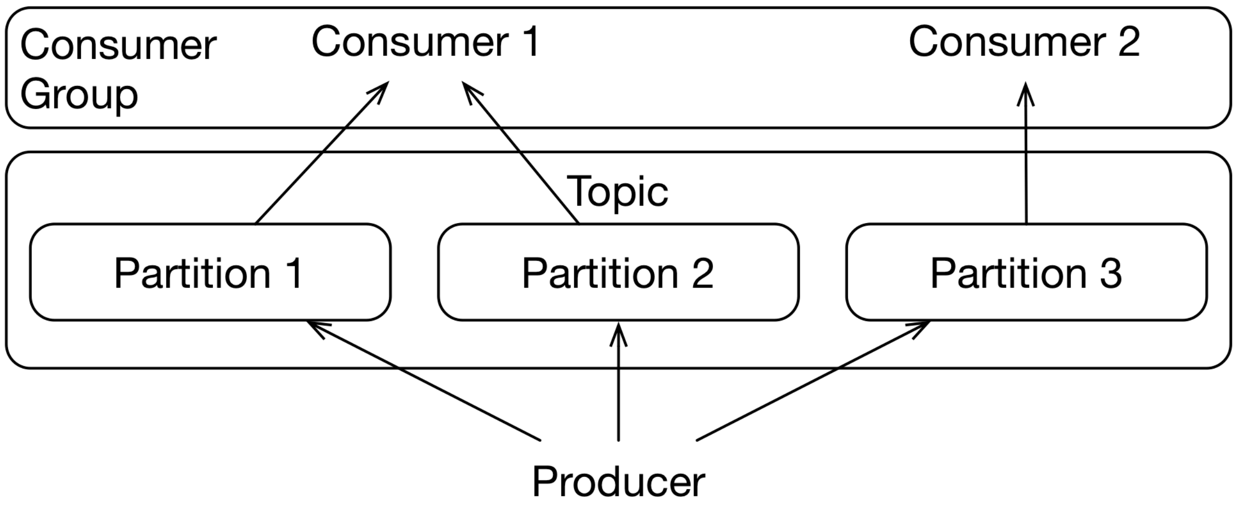


Fig. 11-2: Consumer Groups and Partitions

Thus, a consumer receives the messages of one or multiple partitions. Figure 11-2 shows an example. Consumer 1 receives the messages of partitions 1 and 2. Consumer 2 receives the messages of partition 3.

So the invoicing microservice instances could be organized in a consumer group, ensuring that only one instance of the invoicing microservice processes each record.

When a consumer receives a message from a partition, it will also later receive all messages from the same partition. The order of messages per partition is also preserved. This means that records in different partitions can be handled in parallel, and at the same time the sequence of records in a single partition is guaranteed. Therefore, the instance of the invoicing microservice that receives "neworder42" would later on also receive "updated42" if those records are sent to the same partition. So the instance would be responsible for all events about the order 42.

Of course, this applies only if the mapping of consumers to partitions remains stable. For example, if new consumers are added to the consumer group for scaling, the mapping can change. The new consumer would need to handle at least one partition. That partition was previously handled by a different consumer.

The maximum number of consumers in a consumer group is equal to the number of partitions, because each consumer must be responsible for at least one partition. Ideally, there are more partitions than consumers to be able to add more consumers when scaling.

Consumers are always members of a consumer group. Therefore, they receive only records sent to their partitions. If each consumer is to receive all records from all partitions, then there must be a separate consumer group for each consumer with only one member.

Persistence

Kafka is a mixture of a messaging system and data storage solution. The records in the partitions can be read by consumers and written by producers. The default retention for records is seven days, but it can be changed. The records can also be saved permanently. The consumers merely store their offset.

A new consumer can therefore process all records that have ever been written by a producer in order to update its own state.

If a consumer is much too slow to handle all records in a timely manner, Kafka stores them for quite a long time, thereby allowing the consumer to process the records later to keep up.

Log Compaction

However, this means that Kafka has to store more and more data over time. Some records, however, eventually become irrelevant. If a customer has moved several times, you may only want to keep the last information about the last move as a record in Kafka. Log compaction is used for this purpose. All records with the same key are removed, except for the last one. Therefore, the choice of the key is very important and must be considered from a domain logic point of view in order to have all the relevant records still available after log compaction.

So a log compaction for the order topic would remove all events with the key "updated42" but the very last one. As a result, only the very last update to the order remains available in Kafka.

11.3 Events with Kafka

Systems which communicate via Kafka can quite easily exchange events (see also section 10.2).

- Records can be saved permanently, and so a consumer can read out the history and rebuild its state. The consumer does not have to store the data locally, but can rely on Kafka. However, this means that all relevant information must be stored in the record. Section 10.2 discussed the benefits and disadvantages of this approach.
- If an event becomes irrelevant due to a new event, the data can be deleted by Kafka's log compaction.
- Via consumer groups, Kafka can ensure that a single consumer handles each record. This simplifies matters, for example, when an invoice is to be written. In this case, only one consumer should write an invoice. It would be an error if several consumers were to create several invoices at the same time.

Sending Events

The producer can send the events at different times. The simplest option is to send the event after the actual action has taken place. So the producer first processes an order before informing the other microservices about the order with an event. In this case, though, the producer could possibly change the data in the database and not send the event because, for example, it fails prior to sending the event.

However, the producer can also send the events before the data is actually changed. So when a new order arrives, the producer sends the event before modifying the data in the local database. This doesn't make much sense, because events are actually information about an event that has already happened. Finally, an error may occur during the action. If this happens, the event has already been sent, although the action never took place.

Sending events before the actual action also has the disadvantage that the actual action is delayed. First an event is sent, which takes some time, and only after the event has been sent can the action be performed. So the action is delayed by the time it takes to send the event. This can lead to a performance problem.

It is also possible to collect the events in a local database and to send them in a batch. In this case, writing the changed data and generating the data for the event in the database can take place in one transaction. The transaction can ensure that either the data is changed and an event is created in the database, or neither takes place. This solution also achieves higher throughput because batches can be used in Kafka to send several events in the database table at once. However, the latency is higher: A change can be found in Kafka only after the next batch has been written.

11.4 Example

The example in this section is based on the example for events from section 10.2 (see figure 11-3). The microservice *microservice-kafka-order* is responsible for creating the order. It sends the orders to a Kafka topic. Therefore *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. So each record is just consumed and processed by one instance of the microservice *microservice-kafka-invoicing* and one instance of *microservice-kafka-shipping*.

Data Model for the Communication

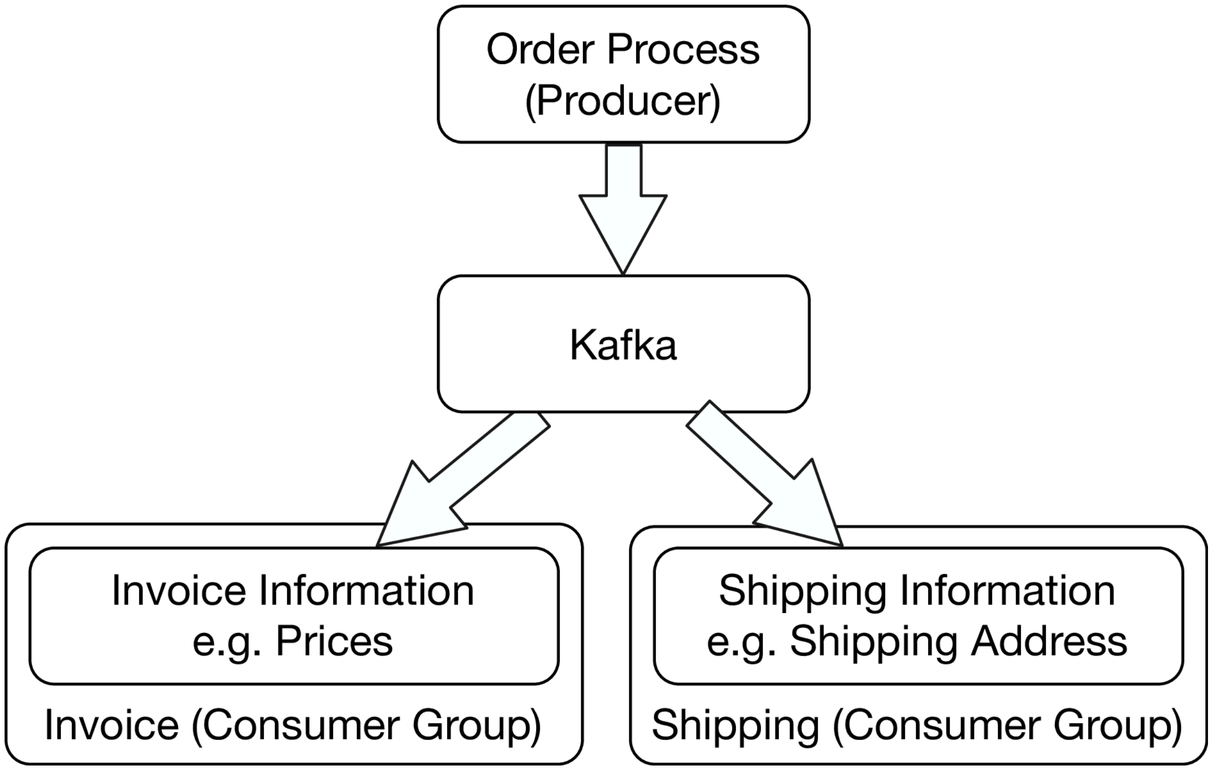

Order Process (Producer)
Kafka
Invoice Information e.g. Prices
Invoice (Consumer Group)
Shipping Information e.g. Shipping Address
Shipping (Consumer Group)

Fig. 11-3: 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 demo, 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 also 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. 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 is needed by shipping and invoicing on the one hand, and order on the other. 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) (see section 2.1). Section 10.2 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.

Data Model for the Database

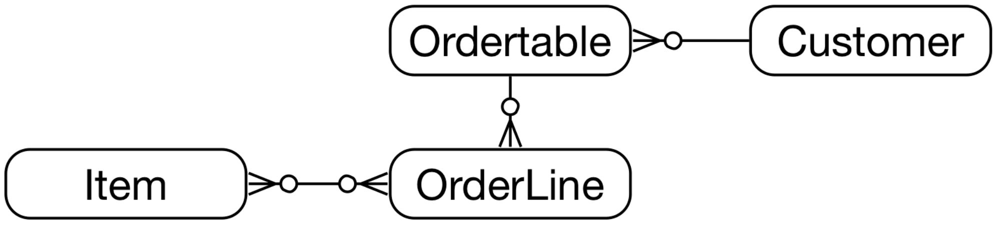


Fig. 11-4: Data Model in the System microservice-kafka-order

The database of the order microservice (see figure 11-4) contains a table for the orders (`Ordertable`) and the individual items in the orders (`OrderLine`). Goods (`Item`) and customers (`Customer`) also have their own tables.

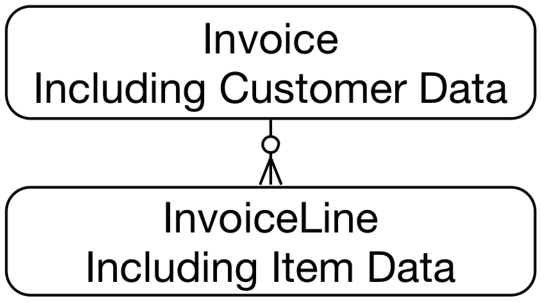


Fig. 11-5: Data Model in the System microservice-kafka-order

In the microservice *microservice-kafka-invoice*, the tables for customers and items are missing. The customer data is stored only as part of `invoice`, and the item data as part of `invoiceLine` (see figure 11-5). The data in the tables are copies of the customers' and items' data at the time when the order was transferred to the system. This means that if a customer changes his or her data or a product changes its price, this does not affect the previous invoices. That is correct from a domain logic perspective. After all, a price change should not affect invoices that have already been written. Otherwise, getting the correct price and customer information at the time of the invoice can be implemented only with a complete history of the data, which is quite complex. With the model used here, it is also very easy to transfer discounts or special offers to the invoice. It is necessary to send a different price for a product.

For the same reason, the microservice *microservice kafka-shipping* has only the database tables `Shipping` and `ShippingLine`. Data for customers and items is copied to these tables so that the data is stored there as it was when the delivery was triggered.

This example illustrates how *bounded context* simplifies the domain models.

Inconsistencies

The example also shows another effect: The information in the system can be inconsistent. Orders without invoices or orders without deliveries can occur, but such conditions are not permanent. At some point the Kafka topic will be read out with the new orders, and the new orders will then generate an invoice and a delivery.

Technical Structure

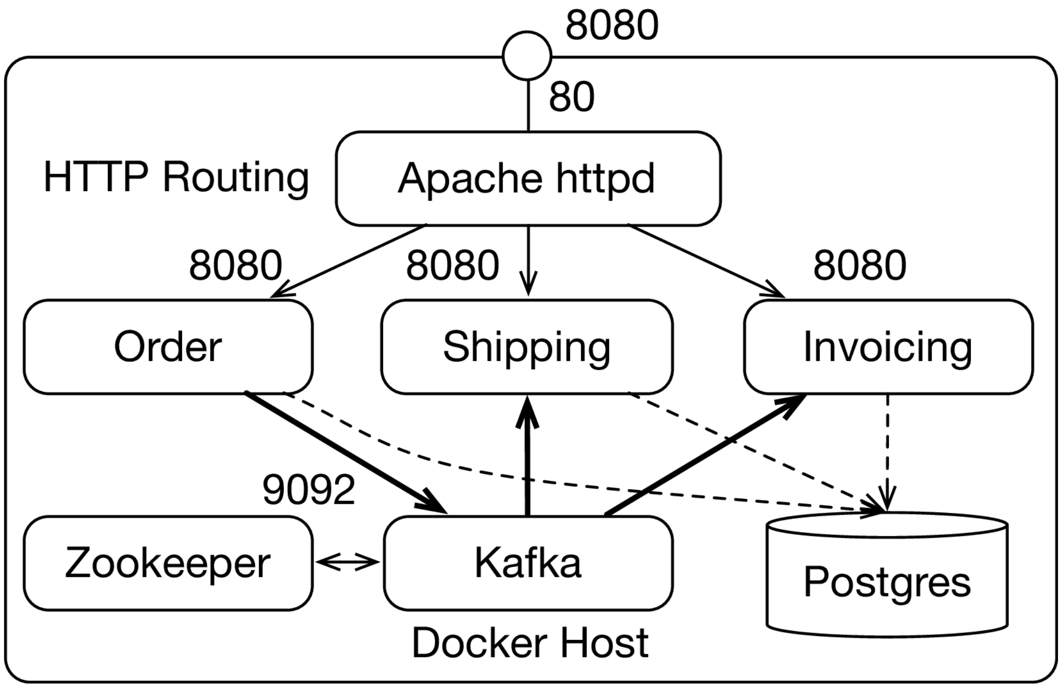


Fig. 11-6: Overview of the Kafka System

Figure 11-6 shows how the example is structured technically.

- 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* serves to coordinate the Kafka instances and stores, among others, information about the distribution of topics and partitions. The example uses the image at https://hub.docker.com/r/wurstmeister/zookeeper/.
- 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/.
- 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 regards 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. Section 0.4 describes what software has to be installed to start the example.

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 appendix B 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 appendix C 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. 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 result from 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 just 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 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.

Such a function is called a partitioner[21]. Therefore, it is possible to write custom code for the distribution of records onto the partitions. This allows a producer to write all records which 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, after all, to use only the order ID as key. To avoid the problem with log compaction, it is possible to send the complete state of the order along with each record

[21] https://kafka.apache.org/0110/javadoc/org/apache/kafka/clients/producer/Partitioner.html

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 can take quite some 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.

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 in five partitions. 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 rather 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 for this 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.

```java
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, for example, 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 first of all 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`.

```java
@KafkaListener(topics = "order")
public void order(Invoice invoice, Acknowledgment acknowledgment) {
  log.info("Received invoice " + invoice.getId());
  invoiceService.generateInvoice(invoice);
  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 also possible to implement more complex logic 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.

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 therefore 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 first 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.

Tests with Embedded Kafka

In a JUnit test, an *embedded Kafka server* can be used to analyze the functionality of the microservices. In this case, a Kafka server runs in the same Java Virtual Machine (JVM) as the test. Thus, it is not necessary to build up an infrastructure for the test, and consequently there is no need to tear down the infrastructure again after the test.

This requires two things essentially:

- An embedded Kafka server has to be started. With a class rule, JUnit can be triggered to start a Kafka server prior to the tests and to shut it down again after the tests. Therefore, a variable with `@ClassRule` must be added to the code.

```java
@ClassRule
public static KafkaEmbedded embeddedKafka =
  new KafkaEmbedded(1, true, "order");
```

- The Spring Boot configuration must be adapted in such a manner that Spring Boot uses the Kafka server. This code is found in a method annotated with `@BeforeClass`, so that it executes before the tests.

```
@BeforeClass
public static void setUpBeforeClass() {
  System.setProperty("spring.kafka.bootstrap-servers",
   embeddedKafka.getBrokersAsString());
}
```

Avro as Data Format

Avro[22] is a data format quite frequently used together with Kafka[23] and Big Data solutions from the Hadoop area. Avro is a binary protocol, but also offers a JSON-based representation. There are, for example, Avro libraries for Python, Java, C#, C++, and C.

Avro supports schemas. Each dataset is saved or sent together with its schema. For optimization, a reference to a schema from the schema repository can be used rather than a copy of the complete schema. Thereby, it is clear which format the data has. The schema contains a documentation of the fields. This ensures the long-term interpretation of the data, and that the semantics of the data are clear. In addition, the data can be converted to another format upon reading. This facilitates the schema evolution[24]. New fields can be added when default values are defined, so that the old data can be converted into the new schema by using the default value. When fields are deleted, again a default value can be given so that new data can be converted into the old schema. In addition, the order of the fields can be changed because the field names are stored.

An advantage of the flexibility associated with schema migration is that very old records can be processed with current software and the current schema. Also, software based on an old schema can process new data. Message-oriented middleware (MOM) typically does not have such requirements because messages are not stored for very long. Only upon long-term record storage does schema evolution turn into a challenge.

11.5 Recipe Variations

The example sends the data for the event along in the records. There are alternatives to this (see section 11.4):

- The entire dataset is always sent along – that is, the complete order.
- The records could contain only an ID of the dataset for the order. As a result, the recipient can retrieve just the information about the dataset it really needs.
- An individual topic exists for each client. All the records have their own data structure adapted to the client.

[22]http://avro.apache.org/
[23]https://www.confluent.io/blog/avro-kafka-data/
[24]https://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html

Other MOMs

An alternative to Kafka would be another MOM. This might be a good idea if the team has already plenty of experience with a different MOM. Kafka differs from other MOMs in the long-term storing of records. However, this is relevant only for event sourcing. And even then, every microservice can save the events itself. Therefore, storage in the MOM is not absolutely necessary. It can even be difficult because the question of the data model arises.

Likewise, asynchronous communication with Atom (see chapter 12) can be implemented. In a microservices system, however, there should only be one solution for asynchronous communication so that the effort for building and maintaining the system does not become too great. Therefore, using Atom and Kafka or any other MOM at the same time should be avoided.

Kafka can be combined with frontend integration (see chapter 7). These approaches act at different levels so that a combined use does not represent a problem. A combination with synchronous mechanisms (see chapter 13) makes less sense because the microservices should communicate either synchronously or asynchronously. Still, such a combination might be sensible in situations where synchronous communication is necessary.

11.6 Experiments

- Supplement the system with an additional microservice.
  - As an example, a microservice can be used which credits the customer with a bonus depending on the value of the order or counts the orders.
  - Of course, you can copy and modify one of the existing microservices.
  - Implement a Kafka consumer for the topic `order` of the Kafka server `kafka`. The consumer should credit the customer with a bonus when ordering or count the messages.
  - In addition, the microservice should also present an HTML page with the information (customer bonuses or number of messages).
  - Place the microservice into a Docker image and reference it in the `docker-compose.yml`. There you can also specify the name of the Docker container.
  - Create in `docker-compose.yml` a link from the container `apache` to the container with the new service, and from the container with the new service to the container `kafka`.
  - The microservice must be accessible from the home page. To do this, you have to create a load balancer for the new Docker container in the file `000-default.conf` in the Docker container `apache`. Use the name of the Docker container, and then add a link to the new load balancer to the file `index.html`.
- It is possible to start additional instances of the shipping or invoicing microservice. This can be done with `docker-compose up -d --scale shipping=2` or `docker-compose up -d --scale invoicing=2`. `docker logs mskafka_invoicing_2` can be used to look at the logs. In the logs the microservice also indicates which Kafka partitions it processes.
- Kafka can also transform data with Kafka streams. Explore this technology by searching for information about it on the web!

- Currently, the example application uses JSON. Implement a serialization with Avro[25]. A possible starting point for this can be https://www.codenotfound.com/2017/03/spring-kafka-apache-avro-example.html.
- Log compaction is a possibility to delete superfluous records from a topic. The Kafka documentation[26] explains this feature. To activate log compaction, it has to be switched on when the topic is generated. See also https://hub.docker.com/r/wurstmeister/kafka/. Change the example in such a way that log compaction is activated.

11.7 Conclusion

Kafka offers an interesting approach for the asynchronous communication between microservices.

Benefits

- Kafka can store records permanently, which facilitates the use of event sourcing in some scenarios. In addition, approaches like Avro are available for solving problems like schema evolution.
- The overhead for the consumers is low. They have to remember only the position in each partition.
- With partitions, Kafka has a concept for parallel processing and, with consumer groups, it has a concept to guarantee the order of records for consumers. In this way, Kafka can guarantee delivery to a consumer and at the same time distribute the work among several consumers.
- Kafka can guarantee the delivery of messages. The consumer commits the records it has successfully processed. If an error occurs, it does not commit the record and tries to process it again.

Therefore, it is worth taking a look at this technology, especially for microservices, even if other asynchronous communication mechanisms are also useful.

Challenges

However, Kafka also provides some challenges.

- Kafka is an additional infrastructure component. It must be operated. Especially with messaging solutions, configuration is often not easy.
- If Kafka is used as event storage, the records must contain all the data that all clients need. This is often not easy to implement because of bounded context. (see section 2.1).

[25]http://avro.apache.org/
[26]https://kafka.apache.org/documentation/#compaction

Part III: Operation

The third part of this book discusses the operation of microservices. In a microservices environment, a system consists of many microservices. These need to be operated. In the case of a deployment monolith, just a single systems must be operated. Therefore, operation is a very important topic for a microservices environment.

Operation: Basics

First, chapter 19 introduces the basics of operating microservices.

Monitoring with Prometheus

Chapter 20 deals with the monitoring of microservices and describes Prometheus as concrete tool.

Log Data Analysis with the Elastic Stack

The topic of chapter 21 is the analysis of log data. The Elastic Stack is introduced as concrete technical approach for log data analysis.

Tracing with Zipkin

Chapter 22 explains how to use Zipkin to trace requests across multiple microservices.

Service Meshes with Istio

Service meshes like Istio add proxies to the network traffic in a microservices system. This allows support for monitoring, tracing and resilience without any impact on the code. Chapter 23 discusses Istio as an example of a service mesh.

Conclusion of the Book

The book ends with an outlook in chapter 24.

Table of Contents of the Complete Book

Part II: Technology Stacks

Part III: Operation

Part I: Principles of Microservices Architecture