

# MICROSERVICES



A Practical Guide  
**Second Edition**  
Principles, Concepts, and Recipes

Eberhard Wolff

Distributed by Manning Publications

# **Microservices - A Practical Guide**

Principles, Concepts, and Recipes

Eberhard Wolff

This version was published on 2019-06-22

ISBN 9781617296505

DISTRIBUTED BY MANNING

This book was created solely by its Author and is being distributed by Manning Publications “as is,” with no guarantee of completeness, accuracy, timeliness, or other fitness for use. Neither Manning nor the Author make any warranty regarding the results obtained from the use of the contents herein and accept no liability for any decision or action taken in reliance on the information in this book nor for any damages resulting from this work or its application.

© 2018 - 2019 Eberhard Wolff

# Contents

<b>0 Introduction</b> . . . . .	i
0.1 Structure of the Book . . . . .	ii
0.2 Target Group . . . . .	vi
0.3 Prior Knowledge . . . . .	vii
0.4 Quick Start . . . . .	vii
0.5 Acknowledgements . . . . .	viii
0.6 Website . . . . .	viii
<b>Part I: Principles of Microservices</b> . . . . .	1
<b>1 Microservices</b> . . . . .	2
1.1 Microservices: Definition . . . . .	2
1.2 Reasons for Microservices . . . . .	3
1.3 Challenges . . . . .	10
1.4 Variations . . . . .	11
1.5 Conclusion . . . . .	12
<b>2 Micro and Macro Architecture</b> . . . . .	13
2.1 Bounded Context and Strategic Design . . . . .	13
2.2 Technical Micro and Macro Architecture . . . . .	20
2.3 Operation: Micro or Macro Architecture? . . . . .	23
2.4 Give a Preference to Micro Architecture! . . . . .	25
2.5 Organizational Aspects . . . . .	26
2.6 Independent Systems Architecture Principles (ISA) . . . . .	28
2.7 Variations . . . . .	31
2.8 Conclusion . . . . .	32
<b>3 Migration</b> . . . . .	33
3.1 Reasons for Migrating . . . . .	33
3.2 A Typical Migration Strategy . . . . .	34
3.3 Alternative Strategies . . . . .	39
3.4 Build, Operation, and Organization . . . . .	40
3.5 Variations . . . . .	42
3.6 Conclusion . . . . .	43

## CONTENTS

<b>Part II: Technology Stacks . . . . .</b>	<b>44</b>
<b>4 Docker . . . . .</b>	<b>46</b>
4.1 Docker for Microservices: Reasons . . . . .	46
4.2 Docker Basics . . . . .	47
4.3 Docker Installation and Docker Commands . . . . .	51
4.4 Installing Docker Hosts with Docker Machine . . . . .	52
4.5 Dockerfiles . . . . .	53
4.6 Docker Compose . . . . .	57
4.7 Variations . . . . .	59
4.8 Conclusion . . . . .	61
<b>5 Technical Micro Architecture . . . . .</b>	<b>62</b>
5.1 Requirements . . . . .	62
5.2 Reactive . . . . .	64
5.3 Spring Boot . . . . .	66
5.4 Go . . . . .	71
5.5 Variations . . . . .	74
5.6 Conclusion . . . . .	75
<b>6 Self-contained Systems . . . . .</b>	<b>77</b>
6.1 Reasons for the Term Self-contained Systems . . . . .	77
6.2 Definition . . . . .	78
6.3 An Example . . . . .	81
6.4 SCSs and Microservices . . . . .	82
6.5 Challenges . . . . .	83
6.6 Benefits . . . . .	85
6.7 Variations . . . . .	85
6.8 Conclusion . . . . .	86
<b>7 Concept: Frontend Integration . . . . .</b>	<b>87</b>
7.1 Frontend: Monolith or Modular? . . . . .	87
7.2 Options . . . . .	90
7.3 Resource-oriented Client Architecture (ROCA) . . . . .	90
7.4 Challenges . . . . .	93
7.5 Benefits . . . . .	94
7.6 Variations . . . . .	95
7.7 Conclusion . . . . .	95
<b>8 Recipe: Links and Client-side Integration . . . . .</b>	<b>97</b>
8.1 Overview . . . . .	97
8.2 Example . . . . .	104
8.3 Variations . . . . .	105
8.4 Experiments . . . . .	107

## CONTENTS

8.5 Conclusion . . . . .	107
<b>9 Recipe: Server-side Integration using Edge Side Includes (ESI) . . . . .</b>	<b>110</b>
9.1 ESI: Concepts . . . . .	110
9.2 Example . . . . .	111
9.3 Varnish . . . . .	113
9.4 Recipe Variations . . . . .	118
9.5 Experiments . . . . .	120
9.6 Conclusion . . . . .	120
<b>10 Concept: Asynchronous Microservices . . . . .</b>	<b>122</b>
10.1 Definition . . . . .	122
10.2 Events . . . . .	125
10.3 Challenges . . . . .	129
10.4 Advantages . . . . .	133
10.5 Variations . . . . .	134
10.6 Conclusions . . . . .	134
<b>11 Recipe: Messaging and Kafka . . . . .</b>	<b>135</b>
11.1 Message-oriented Middleware (MOM) . . . . .	135
11.2 The Architecture of Kafka . . . . .	136
11.3 Events with Kafka . . . . .	142
11.4 Example . . . . .	143
11.5 Recipe Variations . . . . .	152
11.6 Experiments . . . . .	153
11.7 Conclusion . . . . .	154
<b>12 Recipe: Asynchronous Communication with Atom and REST . . . . .</b>	<b>155</b>
12.1 The Atom Format . . . . .	155
12.2 Example . . . . .	161
12.3 Recipe Variations . . . . .	165
12.4 Experiments . . . . .	166
12.5 Conclusion . . . . .	167
<b>13 Concept: Synchronous Microservices . . . . .</b>	<b>169</b>
13.1 Definition . . . . .	169
13.2 Benefits . . . . .	172
13.3 Challenges . . . . .	172
13.4 Variations . . . . .	174
13.5 Conclusion . . . . .	175
<b>14 Recipe: REST with the Netflix Stack . . . . .</b>	<b>176</b>
14.1 Example . . . . .	176
14.2 Eureka: Service Discovery . . . . .	178

## CONTENTS

14.3 Router: Zuul . . . . .	182
14.4 Load Balancing: Ribbon . . . . .	183
14.5 Resilience: Hystrix . . . . .	185
14.6 Recipe Variations . . . . .	189
14.7 Experiments . . . . .	190
14.8 Conclusion . . . . .	192
<b>15 Recipe: REST with Consul and Apache httpd . . . . .</b>	<b>194</b>
15.1 Example . . . . .	194
15.2 Service Discovery: Consul . . . . .	197
15.3 Routing: Apache httpd . . . . .	199
15.4 Consul Template . . . . .	199
15.5 Consul and Spring Boot . . . . .	201
15.6 DNS and Registrar . . . . .	202
15.7 Recipe Variations . . . . .	203
15.8 Experiments . . . . .	204
15.9 Conclusion . . . . .	206
<b>16 Concept: Microservices Platforms . . . . .</b>	<b>208</b>
16.1 Definition . . . . .	208
16.2 Variations . . . . .	210
16.3 Conclusion . . . . .	211
<b>17 Recipe: Docker Containers with Kubernetes . . . . .</b>	<b>212</b>
17.1 Kubernetes . . . . .	212
17.2 The Example with Kubernetes . . . . .	214
17.3 The Example in Detail . . . . .	217
17.4 Additional Kubernetes Features . . . . .	221
17.5 Recipe Variations . . . . .	223
17.6 Experiments . . . . .	224
17.7 Conclusion . . . . .	225
<b>18 Recipe: PaaS with Cloud Foundry . . . . .</b>	<b>227</b>
18.1 PaaS: Definition . . . . .	227
18.2 Cloud Foundry . . . . .	229
18.3 The Example with Cloud Foundry . . . . .	230
18.4 Recipe Variations . . . . .	234
18.5 Experiments . . . . .	235
18.6 Serverless . . . . .	236
18.7 Conclusion . . . . .	236
<b>Part III: Operation . . . . .</b>	<b>238</b>
<b>19 Concept: Operation . . . . .</b>	<b>239</b>

## CONTENTS

19.1 Why Operation Is Important . . . . .	239
19.2 Approaches for the Operation of Microservices . . . . .	242
19.3 Effects of the Discussed Technologies . . . . .	243
19.4 Conclusion . . . . .	244
<b>20 Recipe: Monitoring with Prometheus . . . . .</b>	<b>245</b>
20.1 Basics . . . . .	245
20.2 Metrics for Microservices . . . . .	247
20.3 Metrics with Prometheus . . . . .	248
20.4 Example with Prometheus . . . . .	252
20.5 Recipe Variations . . . . .	254
20.6 Experiments . . . . .	255
20.7 Conclusion . . . . .	257
<b>21 Recipe: Log Analysis with the Elastic Stack . . . . .</b>	<b>258</b>
21.1 Basics . . . . .	258
21.2 Logging with the Elastic Stack . . . . .	261
21.3 Example . . . . .	262
21.4 Recipe Variations . . . . .	264
21.5 Experiments . . . . .	264
21.6 Conclusion . . . . .	265
<b>22 Recipe: Tracing with Zipkin . . . . .</b>	<b>266</b>
22.1 Basics . . . . .	266
22.2 Tracing with Zipkin . . . . .	267
22.3 Example . . . . .	269
22.4 Recipe Variations . . . . .	270
22.5 Conclusion . . . . .	271
<b>23 Recipe: Service Mesh Istio . . . . .</b>	<b>272</b>
23.1 What Is a Service Mesh? . . . . .	272
23.2 Example . . . . .	273
23.3 How Istio Works . . . . .	277
23.4 Monitoring with Prometheus and Grafana . . . . .	279
23.5 Tracing with Jaeger . . . . .	283
23.6 Logging . . . . .	286
23.7 Resilience . . . . .	288
23.8 Challenges . . . . .	292
23.9 Benefits . . . . .	294
23.10 Variations . . . . .	294
23.11 Experiments . . . . .	295
23.12 Conclusion . . . . .	295
<b>24 And Now What? . . . . .</b>	<b>297</b>

## CONTENTS

<b>Appendix A: Installation of the Environment . . . . .</b>	<b>298</b>
<b>Appendix B: Maven Commands . . . . .</b>	<b>299</b>
<b>Appendix C: Docker and Docker Compose Commands . . . . .</b>	<b>301</b>

# 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](#)<sup>1</sup> 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](#),<sup>2</sup> 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](#)<sup>3</sup>.

## 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.

---

<sup>1</sup><http://microservices-book.com>

<sup>2</sup><http://microservices-book.com>

<sup>3</sup><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* exist 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

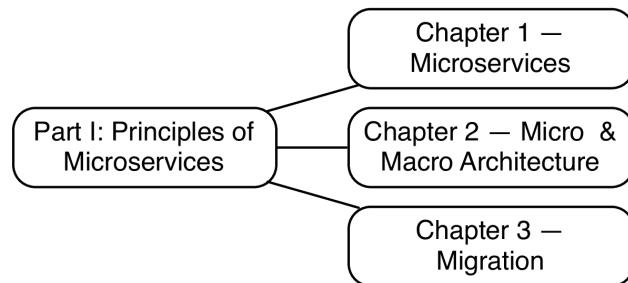
Sample code is provided for almost all the technologies presented in this book. If the reader wants to really understand the technologies, it makes sense to browse the code. It also makes sense to look at the code to understand how the concepts are actually implemented. The reader might even want to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

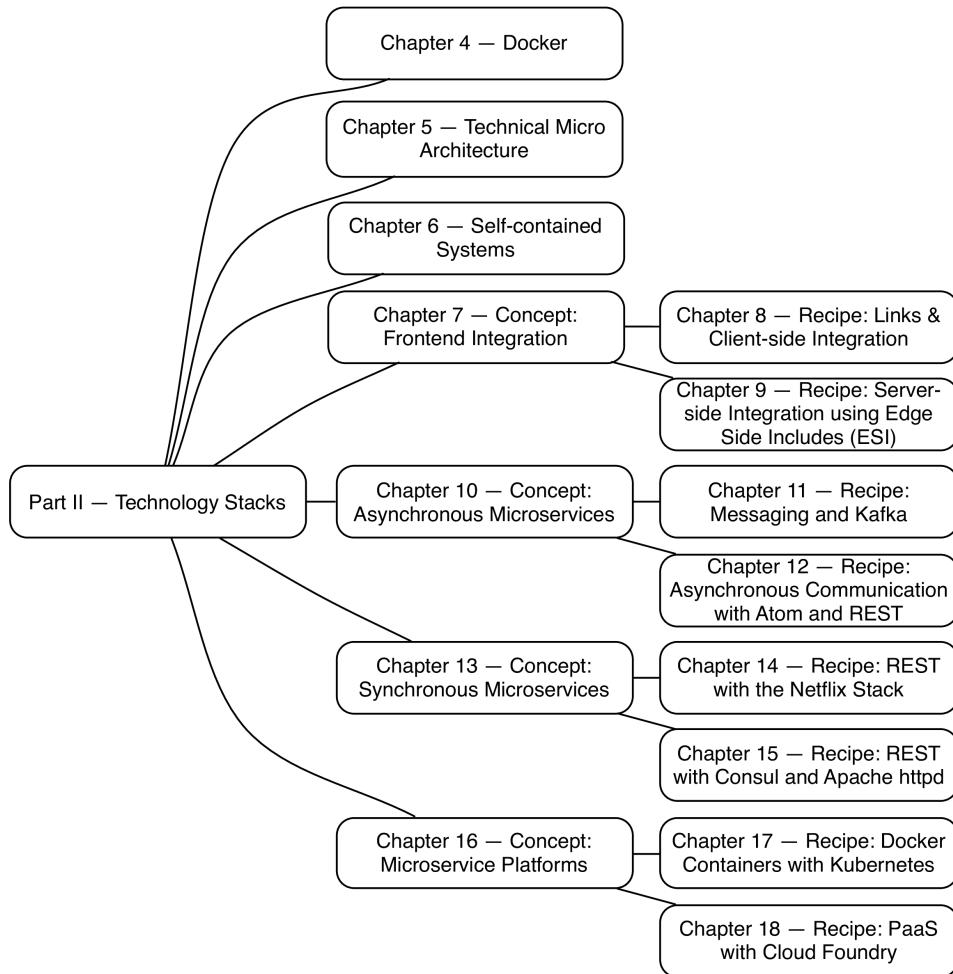


Overview of Part I

- [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](#).



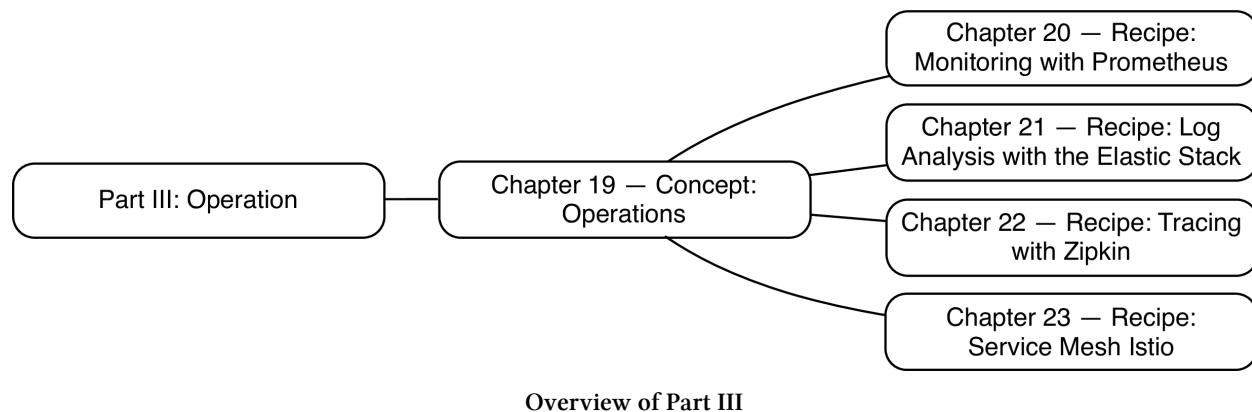
## Overview of Part II

- 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.



- [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.sofcover.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.

# 1 Microservices

This chapter provides an introduction to *microservices* and discusses:

- [Advantages](#) and [disadvantages](#) of microservices to enable the reader to evaluate the applicability and usefulness of this architecture for a specific project.
- The discussion of benefits explains which problems microservices can solve and how this architecture can be adapted for different scenarios.
- The discussion of disadvantages illustrates where technical challenges and risks lie and how these can be dealt with.
- Recognizing benefits and disadvantages is critical for technology and architecture decisions since those have to be aimed at maximizing benefits and reducing disadvantages.

## 1.1 Microservices: Definition

Unfortunately, there is no universally acknowledged definition for the term *microservice*. In the context of this book the following definition will be used:

Microservices are independently deployable modules.

For example, an e-commerce system can be divided into modules for ordering, registration and product search. Normally, all of these modules would be implemented together in one application. In this case, a change in one of the modules can only be brought into production by bringing a new version of the entire application with all its modules into production. However, when the modules are implemented as microservices, the ordering process cannot only be changed independently of the other modules, but it can even be brought into production independently.

This speeds up deployment and reduces the number of necessary tests since only a single module needs to be deployed. Due to this greater level of decoupling, a large project can be turned into a number of small projects. Each project is in charge of an individual microservice.

To achieve this at the technical level, every microservice has to be an independent process. An even better solution for decoupling microservices is to provide an independent virtual machine or Docker container for each microservice. In that case, a deployment will replace the Docker container of an individual microservice with a new Docker container, then start the new version and direct requests to this new version. The other microservices will not be affected if such an approach is used.

## Advantages of This Microservice Definition

This definition of microservices as independently deployable modules has several advantages:

- It is very *compact*.
- It is very *general* and covers all kinds of systems which are commonly denoted as microservices.
- The definition is based on *modules* and thus on an old, well-understood concept. This allows one to adopt many ideas concerning modularization. Besides, this definition highlights that microservices are part of a larger system and cannot function all on their own. Microservices necessarily have to be integrated with other microservices.
- The independent deployment is a feature which creates numerous [advantages](#) and is therefore very important. Thus, the definition, in spite of its brevity, explains what the most *essential feature* of a microservice really is.

## Deployment Monolith

A system which is not made up of microservices can only be deployed in its entirety. Therefore, it is called a *deployment monolith*. Of course, a deployment monolith can be divided into modules. The term *deployment monolith* does not make a statement about the internal structure of the system.

## Size of a Microservice

The given definition of microservices does not say anything about the size of a microservice. Of course the term *microservice* suggests that especially small services are meant. However, in practice, microservices can vary hugely in size. Some microservices keep an entire team busy, while others comprise only a few hundred lines of code. Thus, the size of microservices, even though it figures as part of the name, is ill suited to be part of the definition.

## 1.2 Reasons for Microservices

There are a number of reasons for using microservices.

### Microservices for Scaling Development

One reason for the use of microservices is the easy scalability of development. Large teams often have to work together on complex projects. With the help of microservices, the projects can be divided into smaller units which can work largely independently of each other.

- For example, the teams responsible for an individual microservice can make most technology decisions on their own. When the microservices are delivered as Docker containers, each Docker container only has to offer an interface for the other containers. The internal structure

of the containers does not matter as long as the interface is present and functions correctly. Therefore, it is for example irrelevant in which programming language a microservice is written. Consequently, the responsible team can make this decision on its own. Of course, the selection of programming languages can be restricted in order to avoid increased complexity. However, even if the choice of programming language in a project has been restricted, a team can still independently use an updated library with a bug fix for their microservice.

- When a new feature only requires changes in one microservice, it cannot only be developed independently, but it can also be brought into production on its own. This allows the teams to work on features completely independently.

Thus, with the help of microservices, teams can act independently in regards to domain logic and technology. This minimizes the coordination effort required for large projects.

## Replacing Legacy Systems

The maintenance of a legacy system is frequently a challenge since the code is often badly structured and changes can often not be checked by tests. In addition, developers might have to deal with outdated technologies.

Microservices help when working with legacy systems since the existing code does not necessarily have to be changed. Instead, new microservices can replace parts of the old system. This requires an integration between the old system and the new microservices – for example, via data replication, REST, messaging, or at the level of UI. Besides, problems such as a uniform single sign-on for the old system and the new microservices have to be solved.

But then the microservices are very much like a greenfield project. No pre-existing codebase has to be used. In addition, developers can employ a completely different technology stack. This immensely facilitates work compared to having to modify the legacy code itself.

## Sustainable Development

Microservice-based architectures promise that systems remain maintainable even in the long run.

An important reason for this is the replaceability of microservices. When a microservice cannot be maintained any longer, it can be rewritten. Compared to changing a deployment monolith, this entails less effort because the microservices are much smaller.

However, it is difficult to replace a microservice on which numerous other microservices depend since changes might affect the other microservices. Thus, to achieve replaceability, the dependencies between microservices have to be managed appropriately.

Replaceability is a great strength of microservices. Many developers work on replacing legacy systems. However, when a new system is designed, the question of how to replace this system after it has turned into a legacy system is asked much too rarely. Microservices with their replaceability provide an answer.

To achieve maintainability, the dependencies between the microservices have to be managed in the long term. Classical architectures often have difficulties at this level. A developer writes some new code. In doing so, she/he might unintentionally introduce a new dependency between two modules, which, in actual fact, had been forbidden in the architecture. Typically, the developer does not even notice the mistake because she/he only pays attention to the code level of the system and not to the architectural level. Often, it is not immediately clear which module a class belongs to. So it is also unclear to which module the developer just introduced a dependency. In this manner, more and more dependencies are introduced over time. The originally designed architecture is more and more violated, culminating in a completely unstructured system.

Microservices have clear boundaries due to their interface<sup>4</sup> irrespective of whether the interface is implemented as REST interface or via messaging. When a developer introduces a new dependency to such an interface, he or she will notice this because the interface has to be called appropriately. For this reason, it is unlikely that architecture violations will occur at the level of dependencies between microservices. The interfaces between microservices are in a way architecture firewalls since they prevent architecture violations. The concept of architecture firewalls is also implemented by architecture management tools like [Sonargraph<sup>4</sup>](#), [Structure101<sup>5</sup>](#), or [jQAssistant<sup>6</sup>](#). Advanced module concepts can also generate such a firewall. In the Java world, [OSGi<sup>7</sup>](#) limits access and visibility between modules. Access can even be restricted to individual packages or classes.

Therefore, individual microservices remain maintainable. If the code of a microservice is unmaintainable, it can just be replaced. That would not influence any of the other microservices. The architecture at the level of dependencies between microservices remains maintainable, too. Developers cannot unintentionally add dependencies between microservices. Therefore, microservices can ensure a high architecture quality in the long term both inside each microservice and between the microservices. Thus, microservices enable sustainable development where the speed of change does not decline over time.

## Continuous Delivery

[Continuous delivery<sup>8</sup>](#) is an approach where software is continuously brought into production with the help of a continuous delivery pipeline. The pipeline brings the software into production via the different phases (see figure 1-1).



Fig. 1-1: Continuous Delivery Pipeline

Typically, the software is compiled, and unit tests and a static code analysis are performed in the commit phase. In the acceptance test phase, automated tests assure the correctness of the software

<sup>4</sup><https://www.hello2morrow.com/products/sonargraph>

<sup>5</sup><http://structure101.com/>

<sup>6</sup><https://jqassistant.org/>

<sup>7</sup><https://www.osgi.org/>

<sup>8</sup><http://continuous-delivery-book.com/>

in regards to domain logic. Capacity tests check the performance at the expected load. Explorative tests serve to perform not-yet-considered tests or to examine new functionalities. In this manner, explorative tests can analyze aspects which are not yet covered by automated tests. In the end, the software is brought into production.

Microservices represent independently deployable modules. Therefore each microservice has its own continuous delivery pipeline.

This facilitates continuous delivery.

- The continuous delivery pipeline is significantly *faster* because the deployment units are smaller. Consequently, deployment is faster. Continuous delivery pipelines contain many test stages. The software has to be deployed in each stage. Faster deployments speed up the tests and therefore the pipeline. The tests are also faster because they need to cover fewer functionalities. Only the features in the individual microservice have to be tested, whereas in case of a deployment monolith, the entire functionality has to be tested due to possible regressions.
- Building up a continuous delivery pipeline is *easier* for microservices. Setting up an environment for a deployment monolith is complicated. Most of the time, powerful servers are required. In addition, third-party systems are frequently necessary for tests. A microservice requires less powerful hardware. Besides, not so many third-party systems are needed in the test environments. However, running all microservices together in one integration test can cancel out this advantage. An environment suitable for running all microservices would require powerful hardware as well as an integration with all third-party systems.
- The deployment of a microservice poses a *smaller risk* than the deployment of a deployment monolith. In case of a deployment monolith, the entire system is deployed anew; in the case of a microservice, only one module. This causes fewer problems since less of the functionality is being changed.

In summary, microservices facilitate continuous delivery. Just their better support of continuous delivery can already be reason enough to migrate a deployment monolith to microservices.

However, microservices architectures can only work when the deployment is automated. Microservices increase the number of deployable units substantially compared to a deployment monolith. This is only feasible when the deployment processes are automated.

Really independent deployment means that the continuous delivery pipelines have to be completely independent. Integration tests conflict with this independence. They introduce dependencies between the continuous delivery pipelines of the individual microservices. Therefore, integration tests have to be reduced to the minimum. Depending on the type of communication, there are different approaches to achieve this for **synchronous** and **asynchronous communication**.

## Robustness

Microservices systems are more robust. When a memory leak exists in a microservice, only this microservice crashes. The other microservices keep running. Of course, they have to compensate

the failure of the crashed microservice. This is called resilience. To achieve resilience, microservices can, for example, cache values and use them in case of a problem. Alternatively, there might be a fallback to a simplified algorithm.

Without resilience, the availability of a microservices system might be a problem. It is likely that a microservice fails. Due to the distribution into several processes, many more servers are involved in the system. Each of these servers can potentially fail. Communication between microservices occurs via the network, which can also fail. Therefore, microservices need to implement resilience to achieve robustness.

The [section about Hystrix](#) illustrates how resilience can concretely be implemented in a synchronous microservices system.

## Independent Scaling

Most of the time, scaling the whole system it is not required. For example, for a shop system during Christmas, the catalog might be the most critical and hardware-consuming part. By scaling the complete system, hardware is spent on parts which don't require more power.

Each microservice can be independently scaled. It is possible to start additional instances of a microservice and to distribute the load of the microservice onto the instances. This can improve the scalability of a system significantly. So, in the previous example, just the catalog would need to be scaled up. For this to work, the microservices naturally have to fulfill certain requirements. For example, they must be stateless. Otherwise, requests of a specific client cannot be transferred to another instance, because this instance then would not have the state specific for that client.

It can be difficult to start more instances of a deployment monolith due to the required hardware. Besides, building up an environment for a deployment monolith can be complex. This can require additional services or a complex infrastructure with databases and additional software components.

In the case of a microservice, the scaling can be more fine grained so that normally fewer additional services are necessary and the basic requirements are less complex.

## Free Technology Choice

Each microservice can be implemented with an individual technology. This facilitates the migration to a new technology since each microservice can be migrated individually. In addition, it is simpler and less risky to gain experience with new technologies since they can initially be used for only a single microservice before they are employed in several microservices.

## Security

Microservices can be isolated from each other. For example, it is possible to introduce firewalls into the communication between microservices. Besides, the communication between microservices can be encrypted to guarantee that the communication really originates from another microservice and is authentic. This prevents the corruption of additional microservices if a hacker takes over one microservice.

## In General: Isolation

In the end, many advantages of microservices can be traced back to a stronger isolation.

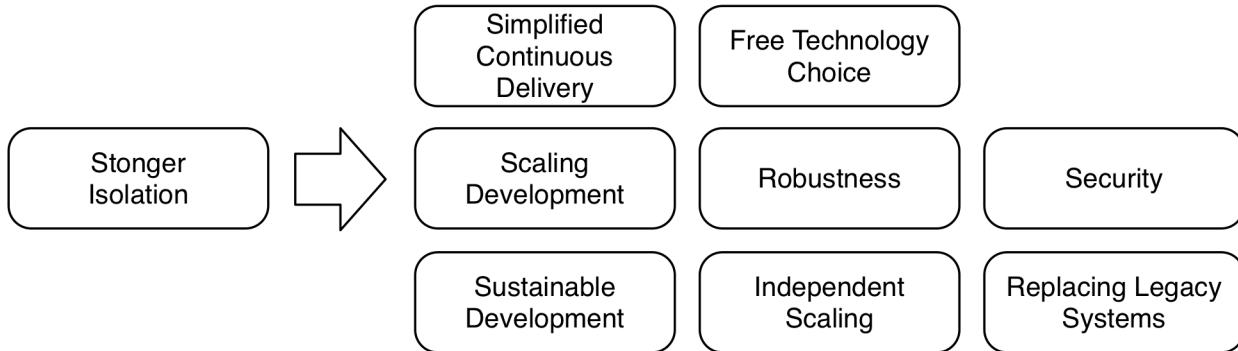


Fig. 1-2: Isolation as the Source of the Advantages of Microservices

Microservices can be deployed in isolation, which facilitates continuous delivery. They are isolated in respect to failures, which improves robustness. The same is true for scalability. Each microservice can be scaled independently of the other microservices. The employed technologies can be chosen for each microservice in isolation, which allows for free technology choice. The microservices are isolated in such a way that they can only communicate via the network. Therefore, the communication can be safeguarded by firewalls, which increases security.

Due to this strong isolation, the boundaries between modules can hardly be violated by mistake. The architecture is rarely violated; this safeguards the architecture. A microservice can in isolation be replaced with a new microservice. This enables the low-risk replacement of microservices and allows one to keep the architecture of the individual microservices clean. Thus, the isolation facilitates the long-term maintainability of the software.

Decoupling is an important feature of modules. With their isolation, microservices push it to extremes. Modules are normally only decoupled in regards to code changes and architecture. The decoupling between microservices goes far beyond that.

So in the end for many purposes, microservices are just smaller and can be handled in isolation due to the decoupling. That also makes it easier to reason about them; The security of a microservice is easier to verify, the performance is easier to measure, and it is easier to figure out whether they work correctly. That makes the design and also the development easier.

## Prioritizing Advantages

Which of the discussed reasons for switching to microservices is the most important depends on the individual scenario. The use of microservices in a greenfield system is the one exception. More often a deployment monolith is replaced by a microservices system (see also [chapter 4](#)). In that case, different advantages are relevant.

- The easier *scaling of development* can be an important reason for the introduction of microservices in such a scenario. Often, it is impossible to work with a large number of developers on

a single deployment monolith sufficiently fast.

- The *easy migration* away from the legacy deployment monolith facilitates the introduction of microservices in such a scenario.
- *Continuous delivery* is often an additional goal. The aim is to increase the speed and reliability with which changes can be brought into production.

The scaling of development is not the only scenario for a migration. When a single Scrum team wants to implement a system with microservices, scaling of development cannot be a sensible reason since the organization of development is not sufficiently large for this. However, also other reasons are possible. Continuous delivery, technical reasons like robustness, independent scaling, free technology choice, or sustainable development all can play a role in such a scenario.

At the end, it is important to focus on the business value increase. Depending on the scenario, an advantage in one of the previously mentioned areas might make the company more profitable or competitive – for example, faster time to market or better reliability of the system.

## **Microservices Are a Trade-Off**

Dependent on the aims, a team can compromise when implementing microservices. When robustness is a goal of introducing microservices, the microservices have to be implemented as separate Docker containers. Each Docker container can crash without affecting the other ones. If robustness does not matter, other alternatives can be considered. For example, multiple microservices can run together as Java web applications in one Java application server. In this case, they all run in one process and therefore are not isolated in respect to robustness. A memory leak in any of the microservices will cause them all to fail. However, such a solution is easier to operate and therefore might be the better trade-off in the end.

## **Two Levels of Microservices: Domain and Technical**

The technical and organizational advantages point to two levels at which a system can be divided into microservices.

- A *coarse-grained division by domain* enables the teams to develop independently, and allows them to roll out a new feature with the deployment of a single microservice. For example, in an e-commerce system, the customer registration and the order process can be examples of such coarse-grained microservices.
- For *technical reasons* some microservices can be *further divided*. When for example the last step of the order process is under especially high load, this last step can be implemented in a separate microservice. This microservice then can be scaled independently of the other microservices. The microservice belongs to the domain of the order process, but for technical reasons is implemented as a separate microservice.

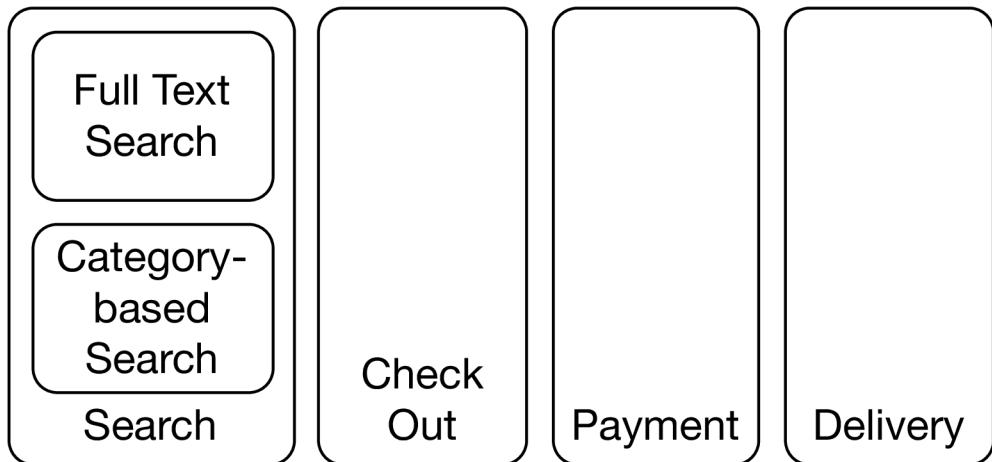


Fig. 1-3: Two Levels of Microservices

[Figure 1-3](#) shows an example for the two levels. Based on the domains, an e-commerce application is divided into the microservices search, check out, payment, and delivery. Search is further subdivided. The full-text search is separated from the category-based search. Independent scaling can be one reason for this. This architecture allows the system to scale the full-text search independently of the category-based search which is advantageous when both have to deal with different levels of load. Another reason could be the use of different technologies. The full-text search can be implemented with a full text search engine, which is unsuitable for a category-based search.

### Typical Numbers of Microservices in a System

It is difficult to state a typical number of microservices per system. Based on the divisions discussed in this chapter, 10-20 coarse-grained domains are usually defined, and each of these might be subdivided into one to three microservices. However, there are also systems with far more microservices.

## 1.3 Challenges

Microservices not only have benefits, they also pose challenges.

- The *operation* of a microservices system requires more effort than running a deployment monolith. This is due to the fact that in a microservices system, many more deployable units exist that all have to be deployed and monitored. This is feasible only when the operation is largely automated and the correct functioning of the microservices is guaranteed via appropriate monitoring. [Part III](#) demonstrates different solutions to deal with this challenge.
- Microservices have to be *independently deployable*. Dividing them, for example, into Docker containers, is a prerequisite for this, but it is not enough on its own. Also, the testing has to be independent. When all microservices have to be tested together, one microservice can block the test stage and thus prevent the deployment of the other microservices. This means that testing might become harder. Due to the split into microservices, there are more interfaces to test, and

testing has to be independent for both sides of the interface. Changes to interfaces have to be implemented in such a way that an independent deployment of individual microservices is still possible. For example, the microservice which implements the interface has to offer the new and the old interface. Then this microservice can be deployed without requiring that the calling microservice be deployed at the same time.

- *Changes which affect multiple microservices* are more difficult to implement than changes which concern several modules of a deployment monolith. In a microservices system, such changes require several deployments. These deployments have to be coordinated. In the case of a deployment monolith, only one deployment would be necessary.
- In a microservices system, the *overview* of the microservices can get lost. However, experience teaches that in practice a sound domain-based division can restrict changes to one or a few microservices. Therefore, the overview of the system is less important in the end because the interaction between the microservices hardly influences development due to the high degree of independence.
- Microservices communicate through the *network*. Compared to local communication, the latency is much higher and it is also more likely that the communication will fail. So a microservices system cannot rely on the availability of other microservices. This makes the systems more complex.

## Weighing Benefits and Disadvantages

The most important rule is that microservices should only be used if they represent the most simple solution in a certain scenario. The previously mentioned benefits should outweigh disadvantages resulting, for example, from the higher level of complexity for deployment and operation. After all, it hardly makes sense to intentionally decide for a too-complex solution.

## 1.4 Variations

Dependent on the concrete scenario, microservice variants such as [Self-contained Systems](#) can be used.

### Technological Variations

[Part II](#) and [part III](#) show different technological variations such as synchronous communication, asynchronous communication, and UI integration. By combining one or multiple recipes out of these parts, a custom microservices architecture can be designed.

### Experiments

The following approach helps to find the right recipes.

- Identify the problems in your current system (for example, resilience, development agility, too slow deployment, and so on)

- For one of the projects that you know, prioritize the benefits of using microservices.
- Weigh which of the challenges in this project could pose a risk.
- Afterwards, the possible technical and architectural solutions in the following chapters can be compared with respect to whether they provide a sensible solution in the context of these requirements.

For the concrete division into microservices and for technical decisions, additional concepts are necessary. Therefore, the question of how best to divide a system into microservices is the focus of section 2.7.

## 1.5 Conclusion

Microservices represent an extreme type of modularization. Their separate deployment is the foundation for a very strong degree of decoupling.

This results in numerous advantages. A crucial benefit is isolation at different levels. This not only facilitates deployment, but also limits potential failures to individual microservices. Microservices can be individually scaled, technology decisions affect only individual microservices, and security problems can also be restricted to individual microservices. The isolation allows one to more easily develop a microservices system with a large team because less coordination between teams is required. In addition, the smaller deployment artefacts make continuous delivery easier. Moreover, replacing a legacy system is much easier with microservices because new microservices can supplement the system without the necessity of large code changes in the legacy system.

The challenges are mostly associated with operation. Appropriate technological decisions should strengthen the intended benefits, and at the same time they should minimize disadvantages like the complexity in operation.

Of course, integration and communication between microservices is more complex than the calls between modules within a deployment monolith. The added technological complexity represents an additional important challenge for microservices architectures.

# 2 Micro and Macro Architecture

Microservices provide much better decoupling. Therefore, they help to modularize and isolate software modules (see [section 1.2](#)). However, microservices are modules of a larger system. Therefore, they have to be integrated. This poses a challenge for the architecture. On the one hand, the architecture has to ensure that the microservices can work together to form the overall system. On the other hand, the freedom of the microservices may not be restricted too much since this would compromise their necessary isolation and independence which are required for most of the benefits of a microservices architecture.

For this reason, it is advisable to divide the architecture into a micro and a macro architecture. The *micro architecture* comprises all decisions that can be made individually for each microservice. The *macro architecture* consists of all decisions which have to be made at a global level and apply to all microservices.

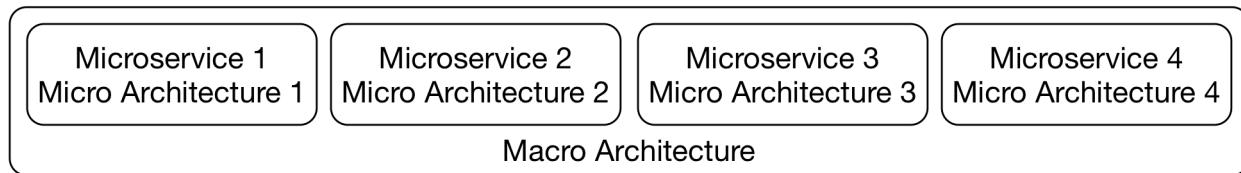


Fig. 2-1: Micro and Macro Architecture

[Figure 2-1](#) illustrates this idea. The overarching macro architecture applies to all microservices, whereas the micro architecture deals with individual microservices, so that each microservice has its own micro architecture.

This chapter illustrates the following:

- The *division of domain logic* into microservices. *Domain-driven design* and *bounded context* are great approaches for such a division.
- The decisions that are part of the *technical micro and macro architecture* and how a *DevOps model* affects these decisions.
- The question *who* divides the decisions into micro and macro architecture and creates the macro architecture.

## 2.1 Bounded Context and Strategic Design

In regards to the domain architecture, the concept of micro and macro architecture has long been common practice. A macro architecture divides the domains into coarse-grained modules. These modules are further divided as part of the micro architecture.

For example, an e-commerce system can be divided into modules for *customer registration* and *order process*. Order process can be further divided into smaller modules such as, for instance, *data validation* or *freight charge calculation*. The internal architecture of the order process module is hidden from the outside and therefore can be altered without affecting other modules. This flexibility to change one module without influencing the other modules is one of the main advantages of modular software development.

### An Example for a Domain Architecture

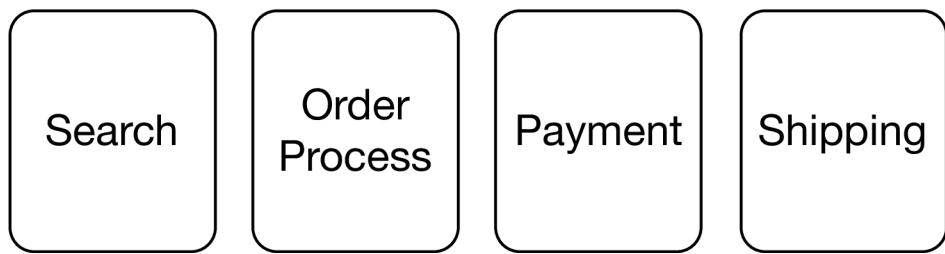


Fig. 2-2: Example for a Split into Bounded Contexts

Figure 2-2 shows an example for the division of a system into multiple domain modules. In this division, each module has its own domain model.

- For *search* data such as descriptions, images or prices have to be stored for products. Important customer data are, for example, the recommendations that can be determined based on past orders.
- *Order process* has to keep track of the contents of the shopping cart. For products, only basic information is required such as name and price. Similarly, not too much data concerning the customer is necessary. The most important component of the domain model of this module is the shopping cart. It is then turned into an order that has to be handed over and processed by the other bounded contexts.
- For *payment*, the payment-associated information like credit card numbers has to be kept for each customer.
- For *shipping*, the delivery address is required information about the customer and size and weight are necessary information about the product.

This list reflects that the modules require different domain models. Not only the data concerning customer and product differ, but the entire model and also the logic.

### Domain-driven Design: Definition

Domain-driven design (DDD) offers a collection of patterns for the domain model of a system. For microservices, the patterns in the area of strategic design are the most interesting. They describe how a domain can be subdivided. Domain-driven design offers many more patterns that, for example,

facilitate the model of individual modules. The original DDD book<sup>9</sup> provides a lot more information. It introduces the term “domain-driven design” and comprehensively describes DDD. The more compact book “Domain-driven Design Distilled”<sup>10</sup> focuses on design, bounded context, and domain events. The [Domain-Driven Design Reference](#)<sup>11</sup> is also by the author of the original DDD book. It contains all DDD patterns but without any additional explanation or examples.

### Bounded Context: Definition

Domain-driven design speaks of a *bounded context*. Each domain model is valid only in a bounded context. Consequently, *search*, *order process*, *payment*, and *shipping* are such bounded contexts because they each have their own domain model.

It would be conceivable to implement a domain model that comprises multiple bounded contexts from the example. However, such a model would not be the easiest solution. For example, a price change affects search; however, it must not result in a price change for orders that have already been processed in payment. It is easier to only store the current price of a product in the bounded context search, and to store in payment the price of the product in each order, which can also comprise rebates and other complex logic. Therefore, the simplest design consists of multiple specialized domain models that are valid only in a certain context. Each domain model has its own model for business objects such as customers or products.

### Strategic Design

The division of the system into different bounded contexts is part of *strategic design*, which belongs to the practices of domain-driven design (DDD). Strategic design describes the *integration* of bounded contexts.

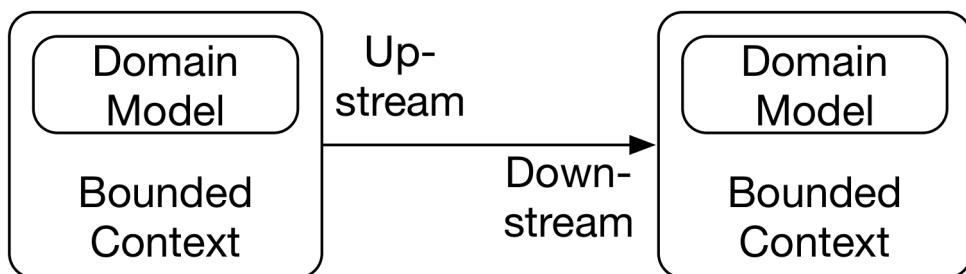


Fig. 2-3: Fundamental Terms of Strategic Design

[Figure 2-3](#) shows the fundamental terms of strategic design.

- The *bounded context* is the context in which a specific *domain model* is valid.
- The bounded contexts depend on each other. Usually, each bounded context is implemented by one team. The *upstream* team can influence the success of the *downstream* team. However, the

<sup>9</sup>Eric Evans: Domain Driven Design: Tackling Complexity in the Heart of Software, Addison Wesley, 2003, ISBN 978 0 32112 521 7

<sup>10</sup>Vaughn Vernon: Domain Driven Design Distilled, Addison-Wesley Professional, 2016, ISBN 978 0 13443 442 1

<sup>11</sup><https://domainlanguage.com/ddd/reference/>

downstream team cannot influence the success of the upstream team. For example, the team responsible for payment depends on the information that the order process teams provides. If data such as prices or credit card numbers are not part of the order, it is impossible to do the payment. However, the order process does not depend on the payment to be successful. Therefore, the order processing is upstream. It can make payment fail. Payment is therefore downstream and also cannot make the order process fail.

## Strategic Design Patterns

DDD describes in several patterns how exactly communication takes place. These patterns not only describe the architecture, but also the cooperation within the organization.

- With the *customer/supplier* pattern, the supplier is upstream. The customer is downstream. However, the customer can factor their priorities into the planning of the upstream project. In [figure 2-4](#), payment uses the model of the order process. However, payment defines requirements for the order process. Payment can only be done successfully if the order process provides the required data. So, payment can become a customer of the order process. That way the customer's requirements can be included in the planning of the order process.

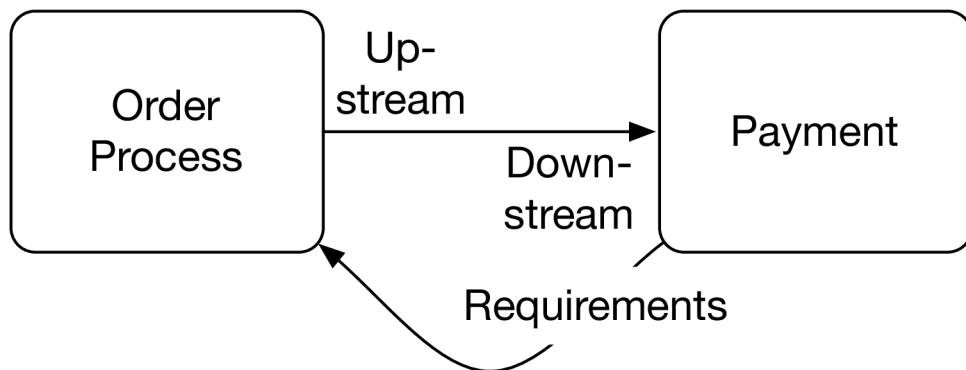


Fig. 2-4: Customer / Supplier Pattern

- Conformist* means that a bounded context simply uses a domain model from another bounded context. In [figure 2-5](#), bounded context statistics and order process both use the same domain model. The statistics are part of a data warehouse. They use the domain model of the order process bounded context and extract some information relevant to store in the data warehouse. However, with the *conformist* pattern, the data warehouse team does not have a say in case of changes to the bounded context. So the data warehouse team could not demand additional information from the other bounded context. However, it is still possible that they would receive additional information out of altruism. Essentially, the data warehouse team is not deemed important enough to get a more powerful role.

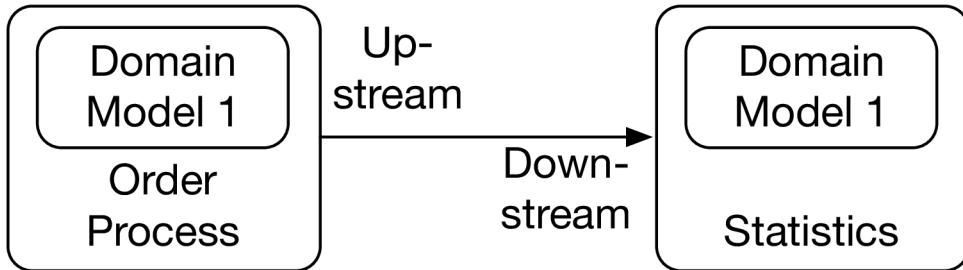


Fig. 2-5: Conformist: Domain Model Used in Other Bounded Context

- In case of an *anti-corruption layer* (ACL), the bounded context does not directly use the domain model of the other bounded context, but it contains a layer for decoupling its own domain model from the model of the bounded context. This is useful in conjunction with *conformist* to generate a separate model decoupled from the other model. Figure 2-6 shows that the bounded context *shipping* uses an ACL at the interface to bounded context *legacy*, so that both bounded contexts have their own independent domain models. That makes sure that the model in the legacy system does not affect the bounded context shipping. Shipping can implement a clean model in its bounded context.

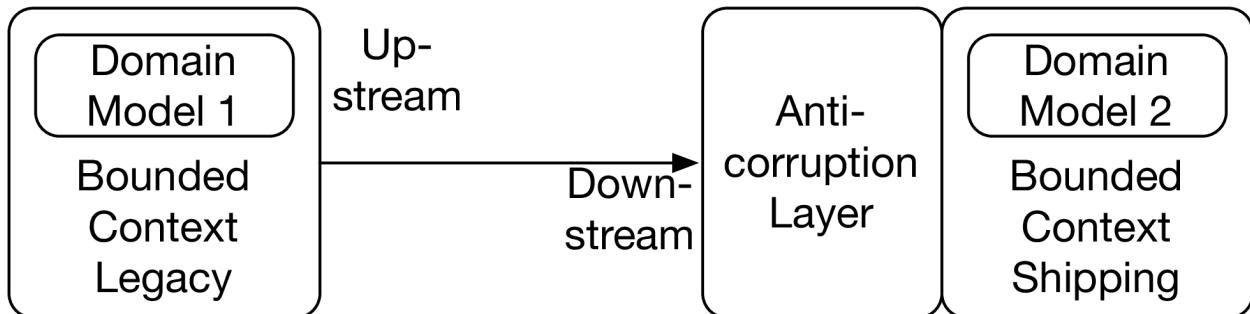


Fig. 2-6: Anti-corruption Layer with Conformist

- With *separate ways*, the bounded contexts do not have a relation at the software level although a relation would be conceivable. Let's assume that in the e-commerce scenario a bounded context purchasing for the purchase department is added. This bounded context could collect the data for listing products, but it is implemented differently. With separate ways, the purchasing would be separate from the remaining system. When the goods are delivered, a user would use another bounded context like listing to enter the necessary data and list the products. The purchasing causes the shipping, which in turn triggers the delivery, and thereby triggers the user to list the product with a different bounded context. The shipping of the products is an event in the real world. In the software, the systems are separate. Consequently, the systems are independent and can be evolved completely independently.



Fig. 2-7: Separate Ways

- *Shared kernel* describes a common core which is shared by multiple bounded contexts. In figure 2-8 domain model order process and payment possess a shared kernel. Data of a customer could be an example for such a scenario. However, *shared kernel* comprises shared business logic and shared database schemata and therefore should not be used in a microservices environment. It is an anti-pattern for microservices systems. But because DDD can also be applied to deployment monoliths, there are still scenarios in which a shared kernel makes sense.

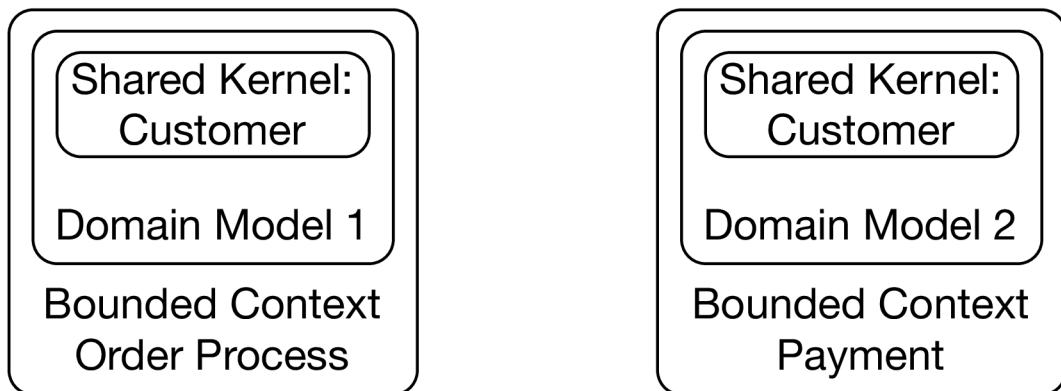


Fig. 2-8: Shared Kernel

Some patterns are primarily useful in cases where more than one bounded context has to be integrated.

- *Open host service* means that the bounded context offers a generic interface with several services. Other bounded contexts can implement their own integration with these services. This pattern is frequently found at public APIs in the Internet. However, it is also a possible alternative within an enterprise.
- *Published language* is a domain model accessible by all bounded contexts. For example, this can be a standard format such as EDIFACT for transactions between companies. But it is also possible to define a data structure that is only used inside a company and published, for example, in the Wiki.

These models can be used together. The *open host service* can use *published language* for communication. For example, the order process might accept orders from external clients. Providing a specific

interface for each external client would be a lot of effort so there is a generic open host service and a published language for orders. Each external client can use this interface to submit orders to the bounded context order process.

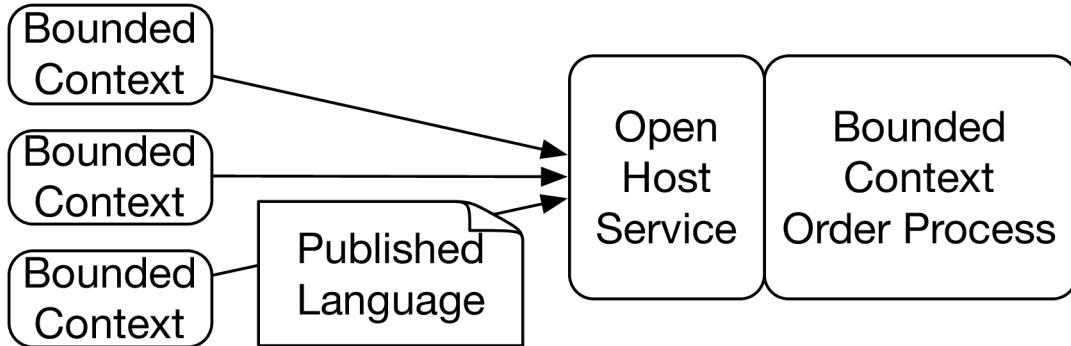


Fig. 2-9: Open Host Service and Published Language

## Selecting Patterns

The choice of patterns has to be in line with the domain, the power structure, and the communication relationships between the teams. When the bounded context payment does not obtain the necessary data from the bounded context order processing, the products can be ordered but not paid for. Therefore, the *customer/supplier* pattern is an obvious choice here. However, this is not a fact found in the domain, but rather a consequence of the power structure, which in turn depends on the business model of the company.

Of course, the selected patterns influence the effort necessary for coordination and therefore the degree of isolation between the teams. They set the rules by which the teams have to work on the integration. Thus, a pattern like *customer/supplier* might not be very desirable because it requires a lot of coordination. Still, it might be the right solution depending on domain aspects. It makes little sense to use a different pattern between payment and the order process just to have less coordination. A different pattern might make it impossible for the business to succeed.

## Domain Events between Bounded Contexts

For the communication between bounded contexts, domain events can be used. Ordering a shopping cart can be modeled as such an event. This event is triggered by the bounded context order process and is received by the bounded contexts shipping and payment to initiate shipping and invoicing of the order. Events can be useful for integrating bounded contexts. Section 10.2 discusses events from a technical perspective.

Domain events are a part of the domain model. They represent something that happened in the domain. That means they should also be relevant to domain experts.

## Bounded Contexts and Microservices

Bounded contexts divide a system by domains. They do not have to be microservices. They can also be implemented as modules in a deployment monolith. If the bounded contexts are implemented as microservices, this results in modules which are very independent at the domain and technical level. Therefore, it is sensible to combine the concepts of microservices and bounded contexts.

The dependencies of the bounded contexts as part of strategic design limit this independence. However, since the microservices are part of a larger system, dependencies between the modules cannot be completely avoided.

### Evolution

Over time, new functionalities might justify new bounded contexts. Also, it might become apparent that one bounded context should really be split in two. That might be the case because new logic is added to the bounded context, or the team understands the bounded context better. In such a case, new bounded context and therefore new microservices might be created.

As mentioned in [section 1.2](#), there are two levels of microservices: a coarse-grained division by domain, and a more fine-grained division for technical reasons. It might therefore happen that new microservices are created to, for example, simplify scalability. If a microservice is split in two, the resulting microservices are smaller and therefore easier to scale. So, such reasons might also lead to a larger number of microservices.

## 2.2 Technical Micro and Macro Architecture

Domain-driven design provides a domain macro architecture. Bounded context and strategic design are patterns for this kind of architecture. Microservices provide technological isolation. Therefore, it is possible to extend the concept of micro and macro architecture to technical decisions. For deployment monoliths, these decisions inevitably have to be taken globally. So only for microservices, technical decisions can be made within the framework of macro or micro architecture. However, some decisions have to be part of the macro architecture. Otherwise, no integrated system will be created, but rather several non-integrated islands.

### Micro or Macro Architecture Decisions

The decisions that can be taken either in the context of micro or macro architecture include:

- The *programming language* can be defined uniformly for all microservices in the macro architecture, but the decision can also be part of the micro architecture. Then each microservice can also be implemented with a different language. The same applies to frameworks and infrastructure such as application servers. If these decisions are part of the *micro architecture*, the best-suited technology can be used to solve the specific problems of each microservice. A *macro architecture* definition is useful if a company's technology strategy allows only certain technologies, or if only developers with knowledge in certain technologies should be hired.

- The *database* can also be defined as a part of the macro architecture or the micro architecture. At first glance, this decision seems to be comparable to the decision concerning the programming language. But databases are different. They store data. The loss of data is usually unacceptable. Therefore, there must be a backup strategy and a disaster recovery strategy for a database. Setting these up for many different databases can cause considerable effort. To avoid too many different databases, the database can be defined as part of the *macro architecture* for all microservices. Even if the database is defined in the macro architecture, multiple microservices must not share a database schema. That would contradict the bounded contexts (see [section 2.1](#)). The domain model in the database schema would be used by several microservices. This would couple the microservices too strongly. Even with a unified database, the microservices must have separate schemata in the database. Each microservice can also have its own instance of the database. That has an advantage. A crash of one database causes only one microservice to fail. However, the higher effort involved, especially concerning operation, is an argument against individual instances.
- If microservices have their own user interface (UI), the *look and feel* of microservices can be a micro or macro architecture decision. Often a system should have a uniform UI; therefore, the look and feel must be a macro architecture decision. But sometimes a system has different types of users – for example, back office and customers. These types of users have different requirements for the UI, which are often incompatible with a uniform look and feel. Then the look and feel must be a micro architecture decision to make sure each microservice can support its group of users best. When defining the look and feel at the macro architecture level, shared CSS and JavaScript are not enough to ensure a common style of the UI of all microservices. Uniform technical artifacts can be used to implement very different types of user interfaces. Therefore, a style guide must become part of the macro architecture. Often there are concerns that microservices with separate UIs cannot provide a consistent look and feel. But the UI can also diverge in a monolithic system. Defining appropriate style guides and artifacts is the only way to achieve a consistent look and feel for large systems, regardless of the use of microservices.
- It may be necessary to standardize the *documentation*. The documentation should be part of the micro architecture if the same team will build and maintain the microservice. Then they can decide for themselves which documentation is necessary for the long-term development. But of course, the decision about the documentation can also be part of the macro architecture. A certain level of documentation makes it easier to later hand the microservice over to another team. It may also be necessary to document certain aspects of the microservices in a uniform manner. For example, for security reasons, some systems need to keep track of the libraries used in the microservices. In the case of a security vulnerability in a specific library, it is then possible to identify which microservices need to be fixed. Standardized documentation can also provide an overview of the system and the dependencies between microservices.

## Typical Macro Architecture Decisions

There are some decisions that must always be taken at the level of macro architecture. Ultimately, all microservices together should result in a coherent system. This requires some standards.

- The *communication protocol* of the microservices is a typical macro architecture decision. Only if all microservices uniformly provide, for example, a REST interface or a messaging interface, can all microservices communicate with each other and thus form a coherent system. In addition, the data format must be standardized. It makes a difference whether systems communicate with JSON or XML, for example. The communication protocol could theoretically be a different one for each communication channel between two microservices, and thus be a micro architecture decision. In the end, however, a coherent system no longer exists. Actually, it has disintegrated into islands that communicate with each other in different ways.
- With *authentication*, a user proves that he or she has a certain identity. This can be done with a password and a user name, for example. Since it is unacceptable for the user to re-authenticate with every microservice, the entire microservices system should use a single authentication system. The user then enters a user name and password once and can then use any microservice.
- Integration testing technology is also a typical macro architecture decision. All microservices must be tested together, and so they have to run together in an integration test. The macro architecture must define the necessary prerequisites for this.

## Typical Micro Architecture Decisions

Certain decisions should be taken for each microservice individually. They are therefore typically part of the micro architecture.

- The *authorization* of the user determines what a user is allowed to do. Like authentication, authorization belongs to the area of security. The authorization should be done in the respective microservice. Authorization is closely linked to domain logic. Which user is allowed to initiate a certain action is part of the domain logic, and therefore belongs to the microservice like the other domain logic. Otherwise, the domain logic would be implemented in a microservice itself, but the decision about which part of the domain logic is available to a user would be made centrally. This makes no sense, especially with complex rules. For example, if orders up to a certain upper limit can be triggered by certain users, authorization, concrete upper limits, and possible exceptions belong to the microservice *order*. Authentication assigns the user roles that can be used in authorization. For example, a microservice can define which actions a user with the role *customer* can trigger and which actions a user with the role *call center agent* can trigger.
- The *testing* can be different for each microservice. Even the tests are ultimately part of the domain logic. In addition, there may be different non-functional requirements for each microservice. For example, one microservice can be particularly performance-critical, whereas another can be more safety-critical. These risks must be covered by an individual focus in the tests.
- Since the tests can be different, the *continuous delivery pipeline* is also different for each microservice. It must include the relevant tests. Of course, the technology for the continuous delivery pipeline can be standardized. For example, each pipeline can use a tool like Jenkins. What happens in the respective pipelines, however, depends on the respective microservice.

The following table shows the typical micro and macro architecture decisions:

Micro or Macro	Micro Architecture	Macro Architecture
Programming Language	Continuous Delivery Pipeline	Communication Protocol
Database	Authorization	Authentication
Look and Feel	Tests of the Microservice in Isolation	Integration Tests
Documentation		

## 2.3 Operation: Micro or Macro Architecture?

Some decisions in the area of micro and macro architecture influence mostly the operation of the applications. These include:

- In *configuration*, it is necessary to define an interface with which a microservice can obtain its configuration parameters. This includes both technical parameters (such as thread pool sizes) and parameters for the domain logic. For example, a microservice can get these settings via an environment variable or read them from a configuration file. This decision affects only how the microservice obtains the configuration. The decision of how to store and generate the configuration data is independent of this. The data can be stored in a database, for example. Either configuration files or environment variables can be generated from the data in the database. Note that the information on which computer and under which port a microservice can be reached, does not belong to the configuration, but to the service discovery (see [section 13.3](#)). Configuring passwords or certificates is also a challenge that can be solved with other tools. To do this, [Vault<sup>12</sup>](#) is a good choice, because this information must be stored in a particularly secure way and must be visible to as few employees as possible in order to prevent unauthorized access to production data.
- *Monitoring* is about the technology that tracks metrics (see [chapter 20](#)). Metrics provide information about the state of a system. Examples include the number of requests processed per second or business metrics such as revenue. The question of which technology is used to track the metrics is independent of which metrics are captured. Every microservice has different interesting metrics, because every microservice has different challenges. For example, if a microservice is under very high load, then performance metrics are useful.
- *Log analysis* defines a tool for managing logs (see [chapter 21](#)). Although logs were originally stored in log files, they are now stored on specialized servers. This makes it easier to analyze and search the logs, even with large amounts of data and many microservices. In addition, new instances of a microservice can be started when the load increases and can be deleted again after the load decreases. In this case, the logs of this microservice instance should still be available, even if the microservice was deleted long ago due to a decreasing load. If the logs are just stored on a local device, the logs would be gone after the microservice has been deleted.

---

<sup>12</sup><https://www.vaultproject.io/>

- *Deployment technology* determines how the microservices are rolled out. For example, this can be done with Docker images (see [chapter 5](#)), Kubernetes Pods (see [chapter 17](#)), a PaaS (see [chapter 18](#)), or installation scripts.

These decisions define how a microservice behaves from an operation point of view. Typically, these decisions are either all part of the macro architecture or all part of the micro architecture.

## Operation Macro Architecture with Separate Operations Teams

Whether decisions in the area of operation belong to micro or macro architecture depends on the organization. For example, a team can develop microservices, but bear no responsibility for their operation. The operations team is responsible for the operation of all microservices. In this scenario, decisions for operation must be made at the level of macro architecture. It is generally unacceptable for the operations team to have to learn a different approach for the operation of each microservice, especially because the number of microservices is much larger than the number of deployment monoliths would be for the same project.

Another reason for a macro architecture decision for the operation of microservices is that individual solutions in this area hardly bring any advantages. Although a programming language or framework can be more or less suitable for a particular problem, the same applies to a much lesser extent to technologies in the field of operation.

## Standardize Only Technologies!

When these decisions are made at the level of macro architecture, they standardize only the technologies. Which configuration parameters, monitoring metrics, log messages, and deployment artifacts a microservice has, is definitely a decision at the level of the individual microservice. The independent deployment must also be retained as a core feature of the microservices. This means that it has to be possible to independently change the configuration parameters for each microservice in order to be able to make adjustments to the configuration when a new deployment takes place.

## Testing the Operation Macro Architecture

Adherence to the macro architecture rules can be checked with tests. The microservices are deployed in an environment. The tests check whether the rules for uniform deployment are adhered to. The test then verifies whether the microservice delivers metrics and log information in the defined way. Something similar is also possible in regards to configuration.

The test environment for these tests should be very minimalistic and should not contain any other microservices or a database. In this manner, the microservice is tested in an environment in which it cannot possibly work. When such a situation occurs in production, it is particularly important that the microservice provides logs and metrics to analyze potential problems. Similarly, the test also checks the resilience of the microservice.

## "You Build It, You Run It": Operation as Micro Architecture

There is a form of organization in which operational aspects have to be part of the micro architecture. If the same team is to develop and operate the microservice, it must also be able to choose the technology. This approach can be described as “You build it, you run it”. The teams are each responsible for a microservice, for its operation *and* development. You can only expect this level of responsibility from the team if you allow them to choose their own technologies.

## Operation as a Whole Is Micro or Macro Architecture

So decisions for operation can be taken either at the level of micro or macro architecture. Making operation decisions part of the macro architecture is useful if there is a separate operations team, while a “You build it, you run it” organization must make these decisions at the level of micro architecture (see [figure 2-10](#)).

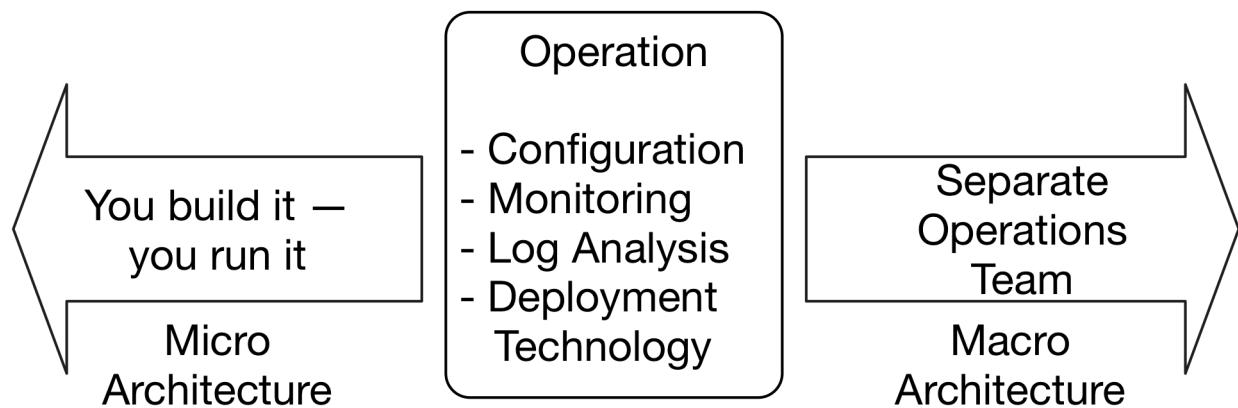


Fig. 2-10: Depending on the organization, operation is part of the micro or macro architecture

## 2.4 Give a Preference to Micro Architecture!

There are good reasons for making as many decisions as possible at the level of micro architecture:

- If only *few points* are specified in the macro architecture, this helps with focusing. Many teams have failed when trying to implement a far-reaching unification in a complex project or IT landscape. If there are few macro architecture rules, the chances increase that the rules are actually successfully implemented.
- The rules should be *minimal*. For example, a macro architecture rule can define the monitoring technology. However, it is not necessary to standardize how the metrics are measured in the application. After all, it is important only that the metrics are created. How this happens is irrelevant. So the macro architecture rule should only define a protocol for transferring metrics, but leave the selection of the library for creating and transferring metrics to the micro architecture. In this way, the teams can choose the most appropriate technologies.

- The macro architecture rules have to be *consequently enforced*. For example, when the metrics are not generated, the operations team simply cannot bring the microservice into production. So it is important to get rid of all unnecessary macro architecture elements.
- In addition, *independence* is an important goal of microservices. Too many macro architecture rules run counter to this goal, because they hinder the independence of the teams through central control.
- Complying with macro architecture should be in the *self-interest* of the teams responsible for the microservices. Violations of macro architecture usually mean that microservices cannot go into production because, for example, operations cannot support them.

## Evolution of Macro Architecture

At the beginning of a project, restrictive rules may initially apply. For example, a single programming language and a fixed stack of libraries can be defined. This reduces learning effort and operating costs. Over the duration of the project, more programming languages and libraries can then be allowed, for example, to introduce modern technologies. This leads to a more heterogeneous system. But a heterogeneous system is certainly preferable to updating all microservices at once because such updates entail a high risk.

## Best Practices and Advice

In addition to mandatory macro architecture rules, recommendations and best practices are of course advisable. However, they do not have to be enforced, but are optional for every microservice.

In the end, the goal of macro architecture is to create freedom instead of letting teams run into the open knife. Advice and references to best practices are therefore definitely good additions.

## 2.5 Organizational Aspects

There is a connection between decision and responsibility. Whoever makes a decision takes responsibility. Therefore, if the decision for a technology for metrics is made as part of the macro architecture, then the macro architecture group must take responsibility – for example, if this technology proves unsuitable in the end because it does not cope with the amount of data. If the responsibility for monitoring microservices is completely transferred to the teams, then the teams must also be allowed to select a technology.

### Uncontrolled Growth?

The freedom regarding micro architecture can lead to a huge number of used technologies. But this is not necessarily the case. If all teams have had good experiences with a particular monitoring technology, then a new microservice will most likely be monitored with the same tool. Using another tool would require a great deal of effort. Other options are only evaluated and used if the tool used

so far is not sufficient. So, even without a macro architecture rule, there is standardization if uniform decisions bring advantages for the teams. The prerequisite for this is, of course, an exchange between the teams about best practices and about the question as to which technologies work and which do not.

## Who Defines Macro Architecture?

Macro architecture restricts the freedom of the teams when it comes to implementing the microservices. This can be counteracted by having a committee define the macro architecture, which consists of one member of each team. However, it is possible that the committee may become too large to work effectively. With ten teams, the team would have ten members and effective work is then hardly possible. You can reduce the number of members by excluding teams or sending individual members as representatives for multiple teams.

Unfortunately, the team members are often too focused on their own microservices to be sufficiently interested in the overall picture of macro architecture. In a way, that is a good thing because they should focus on their microservice.

The alternative is to have an independent architecture committee decide on macro architecture, which is staffed by architects who do not belong to the teams. In such a scenario, it is important that this body's goal be to support the teams in their development of microservices and to moderate decisions rather than force them upon the teams. The most important work takes place in the teams. Therefore, the macro architecture should support the teams and not hinder them. Collaboration between the architecture committee and the teams can also be improved by the members of the architecture committee working at least partly in the teams.

With an independent architecture committee, it is important that the members of the committee are integrated and interested in the developed system. The specific domain and business requirements should never be forgotten. An important part of the work on the architecture is to understand stakeholders and make sure that their goals are supported by the architecture.

## How to Enforce?

The need for macro architecture should be understandable because it ensures that the entire system can be developed and operated. To enforce the macro architecture, the reasons for each rule should be documented. This avoids discussions because the reasoning behind the rules is then comprehensible. For example, certain macro architecture rules may be necessary to allow the software to be brought into production by the operations team, or to make sure that compliance rules are followed.

So it's not so much about getting rules enforced as it is about promoting macro architecture and conveying the ideas and reasons for macro architecture. If good reasons for changing the macro architecture exist, improving the architecture might be a better option than simply enforcing an obsolete one.

## Testing Conformance

It is possible to test the conformance to the macro architecture in some cases – by deploying a microservice and then checking its log output and metrics. This ensures that deployment, logging, and monitoring conform to the defined macro architecture.

Such a test is a black box test – that is, the test checked the behavior of the microservice from the outside. The benefit of this approach is that it does not limit the free choice of technology for implementing microservices, nor does it enforce unnecessary standards, for example, for specific frameworks. Therefore, testing for the conformance on the code level does not make a lot of sense.

## 2.6 Independent Systems Architecture Principles (ISA)

Micro and macro architecture are fundamental to the idea of microservices. However, it is hard to understand why there should be two levels of architecture.

ISA<sup>13</sup> (Independent Systems Architecture) is the term for a collection of fundamental principles for microservices. It is based on experiences with microservices gained from many different projects.

The name already suggests that these principles aim to build software out of independent systems. Macro and micro architecture are very important for this goal. A minimal macro architecture leaves a lot of freedom to the level of the micro architecture, making the systems independent. Technical decisions can be made for each system without influencing the other systems.

ISA defines the term micro and macro architecture. Also, the principles explain what the minimum requirements for macro and micro architecture are.

### Conditions

*Must* is used for principles when they absolutely have to be adhered to. *Should* describes principles which have many advantages but do not strictly have to be followed.

### Principles

1. The system must be divided into *modules* which offer *interfaces*. Accessing modules is only possible via these interfaces. Therefore, modules may not depend directly on the implementation details of another module, such as the data model in a database.
2. The system must have two clearly separated levels of architectural decisions:
  - *Macro architecture* comprises decisions which concern all modules. All further principles are part of the macro architecture.
  - *Micro architecture* comprises those decisions which can be made differently for each individual module.

---

<sup>13</sup><http://isa-principles.org>

3. Modules must be *separate processes, containers, or virtual machines* to maximize independence.
4. The choice of integration and communication options must be limited and standardized for the system. The integration might be done with synchronous or asynchronous communication, and/or on the UI level. Communication must use a limited set of protocols like RESTful HTTP or messaging. It might make sense to use just one protocol for each integration option.
5. *Metadata*, for example, for *authentication*, must be standardized. Otherwise, the user would need to log in to each microservice. This might be done using a token that is transferred with each call / request. Other examples might include a trace id to track a call and its dependent calls through the microservices.
6. Each module must have its *own independent continuous delivery pipeline*. Tests are part of the continuous delivery pipeline; therefore, the tests of the modules have to be independent, too.
7. *Operation* should be standardized. This comprises configuration, deployment, log analysis, tracing, monitoring, and alarms. There can be exceptions from the standard when a module has very specific requirements.
8. *Standards* for operations, integration, or communication should be enforced on the interface level. For example, the communication protocol and data structures could be standardized to a specific JSON payload format exchanged using HTTP, but every module should be free to use a different REST library/implementation.
9. Modules have to be *resilient*. They may not fail when other modules are unavailable or when communication problems occur. They have to be able to shut down without losing data or state. It has to be possible to move them to other environments (server, networks, configurations and so on) without the module failing.

## Evaluation

The ISA principles are not just a great guideline for building microservices. They also explain why macro and micro architecture are so important. Also, ISA explains the structure of the book.

- The *first* ISA principle just states that a system must be built from modules. This is common knowledge.
- The *second principle* defines two levels of architecture: macro and micro architecture. These are the terms defined in this chapter, too.
- In a deployment monolith, most of the decisions will be on the macro architecture level. For example, a deployment will be written in one programming language, so the programming language has to be a decision on the macro architecture level. The same is true for frameworks and most of the other technologies. To make more decision on the micro architecture level, each module must be implemented in a separate container as *principle three* states. So ISA says that the reason why microservices run in containers is the additional technological freedom that cannot be achieved in a deployment monolith. Therefore, microservices add more independence and decoupling to the architecture. An approach where each microservice is a WAR and all run together in one Java application server does not fit this principle. Actually,

the compromise concerning the free choice of technology and the robustness is so high that this approach usually does not make a lot of sense. Because decoupling is so important, ISA and microservices actually provide fundamental improvements to modularization.

- Although the goal of ISA is to create a minimal macro architecture, some decisions still need to be made on the macro level. This is what the rest of the principles explain. As a start, *principle four* states that integration and communication must be standardized; [part II](#) discusses technology stacks for integration and communication. The decision to use a specific technology for integration and communication influences all modules and must therefore be done on the macro architecture level. It is therefore a very important decision in microservices system. Without a common integration approach and communication technology, it is hard to consider the system a system and not just a few services that cannot even really communicate with each other.
- *Principle five* states that metadata for tracing and authentication must be standardized. Such metadata must be transferred between the microservices and must therefore also be a part of the macro architecture. For that reason, [chapter 22](#) discusses tracing in more detail. However, this book does not discuss security aspects of microservices, including metadata for authentication.
- *Principle six* (independent deployment pipelines) extends the idea of independent deployment as the definition of microservices from [section 1.1](#). [Chapter 16](#) explains platforms for the deployment of microservices. [Chapter 17](#) discusses Kubernetes as a concrete example, and [chapter 18](#) the PaaS Cloud Foundry.
- *Principle seven* says that operations of microservices should be standardized. It is not in all cases necessary to standardize operations. With a separate operations department, standardization is the only way to handle a large number of microservices. However, with a “you build it – you run it” organization, standards are not necessary as each team operates their microservices. Actually, a standardized operations approach might not fit all microservices. In that case, the teams need to come up with their own operations technologies. A standard makes little sense then.
- *Principle eight* states that standards should only be defined on the interface level. The technologies discussed throughout [part II](#) can be used in this way. They provide interfaces and client libraries for all commonly used programming languages.
- *Principle nine* talks about resilience. This books focuses on asynchronous communication, which makes resilience easier. If a microservice fails, a message will be transferred later but the failed microservice will not cause another microservice to fail. Also, the chapters about synchronous communication show how systems can be resilient even if using synchronous communication.

Therefore, the ISA principles represent a good summary of the ideas introduced in this chapter – that is, a division between micro and macro architecture as the main benefit of microservices. ISA also explains why the rest of the book focuses on integration and communication technologies and technologies for operations. These are the fields that a macro architecture has to cover. Therefore, these decisions are very important and also hard to change because they influence all microservices in the system.

## 2.7 Variations

In the domain macro architecture, strategic design and domain-driven design are ultimately unrivaled as approaches. However, the bounded contexts depend on the specific project. Identifying the right bounded contexts is a central challenge when designing the architecture of a microservices system.

The technical micro and macro architecture also has to be devised for each project. This depends on many factors:

- Organizational aspects such as a *DevOps organization* or having a separate operations team has an influence.
- In addition, *strategic technology decisions* can play a role.
- Even the *hiring policy* can be a factor. Eventually, there have to be experts available for the technologies who can work in the teams.

### More Complex Rules

In reality, the rules of micro and macro architecture are often more complex. For example, a whitelist can exist for the programming language. In addition, there can be a procedure for adding more programming languages to the whitelist, for example, via a committee for making such decisions. And finally, there can be a general limitation to programming languages that run on the JVM (Java Virtual Machine) because there is a lot of experience with it in production.

Such a rule has elements of a macro architecture decision. There is a whitelist and a restriction to JVM languages. At the same time, it also has micro architecture elements. After all, a team can select one of the programming languages from the whitelist and even extend the whitelist.

Therefore, rules are often in place for every point in practice that allow the teams and microservices a certain amount of leeway. These rules are not purely micro or macro architecture rules, but lie somewhere in between.

### Experiments

The approach for defining micro and macro architecture can look like this:

- Consider a project you are familiar with. Look at its domain model.
  - Would a division into multiple domain models and bounded contexts make the system easier?
  - In how many bounded contexts would you split the system? Typical projects consist of about ten bounded contexts. However, the exact number will vary for each individual project.

- Determine the use cases which the system implements. Group use cases and analyze whether these use cases can be addressed by a domain model. By doing so, these use cases form a bounded context in which the domain model is valid.
- Is a further division for technical reasons sensible? The technical reasons can comprise independent scalability or security (see also “Two Levels of Microservices” in [section 1.2](#)).
- In this chapter, areas such as programming language and DevOps are addressed which can belong either to micro or macro architecture. Define for your project whether the individual decision should be part of micro or macro architecture.
- Work out at least one of the decisions in more detail. For example, there could be whitelist of programming languages, or only one programming language might be allowed that can be used by all microservices – or even a procedure for extending the whitelist.

## 2.8 Conclusion

Microservices and Self-contained Systems allow architecture decisions to be made individually for each microservice. If decisions can actually be different for each microservice, then they are part of the micro architecture. The macro architecture, on the other hand, contains those parts of the architecture that apply uniformly to all microservices. The division into these two levels gives the individual microservices freedom, while at the same time ensuring the integrity of the overall system.

Decisions at the micro architecture level are better suited to the self-organization of teams and use the technical freedom offered by microservices. Even if decisions are part of the micro architecture, a standardization can still result, because using technologies with which other teams already had positive experiences decreases the risk and allows for the use of synergies.

In any case, decisions must be made explicitly. The teams must consciously deal with the macro architecture and the freedoms in micro architecture. Micro and macro architecture form a trade-off, which can be different in each project.

# 3 Migration

Migration from a deployment monolith to a microservices architecture is the common case for introducing microservices. Most projects start with a deployment monolith that the team wants to split into microservices later because the deployment monolith has too many disadvantages.

Of course, it is also possible to implement a new system directly with microservices from scratch.

This chapter provides an overview of the challenges involved in migrating to a microservices system.

- The chapter discusses possible *reasons* for a migration. In this way, readers can assess whether a migration makes sense in their context. The approach to migration depends on the objectives of the migration. Therefore, knowing possible reasons is helpful for choosing a migration strategy.
- The chapter then shows a *typical strategy* for migration, but also alternatives. In this way, the reader can choose a migration approach suitable for his or her scenario.

## 3.1 Reasons for Migrating

When migrating to microservices, it is important to know the objectives for taking this step. Depending on the several possible reasons that led to the decision to migrate to microservices, the procedure for implementing them may vary.

### Microservices Offer a Fresh Start

Especially when replacing legacy systems, microservices have several advantages. The code of the legacy system no longer needs to be used in the new microservices because the microservices are implemented separately from the legacy system. So the microservices offer an unencumbered restart. The code of a legacy system is often no longer maintainable, and the technologies are frequently outdated. Therefore, reusing the old code would hinder the development of a clean new system. Microservices thus solve the most important challenges when dealing with legacy systems, because otherwise a restart is difficult since newer systems have to be integrated with old code.

Migration to microservices has the potential to finally solve the problem with the legacy system. After migration to microservices, further migrations can be limited to one or a few microservices. A migration of the entire system will probably not be necessary again. A typical reason for a system migration is an outdated technology basis. In a microservices system, such a migration can take place step by step – that is, microservice by microservice. Another migration reason is an unmaintainable system. In this case also, each microservice can be replaced individually.

## The Reasons Are Already Known

The reasons for a migration are in the end identical to the reasons for using microservices. These have already been discussed in detail in [section 1.2](#) and may include increased security, robustness, and independent scaling of individual microservices.

### A Typical Reason: Speed of Development

A typical reason for introducing microservices is the lack of speed in developing with a deployment monolith. When many developers are working on a deployment monolith, they need to closely coordinate their work. This costs time and therefore slows down development. But even with a small team, a deployment monolith can be problematic because the deployment is quite huge. The size makes continuous delivery difficult to implement, and each release requires a lot of testing.

## 3.2 A Typical Migration Strategy

Often there is a concept for the final target architecture that the migration should achieve, but no concrete plan for the first steps to be taken or for the first microservices to be implemented. In particular, the small steps into which development can be broken down are a major advantage of microservices. A simple microservice is written quickly. Because of its small size, it is also easy to deploy. And if the microservice should not prove itself, not much effort has gone into the microservice and it can easily be removed again. In the further course of the project, the new architecture can be implemented step by step and microservice by microservice. In this way, large risks can be avoided.

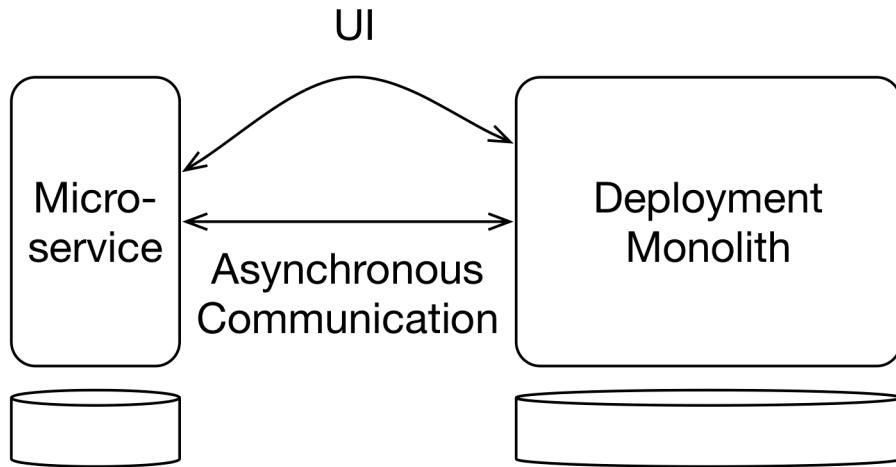
Because the migration process depends on the goals and on the structure of the legacy system, there is no universal approach. Therefore, the strategy presented here must not simply be used as is, but must be adapted to the respective situation.

### A Typical Scenario

A typical scenario for a migration to microservices is:

- The aim of the migration is to increase the development speed. Microservices need fewer tests for a release and provide easier continuous delivery because they are smaller. Also, development of individual microservices is quite independent, so less time is spent with coordination. All of this can make the development faster.
- The migration to microservices must provide an advantage in development as quickly as possible. It simply makes no sense to invest in an architectural improvement that only leads to an improvement much later.

The migration strategy proposed here is based on extracting individual microservices (see [figure 3.1](#)) in order to achieve an improvement of the situation as quickly as possible.



**Fig. 3-1: Approach for a Migration: Asynchronous Communication and UI Integration Between Legacy System and Microservice. The Microservice has its own Data Storage.**

## Give a Preference to Asynchronous Communication

Integration with the legacy system should take place via asynchronous communication. This decouples the domain logic of the microservice from the legacy system. The legacy system sends events. To do this, the legacy system must be adapted to create events. Since the legacy system is usually poorly maintained, this can be a challenge.

Microservices then can decide how to react to these events. One of the possible benefits is availability: A failure of the legacy system does not lead to the failure of the microservice, and a failure of the microservice does not lead to the failure of the legacy system.

## Give a Preference to UI Integration

Further integration is conceivable at the UI level. If the legacy system and the microservice are integrated with each other via links, then only URLs are known. What hides behind the URLs can be decided by the linked system and can change without much impact on other systems.

Via links, additional resources can be available. This is the basis of [HATEOAS<sup>14</sup>](#) (Hypermedia as the Engine of Application State). The client can interact with the system through the links, and does not need to know about possible interaction possibilities, but can follow the links. For example, a link to cancel an order would be sent along with the order. New interaction possibilities can be easily supplemented by new links.

The UI integration also offers an easy way to operate the microservice and the legacy system in parallel. Individual requests can be redirected to the microservices, while the remaining requests

<sup>14</sup><https://en.wikipedia.org/wiki/HATEOAS>

are still processed by the legacy system. Often, there is a web server anyway which processes every request and carries out TLS/SSL termination, for example. A parallel operation of microservices and legacy system is then quite simple. The web server only has to forward each request either to a microservice or to the legacy system.

UI integration is particularly easy if the legacy system is a web application. But it is also possible, for example, to integrate web views in a mobile application to thereby integrate parts of the UI as web pages. In such cases, UI integration should really be considered as an option because of its many advantages.

## Avoid Synchronous Communication

Synchronous communication should be used sparingly. It leads to a close dependence in regards to availability. If the called system fails, the calling system must be able to deal with this. The degree of coupling in the domain logic is also quite high. A synchronous call usually describes exactly what needs to be done. Synchronous communication may be necessary if you want the last changes to be visible in the other systems as soon as possible. In a synchronous call, the state at the time of the call is always used, whereas asynchronous communication and replication can lead to a delay until the current state is known everywhere.

## Reuse Old Interfaces?

If there is already an interface, it may be useful to use this interface to save the effort of introducing a new interface. However, the interface may not be well adapted to the needs of the microservice. In addition, it cannot be changed easily because there are already other systems that also use this interface and are influenced by changes.

The technology the interfaces uses is not too important for the decision of whether it should be used by a microservice. Microservices can use almost any type of interface. For a migration, it might be a lot easier to use an existing interface even with an awkward technology than to create a new one.

More important are the dependencies the microservices establish: As discussed in [section 2.1](#), the selected pattern for the integration influences the coordination effort and the degree of independence. Just reusing an existing interface might compromise a goal like independent development.

## Integrating Authentication

For a system that consists of a legacy system and microservices, the user should have to log in only once. Legacy systems and microservices do not necessarily have the same authentication technologies, but the systems must be integrated in such a way that a single sign-on is possible, and the user does not have to log on to the legacy system and microservices separately. Authentication may also need to provide roles and permissions for authorization in the microservices. Adjustments may also be necessary here.

## Replicating Data

Even in a migration scenario, each microservice should have its own database or at least its own database schema. The goal of the migration is to achieve independent development and simple continuous delivery of the microservices. This is not possible if the microservices and the legacy system use the same database. A change to the database schema then might have hard-to-predict effects. As a result, the microservice is hardly changeable and difficult to put into production.

Together with asynchronous communication, a separate database means data replication. This is the only way for the microservices to implement their own data model. Changes to the data can be communicated via events.

Replication for a specific part of the data should take place in one direction only. Usually replication can be done using business events – that is, events that have a meaning for a business expert. One system (a microservice or the legacy system) should trigger the events, and the other system reacts to the events. So, for example, one system could generate a system like “customer registered” and the other system could store the customer data relevant to them. However, there should be only one source of each type of event. Otherwise, it can be very complicated to bring together the changes of the different systems into a consistent state.

## Black Box Migration

Often the code of the deployment monolith is hard to understand and modify. This might even be a reason for a migration. Therefore, it does not make a lot of sense to reverse-engineer the existing code or even restructure it. That way, migrating an existing system requires little or a minimum of knowledge of the system.

## Choosing the First Microservice for the Migration

A legacy system comprises numerous domain functionalities. For deciding about the migration strategy, it can be useful to analyze the domain logic of the legacy system. The result should be a complete and ideal split of the legacy system into bounded contexts (see [section 2.1](#)). It is not implemented in the legacy system but can be the goal for the migration to microservices. This analysis can be done without understanding the code. It is about what the system does – that is, it is enough to treat it as a black box.

Extracting one of these bounded contexts as a microservice has this advantage: A bounded context is largely independent of other bounded contexts from a domain perspective since it has its own domain model.

However, the question is which bounded context to extract first from the legacy system. There are different approaches.

- To keep the *risk* as low as possible, an unimportant bounded context with little load can be the right choice. This makes it possible to gain experience with the challenges of microservices, for example, concerning operation, without taking too great a risk.

- Microservices are meant to simplify development. In order to exploit the advantages of this approach as quickly as possible, you can migrate a bounded context to a microservice that *will have to be changed a lot* in the foreseeable future. The changes should become easier to introduce after migrating to a microservice, so that the cost of migration quickly pays for itself.

## Extreme Migration Strategy: All Changes in Microservices

An extreme migration strategy is to no longer allow any changes to the legacy system, but only allow changes to microservices. When a change would have to be made to the legacy system, a new microservice must be created first. The change is then implemented in this microservice instead. This automatically results in a migration to microservices as more and more logic is implemented in microservices over time. It is very easy to follow this rule.

One problem with this approach is that the microservices are created in random places, namely where the system is currently being changed. This can result in microservices that implement only parts of a bounded context, whereas other parts of the bounded context are still implemented in the legacy system. Therefore, microservices and legacy systems have numerous dependencies, making independent development difficult.

## Further Procedure: Step-by-Step Migration

The legacy system can be gradually replaced by microservices. In the course of the migration, the focus should be on converting parts of the system into microservices that are currently undergoing major changes so that the migration to microservices is worthwhile. This is called the *strangler pattern*<sup>15</sup>. The microservices increasingly strangle the legacy system until nothing is left of the legacy system anymore (see figure 3-2).

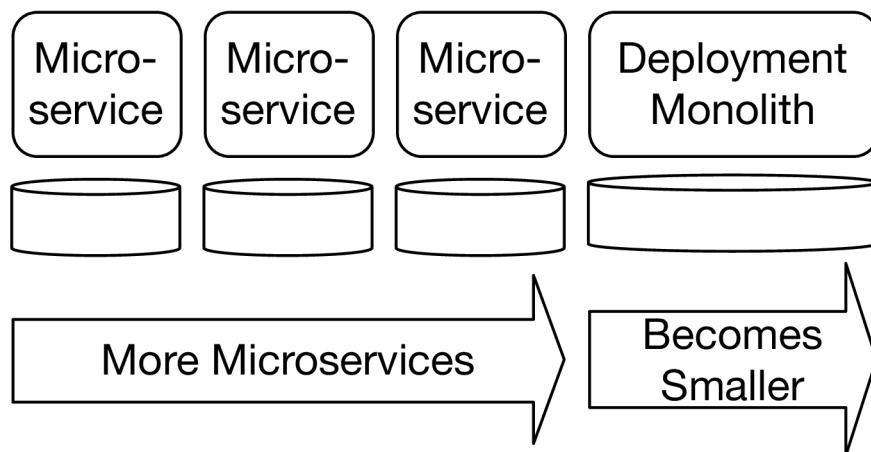


Fig. 3-2: Further Migration: An Increasing Number of Microservices Take over Functionalities from the Legacy System

<sup>15</sup><https://www.martinfowler.com/bliki/StranglerApplication.html>

The full migration to microservices can take a very long time. However, this is no problem: Only those parts of the system are migrated for which migration brings an advantage. For example, a part of the system must be changed. It is migrated to a microservice. That makes the changes much easier. Parts that are never or very seldom changed will be migrated later or even never. So the time the full migration takes is a result of the flexibility to migrate only what is actually needed. The migration stops if there is nothing worth migrating any more. In the end, it makes no sense to invest into the optimization of system parts which are rarely or never changed.

It might even happen that the legacy system could in principle be completely migrated, but is still retained. When hardly any changes to the legacy system are necessary anymore, because all parts that require changes have already been migrated to microservices, retaining the legacy system can be the best solution.

## 3.3 Alternative Strategies

There are many more strategies. There is a [presentation<sup>16</sup>](#) that gives a good overview of the different approaches to the migration to microservices. However, this section also shows some of the more common approaches.

### Goal Reliability

As mentioned previously, there can be very divergent approaches for the migration to microservices. The strategy depends mainly on the objectives to be achieved. When the main objective for switching to microservices is an increase in robustness, at first reliability can be improved at the interfaces to external systems, or databases with libraries like Hystrix (see [section 13.5](#)).

Then the system can be split step by step into individual microservices that run independently of each other so that a failure of one microservice no longer affects the other microservices. There is an interesting talk about this approach to which [slides<sup>17</sup>](#) are available.

### Migration Based on Layers

Another alternative is a migration based on layers. For example, the UI can be migrated first. This can make sense when changes to the UI are imminent, and therefore the migration can be combined with necessary changes. Of course, this migration strategy is in contrast to the idea of combining UI, logic, and data in one Self-contained System (see [chapter 3](#)). However, it can still be a first step towards this goal. In that case, the remaining layers would have to be migrated into the same microservice afterwards. Alternatively, one stays with a division of microservices in layers, although it is not optimal. An ideal architecture, however, into which it is impossible to migrate, is in the end a lot less helpful than a less optimal architecture which can actually be implemented.

<sup>16</sup><https://speakerdeck.com/ewolff/monolith-to-microservices-a-comparison-of-strategies>

<sup>17</sup><https://www.innoq.com/de/talks/2015/11/javaday-kyiv-modernization-legacy-systems-microservices-hystrix/>

## Copy/Change

Another possibility is copy/change. Here, the code of the legacy system is copied. In one copy, one part of the system is developed further, while the other part is removed. In a second copy, it is the other way around. In this manner, the legacy system is converted into two microservices. This approach has the advantage that the old code is still used, and therefore the functionalities of the microservices very likely correspond accurately to the functionalities of the legacy system.

However, at the same time it is a great disadvantage to continue using the old code. In most cases the code of a legacy system is hard to maintain, and so it is problematic to keep using this code. In addition, the database schema remains unaltered. The shared use of the database schema by the legacy system and the microservices results in a tight coupling between the two, which really should be avoided to be able to profit from the advantages microservices offer. That is why a black box migration might be better.

Moreover, the structure of the legacy system and the technology stack largely remain the same.

Thus, the project has a lot of technical debt from the very beginning and does not represent a new start. This approach does not take advantage of the benefits of microservices such as freedom of technology. Therefore, it should only be used in exceptional cases.

## 3.4 Build, Operation, and Organization

Code migration alone is not enough to turn a legacy system into a microservices system.

- The microservices must also be *built*. A suitable tool must be selected for this purpose. In addition, the continuous integration server has to cope with the multitude of microservices.
- Similarly, technologies and approaches must be introduced to enable the *deployment* and *operation* of microservices.
- Finally, a suitable *test strategy* must be established. This also requires the automated setup of test environments, and the assurance that the tests are independent. For example, stubs that simulate microservices or the legacy system are useful for this purpose, as are [consumer-driven contract tests<sup>18</sup>](#). They safeguard the requirements for the interfaces of microservices or legacy systems with the help of tests. However, legacy systems are often very complicated, so these techniques are difficult to implement.

Therefore, dealing with the first microservice can require extra effort because the infrastructure for build and deployment needs to be set up. It is conceivable to build the infrastructure later, but it is recommended to start building the infrastructure as early as possible in order to reduce the risk of migration. One or a few microservices can still be operated with an inadequate solution for build and deployment. However, once the number of microservices increases, without an appropriate infrastructure the necessary effort will become so high that it can lead to the failure of the project.

---

<sup>18</sup><https://martinfowler.com/articles/consumerDrivenContracts.html>

## Co-existence between Microservices and Legacy Systems

During a migration, the legacy system must be deployed and further developed in addition to the microservices. It is unrealistic to deploy the legacy system as often as the microservices because the effort of deploying the legacy system is usually far too high. Therefore, changes affecting both the legacy system and the microservices are difficult to implement. They require at least one deployment of the microservices and one deployment of the legacy system. Solutions can be found at the architectural level. If a new feature is implemented only in a microservice, then a deployment of only this microservice is necessary. This speaks for a division of the microservices according to bounded context. Another option would be to integrate the monolith with patterns such as *open host service* or *published language* (see [section 2.1](#)) to provide a generic interface that rarely needs to be changed.

## Integration Test of Microservices and Legacy Systems

There must also be integration tests that test microservices with the version of the legacy system currently in production and with the one currently being developed. This ensures that the microservices continue to work when the legacy system is deployed. The legacy system can support two different versions of the interfaces, so that microservices can switch to a new version of an interface when it is provided. However, no microservice is forced to use a new interface that has not yet been tested together with the microservice. In this way, the version of the microservice that uses a new interface of the legacy system can be deployed at any time.

## Coordinated Deployment between Legacy Systems and Microservices

A coordinated deployment of microservices together with the legacy system would be an alternative. When a change is made, the new version of the microservices and the legacy system are rolled out at the same time. This increases the risk because more changes occur at the same time, and it is harder to roll back the deployment. It is also difficult to implement this approach without downtime. With a complex microservices environment, this option is hardly possible anymore because too many microservices would have to be deployed at once. Therefore, the deployment of microservices and legacy systems should be decoupled from the outset.

## Organizational Aspects

An essential advantage of microservices is the possibility to scale the development process (see [section 1.2](#)).

If the goal of a migration to microservices is to have independent teams, the migration of the architecture must be accompanied by a reorganization. [Section 2.5](#) discusses the essential aspects of the target organization.

The organizational change must be coordinated with the technical migration. For example, a microservice can be detached from the legacy system and then developed autonomously by a team. At the same time, the other organizational structures can be set up, – for example, the ones required for defining the macro architecture.

## Recommendation: Do Not Implement All Aspects at Once

Microservices require changes in architecture and organization as well as the introduction of new technologies. Implementing all these changes at once is risky and complicated. Unfortunately, many of the changes are connected. Without new technologies, the architecture is difficult to implement. Without the architecture, organizational changes are difficult to introduce. However, making all these changes at once should still be avoided. For each change, the question as to whether the change is actually necessary should be asked, in order to implement it at a later point in time, if possible.

## 3.5 Variations

The ideas for migration can easily combine with many other approaches.

- The ideas concerning typical migration strategies from [section 3.2](#) fit very well to the concept of *Self-contained Systems* ([chapter 3](#)). The migration can therefore simply separate a part of the legacy system into an SCS.
- Rules for authentication or communication between microservices as well as between microservices and the legacy system can be the starting point of a *macro architecture* (see [chapter 2](#)). A domain macro architecture is very useful, which can also include the legacy system in addition to microservices.
- *Frontend integration* (see [chapter 7](#)) can make sense for the integration between legacy system and microservices.
- *Asynchronous microservices* ([chapter 10](#)) fit very well to migration because they allow for a loose coupling. Especially for a migration it can be sensible to continue to use an existing messaging technology for asynchronous communication to minimize the effort.
- *Synchronous microservices* ([chapter 13](#)) should be used cautiously because this creates a tight coupling and resilience becomes a difficult topic.
- *Kubernetes* ([chapter 17](#)), *PaaS* ([chapter 18](#)), or *Docker* ([chapter 5](#)) are certainly also interesting in a migration scenario. However, they represent a *new environment* that needs to be operated. It may therefore make sense to use a classical deployment and operation environment at least at the beginning to reduce the initial migration effort. In the long term, however, such environments have many advantages. In addition, of course, the old system can be operated in such an environment.

## Experiments

The migration strategy must match the respective scenario. The following questions are important in order to design your own strategy.

- What are the goals of the migration to microservices?
  - Which are especially important?

- What impact does this have on the migration strategy?

In principle, migration should take place gradually. The selection of the parts to be migrated to microservices can be made according to technical or domain criteria. However, domain criteria are better suited, at least in the long term.

The following approach is suitable for a migration based on domain criteria:

- Split the system into bounded contexts.
- Which of the bounded contexts will you migrate first? Why? Reasons can be the simple migration of the bounded context or many planned changes in the bounded context. Consider different scenarios.

## 3.6 Conclusion

Migration to microservices is the typical approach for the introduction of microservices. Implementing a completely new system with microservices is rather the exception, but of course it is also possible. One of the most important advantages of microservices is that they do work well for scenarios other than greenfield projects.

Choosing the right migration strategy is a complex task. It depends on the legacy system and migration goals. This chapter shows a starting point from which each project must develop its own strategy.

Because of the benefits of migration, microservices should be considered in any project meant to modernize a legacy system. Microservices enable step-by-step modernization, in which completely different technologies can be used. This is very helpful.

The migration strategy can have a significant impact on the architecture and technology selection. A split into microservices similar to the split of the modules in the legacy system can greatly simplify migration. Such a compromise has far-reaching consequences and can lead to a worse target architecture, but it can still make sense. Finally, it must be possible to actually implement the architecture, and for this a simple approach for the migration into the architecture is essential.

# 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.

# 4 Docker

This chapter provides an introduction to Docker and covers the following:

- After studying the chapter, the reader is able to run the examples provided in the following chapters in a Docker environment.
- Docker and microservices are nearly synonymous. This chapter explains why Docker fits so well to microservices.
- Docker facilitates software installation. Important for this is the `Dockerfile`, which describes the installation of the software in a simple way.
- Docker Machine and Docker Compose support Docker on server systems and complex software environments with Docker.
- The chapter lays the foundation for an understanding of technologies such as Kubernetes ([chapter 17](#)) and Cloud Foundry ([chapter 18](#)), which are based on Docker.

## Licences and Projects

Docker is under the Apache 2.0 licence. It is developed by the company [Docker, Inc<sup>19</sup>](#), among others. Some core components, such as, for example, [Moby<sup>20</sup>](#), are under an Open Source licence and thus allow other developers to also implement systems similar to Docker. Docker is based on Linux containers, which isolate processes in Linux systems from each other. The [Open Container Initiative<sup>21</sup>](#) ensures via standardization the compatibility of the different container systems.

## 4.1 Docker for Microservices: Reasons

[Chapter 1](#) defined microservices as separately deployable units. The separate deployment not only results in a decoupling at the architectural level, but also in regards to technology choice, robustness, security, and scalability.

### OS Processes for Microservices?

If microservices are supposed to have all these characteristics, the question arises as to how they can be implemented. Microservices must be scalable independently of each other. In the event of a crash, a microservice must not be allowed to make other microservices unavailable, too, and thus endanger the robustness of the whole system. Therefore, microservices must at least be separate processes.

---

<sup>19</sup><https://www.docker.com/>

<sup>20</sup><https://github.com/moby/moby>

<sup>21</sup><https://www.opencontainers.org/>

*Scalability* can be guaranteed by multiple instances of a process. When an application is started, the operating system generates a process and allocates resources such as CPU or memory to it. Therefore, more processes can use more resources. But processes are limited concerning scaling. If the processes all run on one server, then only a limited amount of hardware resources are available. Instead, the microservices should run in a cluster. Kubernetes ([chapter 17](#)) and Cloud Foundry ([chapter 18](#)) support running microservices in a cluster.

With processes, *robustness* is guaranteed to a certain extent because the crash of one process does not affect the other processes. However, a server failure still causes a large number of processes, and thus microservices, to fail. But there are also other problems. All processes share one operating system. It must provide the libraries and tools for all microservices. Each microservice must be compatible with the operating system version. It is difficult to configure the operating system to support all microservices. In addition, the processes must coordinate in such a way that each process has its own network port. If you have a large number of processes, it becomes increasingly harder to find unused ports. Also, it's hard to figure out which ports are used by which process.

## **Virtual Machines: Too Heavy-Weight for Microservices**

Instead of a process, each microservice can run in its own virtual machine. Virtual machines are simulated computers that all run on the same physical hardware. For the operating system and application, virtual machines look exactly like a physical server. Through virtualization, the microservice has its own operating system installation. Thus, the configuration of the operating system can be adapted to the specific microservice, and there is also complete freedom in choosing the network port.

However, a virtual machine has a substantial overhead:

- The virtual machine must give the operating system the illusion of running directly on the hardware. This leads to an overhead. Therefore, *performance* is poorer than with physical hardware.
- Each microservice has its own instance of the operating system. This consumes a lot of memory in the *RAM*.
- Finally, the virtual machine has virtual disks with a complete operating system installation. This means that the microservice occupies a lot of *hard disk space*.

So virtual machines have an overhead, making their operation expensive. In addition, operations has to manage a large number of virtual servers. This is complicated and time-consuming.

The ideal solution would be a light weight alternative to virtualization, which possesses the isolation of virtual machines, but consumes as little resources as processes do and is similarly easy to operate.

## **4.2 Docker Basics**

Docker represents a light weight alternative to virtualization. Although Docker does not provide as much isolation as a virtualization, it is practically as light weight as a process.

- Instead of having a complete virtual machine of their own, Docker containers *share the kernel* of the operating system on the Docker host. The Docker host is the system on which the Docker containers run. The processes from the containers therefore appear in the process table of the operating system on which the Docker containers are running.
- The Docker containers have their *own network interface*. In this way, the same port can be used in each Docker container, and each container can use any number of ports. The network interface is in a subnet where all Docker containers are accessible. The subnet is not accessible from the outside. This is at least the standard configuration of Docker. The Docker network configuration offers many other alternatives. To still allow external access to a Docker container from the outside, ports of a Docker container can be mapped to ports on the Docker host. When mapping the ports of the Docker containers to the ports of the Docker host, be careful because each port of the Docker host can only be mapped to one port of a Docker container.

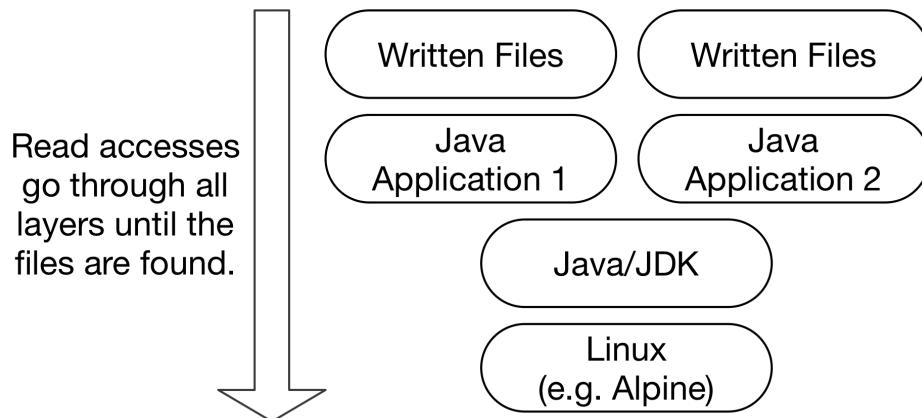


Fig. 4-1: File System Layers in Docker

- Finally, the file system is optimized. There are *layers in the file system*. When a microservice reads a file, it goes through the layers from top to bottom until it finds the data. The containers can share layers. [Figure 4-1](#) shows this more precisely. The file system layer at the bottom represents a simple Linux installation with the Alpine Linux distribution. Another layer is the Java installation. Both applications share these two layers, which are stored only once on the hard disk, although both microservices use them. Only the applications are stored in file system layers that are exclusively available to a single container. The lower layers cannot be changed. The microservices can only write to the top layer. The reuse of the layers reduces the storage requirements of the Docker containers.

It is easily possible to start hundreds of containers on a laptop. This is not surprising: after all, it is also possible to start hundreds of processes on a laptop. Docker has no significant overhead compared to a process. Compared to virtual machines, however, the performance benefits are outstanding.

## One Process per Container

Ultimately, Docker containers are highly isolated processes due to their own network interface and file system. Therefore, only one process should run in a Docker container. Running more than one process in a Docker container contradicts the idea of separating processes from each other by means of Docker containers. Because only one process is supposed to run in a Docker container, there should be no background services or daemons in Docker containers.

## Docker Image and Docker Registry

File systems of Docker containers can be exported as Docker images. These images can be passed on as files or stored in a Docker registry. Many repositories such as [Nexus<sup>22</sup>](#) and [Artifactory<sup>23</sup>](#) can store and provide Docker images just like compiled software and libraries. This makes it easy to exchange Docker images with a Docker registry for installation in production. The transfer of images from and to the registry is optimized. Only the updated layers are transferred.

## Supported Operating Systems

Docker was originally a Linux technology. For operating systems such as macOS and Windows, Docker installations are available that allow Linux Docker containers to be started. For this purpose, a virtual machine with a Linux installation is running in the background. This is transparent for the user. It seems as if the Docker containers are running directly on a computer.

In addition Windows, since Windows Server 2016, provides Windows Docker containers. Linux applications run in a Linux Docker container, Windows applications in a Windows Docker container.

## Operating Systems for Docker

Docker changes the requirements for operating systems.

- Only one process is supposed to run in one *Docker container*. This means that only as much of the operating system is required as is needed to run that process. For a Java application, this is the Java Virtual Machine (JVM), which requires some Linux libraries that are loaded at runtime. A shell is not necessary, for example. Therefore, distributions like [Alpine Linux<sup>24</sup>](#), which are just a few megabytes in size, only contain the most important tools, making them an ideal basis for Docker containers. The programming language Go can create statically linked programs. In that case, nothing else has to be available in the Docker container besides the program itself, and no Linux distribution is required at all.
- The *Docker host* on which the Docker containers run only has to run Docker containers. Many Linux tools are therefore superfluous. [CoreOS<sup>25</sup>](#) is a Linux distribution that can run little more

---

<sup>22</sup><https://www.sonatype.com/nexus-repository-sonatype>

<sup>23</sup><https://www.jfrog.com/open-source/#artifactory>

<sup>24</sup><https://alpinelinux.org/>

<sup>25</sup><https://coreos.com/>

than Docker containers and, for example, considerably simplifies operating system updates of an entire cluster. CoreOS can also serve as a basis for Kubernetes (see also [chapter 17](#)). Another example is [boot2docker<sup>26</sup>](#), which Docker Machine installs (see [section 4.4](#)) on servers to run Docker containers on these servers.

## Overview

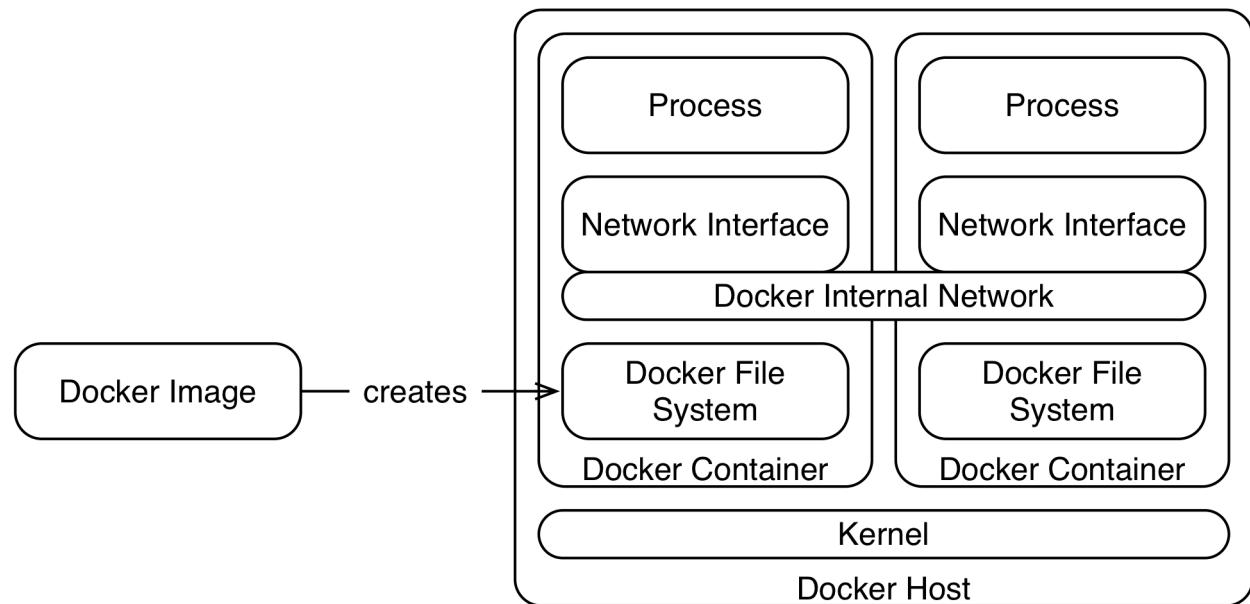


Fig. 4-2: Overview of Docker

[Figure 4-2](#) shows an overview of the Docker concepts.

- The *Docker host* is the machine on which the Docker containers run. It can be a virtual machine or a physical machine.
- *Docker containers* run on the Docker host.
- The containers typically contain a *process*.
- Each container has its own *network interface* with its own IP address. This network interface is only accessible from the Docker internal network. However, there are also ways to allow access from outside this network.
- In addition, each container has its own *file system* (see [figure 4-1](#)).
- When a container is started, the *Docker image* creates the first version of the Docker file system. When the container has been started, the image is extended by another layer into which the container can write its own data.
- All Docker containers share the *kernel* of the Docker host.

<sup>26</sup><http://boot2docker.io/>

## Does It Always Have to Be Docker?

Docker is a very popular option for deploying microservices. However, there are alternatives. Two alternatives were already mentioned in [section 4.1](#): virtual machines or processes.

### Microservices as WARs in Java Application Servers

However, you can also deploy several microservices as WAR files in a Java application server or Java web server. WARs contain a Java web application, and can be deployed separately. However, the deployment of a WAR may require the server to be restarted. Because microservices should only be deployable separately (see [chapter 1](#)), microservices can be implemented as WARs. Then, however, compromises are made with regards to robustness. A memory leak in a microservice can lead to failure of all microservices. The microservice would allocate more and more memory until an *OutOfMemoryError* is thrown and the entire Java application server crashes.

Separate scalability is also difficult to implement because each server contains all microservices and, therefore, only all microservices can be scaled together. This makes the scaling more complex than in cases where each server contains only one microservice, because unneeded microservices are also scaled. Of course, it would be possible to run each WAR on a separate Java web server and have multiple instances of each of these Java web servers. But then the WARs would no longer run together on one Java web server.

And finally, all microservices run in one operating system process, which is a compromise in terms of security. When a hacker can take over the process, he or she has access to the entire functionality and data of all microservices.

In return, these approaches consume less resources. An application server with several web applications requires only one JVM (Java Virtual Machine), only one process, and only one operating system instance. Furthermore, there is no need to introduce a new infrastructure if application servers are already in use, which can reduce the workload for operations.

## 4.3 Docker Installation and Docker Commands

Because the examples in this book all require Docker, a Docker installation is an essential prerequisite for actually starting the demos and working through them. [Appendix A](#) describes the installation of the necessary software for the examples. This also includes the installation of Docker and Docker Compose.

Docker is controlled with the command line tool `docker`. It offers a lot of commands to work with the images and containers. The [reference<sup>27</sup>](#) contains a complete list of all commands with all options. [Appendix C](#) shows an overview of the Docker commands.

---

<sup>27</sup><https://docs.docker.com/engine/reference/commandline/cli/>

## 4.4 Installing Docker Hosts with Docker Machine

Docker Machine is a tool that can install Docker hosts. From a technical point of view, the installation is quite easy to do. Docker Machine loads an ISO CD image with boot2docker from the Internet. boot2docker is a Linux distribution and provides an easy way to run Docker containers. After that, Docker Machine starts a virtual machine with this boot2docker image.

Particularly convenient with Docker machine is the fact that using the Docker containers on external Docker hosts is just as easy as using local Docker containers. The Docker command line tools only need to be configured to use the external Docker host. Afterwards, the use of the Docker host is transparent.

### Overview

[Figure 4-3](#) shows an overview of Docker Machine. Docker Machine installs a virtual machine on which Docker is installed. Docker and other tools, such as Docker Compose then can use this virtual machine as if it were the local computer.

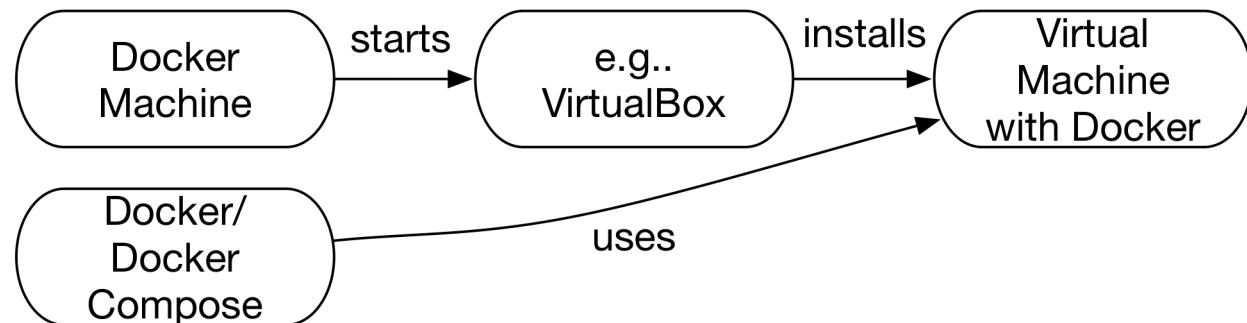


Fig. 4-3: Docker Machine

The command

```
docker-machine create --driver virtualbox dev
```

creates a Docker host with the name `dev` with the virtualization software Virtualbox. This requires that Virtualbox be installed on the computer.

Afterwards,

```
eval "$(docker-machine env dev)"
```

on Linux/macOS configures Docker in such a way that the docker command line tools use the Docker host in the virtual Virtualbox machine. If necessary, the shell used must be specified.

```
eval "$(docker-machine env --shell bash dev)"
```

For Powershell on Windows, the command is

```
docker-machine.exe env --shell powershell dev
```

and for cmd.exe on Windows, it is

```
docker-machine.exe env --shell cmd dev
```

`docker-machine rm dev` deletes the Docker host again.

## Docker Machine Drivers

Virtualbox is only one option. There are many more [Docker Machine drivers<sup>28</sup>](#) for cloud providers such as Amazon Web Services (AWS), Microsoft Azure, or Digital Ocean. In addition, there are drivers for virtualization technologies such as VMware vSphere or Microsoft Hyper-V. Using any of these, Docker Machine can easily install Docker hosts on many different environments.

### Advantage: Separate Environments and Docker on Servers

Docker Machine allows one to completely separate Docker systems from each other so that, for example, after a test, nothing remains of the system and all resources are indeed released again. In addition, Docker containers can thus be started very easily on a cloud or virtual infrastructure.

Running the examples in this book directly with Docker is the easiest option and therefore recommended. Docker Machine should be used for the examples only if they are to run on a server or should be completely separated from other Docker installations.

## 4.5 Dockerfiles

The creation of Docker images is done via files named `Dockerfile`. One of Docker's strengths is that Dockerfiles are easy to write and therefore, the rolling out of software can be automated without any problems.

The typical components of a `Dockerfile` are:

- `FROM` defines a base image on which the installation is based. A base image for a microservice usually contains a Linux distribution and basic software, such as the JVM, for example.
- `RUN` defines commands that are executed to create the Docker image. In essence, a `Dockerfile` is a shell script which installs the software.

---

<sup>28</sup><https://docs.docker.com/machine/drivers/>

- `CMD` defines what happens when the Docker container is started. Typically, only one process should run in one Docker container. This is started by `CMD`.
- `COPY` copies files in the Docker image. `ADD` does the same; however, it can also unpack archives and download files from a URL on the Internet. `COPY` is simpler to understand because it does not extract archives, for example. Also, from a security perspective, it can be problematic to download software from the Internet into Docker containers. Therefore, `COPY` should be given preference over `ADD`.
- `EXPOSE` exposes a port of the Docker container. This can then be contacted by other Docker containers or can be tied to a port of the Docker host.

A comprehensive [reference<sup>29</sup>](#) is available on the Internet which contains additional details to the commands in `Dockerfile`.

## An Example for a Dockerfile

A simple example of a `Dockerfile` for a Java microservice looks like this:

```
FROM openjdk:11.0.2-jre-slim
COPY target/customer.jar .
CMD /usr/bin/java -Xmx400m -Xms400m -jar customer.jar
EXPOSE 8080
```

- The first line defines the base image with `FROM`. It is downloaded from the public Docker hub. The image contains a Linux distribution and a Java Virtual Machine (JVM).
- The second line adds a JAR file to the image with `COPY`. A JAR file (Java ARchive) contains all components of a Java application. It has to be available in a sub directory `target` below the directory in which the `Dockerfile` is stored. The JAR file is copied into the root directory of the container.
- The `CMD` entry determines which process should be started when the container is started. In this example, a Java process runs the JAR file.
- Finally, `EXPOSE` makes a port available to the outside. This is the port under which the application is available. `EXPOSE` only means that the container provides the port. It is then available in the internal Docker network. Access from outside is only possible when this is enabled at the start of the container.

The Docker image can be built with the command `docker build --tag=customer`. `microservice-customer`.`docker` is the command line tool with which most functionalities of Docker can be controlled. The created Docker image has the tag `microservices-customer` as defined by the `--tag` parameter. The `Dockerfile` has to be in the sub directory `microservice-customer`. The name of this directory is the second parameter.

---

<sup>29</sup><https://docs.docker.com/engine/reference/builder/>

## File System Layers in the Example

Figure 4-1 shows that a Docker image consists of multiple layers. Although no layers have been defined in Dockerfile, the image `microservices-customer` contains multiple layers. Each line of the Dockerfile defines a new layer. These layers are reused. Thus, if `docker build` is called again, Docker will go through the Dockerfile again. However, it will find that all actions in Dockerfile have already been executed once. As a result, nothing happens. If the Dockerfile was modified in such a way that, after the `COPY`, another line with a `COPY` of another file is inserted, Docker would reuse the existing layer with the first `COPY`, but the second `COPY` and all further lines would then create new layers. In this manner, Docker only re-creates the layers that need to be rebuilt. This not only saves storage space but is also much faster.

## Problem with Caching and Layers

A Dockerfile for obtaining a Ubuntu installation with updates looks like this:

```
FROM ubuntu:15.04
RUN apt-get update ; apt-get dist-upgrade -y -qq
```

First, a Ubuntu base image is loaded from the public Docker hub on the Internet. The commands `apt-get update` and `apt-get dist-upgrade -y -qq` are used to update the package index and then install all packages with updates. The options ensure that `apt-get` does not ask the user for permission and outputs only a few messages on the console.

The two commands are separated in the line by a `;`. This causes them to be executed one after the other. A new file system layer is created only after both commands have been executed. This is useful for creating more compact images with fewer layers.

However, this Dockerfile also has a problem. If the Docker image is built again, no current updates will be downloaded. Instead, nothing happens because the images are already there. Layer caching is based only on commands. Docker does not recognize that the external package index has changed. To ignore the existing images and force the rebuilding of the images, the parameter `--no-cache=true` can be passed to `docker build`.

## Docker Multi Stage Builds

Everything needed to build a Docker image can also be found in the Docker image. Therefore, all of it is available at runtime of the Docker container. If code is being compiled in a Dockerfile, the compiler is also available at runtime. This is unnecessary. It can even be a security problem. If the container is compromised, the attacker can compile code inside the container with the original tools, which might allow more attacks. It is not easy to delete all of the build environment because today there is usually a complex tool chain for building software.

Building the software outside of Docker might also not be an option. Docker is based on Linux; therefore, on macOS, you would need to run a cross compiler to generate Linux binaries.

To solve this problem, Docker has Multi Stage Builds. They make it possible to compile the program in one phase of the build in a Docker image, and to transfer only the compiled program to the next phase into a different Docker image. Afterwards, the build tools are no longer available at runtime. They do not have to be deleted, and they do not have to be installed on the host machine.

[Section 5.4](#) shows a Docker Multi Stage Build using a Go program as example.

## Immutable Server with Docker

*Immutable server* is an idea that predates Docker. The idea of an immutable server is that a server will never be changed; therefore, the software on the server will never be updated or modified. The server will always be completely rebuilt from scratch. In this way, the state of the server can be reconstructed cleanly. So for each server, an installation script installs all the needed software on a basic OS image.

However, immutable servers are hard to implement. It is very cumbersome to completely reinstall a server. The process might take minutes or hours. That is far too much time to, for example, just change a configuration file.

This is exactly where Docker helps. Because of the optimizations, only the necessary steps are taken, so that *immutable servers* can also be an option from this perspective. A `Dockerfile` describes how to create a Docker image starting from a base image. With each build, it will seem as if the complete Docker image is being created. Behind the scenes, however, optimizations ensure that only what is really necessary is built. For example, if in the very last step a configuration file is added and just that configuration file has been modified, Docker is smart enough to reuse the results of all other installation steps and just add the new configuration file. This just takes a few seconds.

So Docker is conceptually as clear as an immutable server but much more efficient for actually implementing them.

## Docker and Tools Such as Puppet, Chef or Ansible

Besides immutable servers, there are other ways to handle the installation of software. *Idempotent installation* means that an installation script provides the same results no matter how often it runs. For an idempotent installation, there are no steps like “install the Java package,” but rather a definition of the desired state: “Ensure that the Java package is installed”. So if the installation is run on a fresh OS, Java would be installed. If the installation is run on a system that already has Java installed, nothing happens.

Idempotent installation is particularly useful to enable updates. For each update the script would check if all software is installed in the correct version. If that is not the case, the correct version is installed. Tools such as Puppet, Chef, or Ansible support the concept of idempotent installation.

A `Dockerfile` is a very easy way to install software. The use of tools such as Puppet, Chef, or Ansible for the installation of software in a Docker image is possible, but does not make a lot of sense. In particular, it is not necessary to use the update functionalities of these tools, because the image is

typically freshly built with the *immutable server* approach. This approach is easier than writing Puppet, Chef or Ansible scripts because defining the desired state is usually quite complex. The `Dockerfile` only describes how to build an image, whereas the other tools must also enable updates of the servers and are therefore more difficult to use.

## 4.6 Docker Compose

A typical microservices system contains more than a single Docker container. As explained in chapter 1, microservices are modules of a system. So it would be good to have a way to start and run several containers together for starting all the modules that the system consists of in one go. This can be done with [Docker Compose](#)<sup>30</sup>.

### Service Discovery with Docker Compose Links

Actually coordinating a system of multiple Docker containers requires more than just starting multiple Docker containers. It also requires configurations for the virtual network with which the Docker containers communicate with each other. In particular, containers must be able to find each other in order to communicate. In a Docker Compose environment, a service can simply contact another service via a Docker Compose link and then use the service name as host name. So it could use a URL like `http://order/` to contact the order microservice.

Docker Compose links offer some kind of service discovery – that is, a way for microservices to find other microservices. Synchronous microservices require a form of service discovery (see also section 13.3).

Docker Compose links extend Docker links. Docker links only allow communication. Docker Compose links also implement load balancing and set the start order so that the dependent Docker containers start first.

### Ports

In addition, Docker Compose can bind ports from the containers to the ports of the Docker host where the Docker containers run.

### Volumes

Docker Compose can also provide volumes. These are file systems that can be shared by multiple containers. This allows containers to communicate by exchanging files.

---

<sup>30</sup><https://docs.docker.com/compose/>

## YAML Configuration

Docker Compose configures the interaction of the Docker containers with a YAML configuration file `docker-compose.yml`.

The following file comes from a project for [chapter 9](#), which implements Edge Side Includes as a way to compose websites from different sources. For this purpose, three containers must be coordinated.

- `common` is a web application that is supposed to deliver common artifacts.
- `order` is a web application for processing orders.
- `varnish` is a web cache to coordinate the two web applications.

```
1 version: '3'
2 services:
3   common:
4     build: ../scs-demo-esi-common/
5   order:
6     build: ../scs-demo-esi-order
7   varnish:
8     build: varnish
9     links:
10    - common
11    - order
12   ports:
13    - "8080:8080"
```

- The first line defines the used version of Docker Compose – in this case three.
- The second line starts the definition of the services.
- Line three defines the service `common`. The directory specified in line four contains a `Dockerfile` with which the service can be built. An alternative to `build` would be `image` to use a Docker image from a Docker registry.
- The definition of the service `order` also specifies a directory with a `Dockerfile`. No other settings are required for this service (lines 5/6).
- The service `varnish` is also defined by a directory with a `Dockerfile` (lines 7/8).
- The service `varnish` must have Docker Compose links to the services `common` and `order`. Therefore, it has entries under `links`. The `varnish` service can therefore reach the other services using the host names `common` and `order` (lines 9-11).
- Finally, port 8080 of the service `varnish` is bound to port 8080 of the Docker host, on which Docker containers run (lines 12-13).

## Additional Options

Further elements of the YAML configuration are described in the [reference documentation<sup>31</sup>](#). For example, Docker Compose supports volumes shared by multiple Docker containers. Docker Compose can also configure the Docker containers using environment variables.

## Docker Compose Commands

Docker Compose is controlled by the command line tool `docker-compose`. It must be started in the directory where the file `docker-compose.yml` is stored. The [reference documentation<sup>32</sup>](#) lists all command line options for this tool. [Appendix C](#) shows an overview of the Docker Compose commands. The most important ones are:

- With `docker-compose up`, all services are started. The command returns the combined standard output of all services. This is rarely helpful, so `docker-compose up -d` is often the better choice. In this case, the standard output is not returned. With `docker log` the output of individual containers can be viewed.
- `docker-compose build` builds the images for the services.
- `docker-compose down` shuts down all services and deletes the containers.
- With `docker-compose up --scale <service>=<number>`, a larger number of containers for a service can be started. In the example from the listing, `docker-compose up --scale common=2` could ensure that two containers for the service `common` are started.

Since the examples often require the interaction of several Docker containers, most examples have a `docker-compose.yml` file to run the containers together.

## 4.7 Variations

There are not really any fundamental alternatives to Docker:

- *Virtualization* has too much overhead.
- *Processes* are not sufficiently isolated. The required libraries and runtime environments for all microservices must be installed in the operating system. This can be difficult because each process occupies one port, and therefore the allocation of ports must be coordinated.
- Other *container solutions* such as [rkt<sup>33</sup>](#) are far less common.

---

<sup>31</sup><https://docs.docker.com/compose/compose-file/>

<sup>32</sup><https://docs.docker.com/compose/reference/overview/>

<sup>33</sup><https://coreos.com/rkt>

## Clusters

For production, applications should run in a cluster. Only in this way can the system be scaled across several servers and secured against the failure of individual servers.

Docker Compose can use [Docker Swarm Mode<sup>34</sup>](#) for cluster management. Docker Swarm Mode is built into Docker.

Kubernetes is widely used for operating Docker containers in a cluster (see [section 17](#)). There are also other systems like [Mesos<sup>35</sup>](#). Mesos is actually a system for managing batches for data analysis in a cluster, for example, but it also supports Docker containers. Offers in the cloud are also available, such as AWS (Amazon Web Services) and [ECS<sup>36</sup>](#) (EC2 Container Service).

These approaches have one thing in common. How the load is distributed in the cluster is decided by the scheduler – that is, by Kubernetes or Docker Swarm Mode. This means that the scheduler is of crucial importance for fail-safety and load balancing. If a container fails, a new container must be started. Likewise, additional containers must be started at times of high loads, ideally on machines that are not too busy.

Schedulers such as Kubernetes solve many challenges, especially with synchronous microservices. It is highly recommended to use such a platform for operating microservices. But the introduction of microservices requires many changes. The architecture needs to be adapted, and developers need to learn new approaches and technologies, as well as having adapt the deployment pipeline and the tests. Implementing an additional technology such as a Docker scheduler should therefore be well thought out, because many other changes also have to be made.

## Docker without Scheduler

An alternative is to install Docker containers directly on a server. In this scenario, the servers are provided with classic mechanisms – for example, with virtualization. Linux or Windows is installed on the server. The microservice runs in a Docker container.

The only difference to the procedure without Docker is that Docker containers are now used for deployment. But this already makes it much easier to keep production and test environments identical. Concepts such as *immutable servers* are also easier to implement, as is the technology freedom for microservices. Traditional virtualization is still responsible for high availability, scaling, and distribution to the servers.

So this approach offers some of the advantages of Docker and reduces the effort.

## PaaS

Another alternative is a PaaS (see [chapter 18](#)). A PaaS has a higher degree of abstraction than Docker schedulers because only the application needs to be provided. The PaaS creates the Docker images.

---

<sup>34</sup><https://docs.docker.com/engine/swarm/>

<sup>35</sup><http://mesos.apache.org/>

<sup>36</sup><https://aws.amazon.com/documentation/ecs/>

Therefore, a PaaS can be the simpler and therefore better solution. Chapter 18 shows Cloud Foundry as a concrete example of a PaaS.

## Experiments

- Docker Machine can use clouds like Amazon Cloud or Microsoft Azure with the appropriate drivers<sup>37</sup>. Create an account with one of the cloud providers. Most cloud providers offer free capacity to a new user. Install Docker with Docker Machine and the appropriate driver on the virtual machines in the cloud, and use this configuration to run one of the examples from the following chapters (see section 4.4).
- Create an account in the Docker Hub<sup>38</sup>. Build a Docker image, for example, based on one of the microservices examples of the following chapters and place it in the Docker hub with docker push.
- Use the tutorials<sup>39</sup> to familiarize yourself with the Docker Swarm Mode, which can be used to run Docker in a cluster.

## 4.8 Conclusion

Docker is a light weight alternative for deploying and operating microservices. A microservice with all its dependencies can be packed into a Docker image and can then be well isolated from other microservices as a Docker container. Virtual machines appear too heavy weight by comparison, whereas simple processes do not provide the necessary isolation.

Docker makes it easier to deploy the software. It is only necessary to distribute Docker images. Dockerfiles are used for this purpose, which are very easy to write. Concepts such as *immutable server* are also much easier to implement. With Docker Compose, multiple containers can be coordinated to thereby build and launch an entire system of microservices in Docker containers. Docker Machine can very easily install Docker environments on servers.

However, Docker requires rethinking regarding operation. Therefore, in some cases, alternatives might be helpful. This can be, for example, the deployment of several Java web applications on a single Java web server.

---

<sup>37</sup><https://docs.docker.com/machine/drivers/>

<sup>38</sup><https://hub.docker.com/>

<sup>39</sup><https://docs.docker.com/engine/swarm/swarm-tutorial/>

# 5 Technical Micro Architecture

One of the strengths of microservices is that different technologies can be used in each individual microservice. The technologies in the microservices can be defined as part of the micro architecture (see [chapter 2](#)).

However, there are technical challenges to consider when selecting technologies for microservices.

This chapter explains how to deal with the technical micro architecture.

- The reader gets to know the *requirements* in regards to, for example, operation or resilience which the micro architecture has to fulfill.
- Often microservices are implemented with *reactive technologies*. Thus, this chapter discusses this option in more detail and explains when this approach makes sense.
- As a concrete example of a technical micro architecture, this chapter shows *Spring Boot* and *Spring Cloud*, which are used in the following chapters for most examples.
- Based on Spring Boot and Spring Cloud, this chapter shows how the *technical requirements* that the micro architecture has to address can be fulfilled.
- In addition, this chapter shows how the programming language *Go*, in conjunction with appropriate frameworks, fulfills the requirements for implementing microservices.

## 5.1 Requirements

A technology for implementing microservices has to fulfill different requirements (see [figure 5-1](#)).

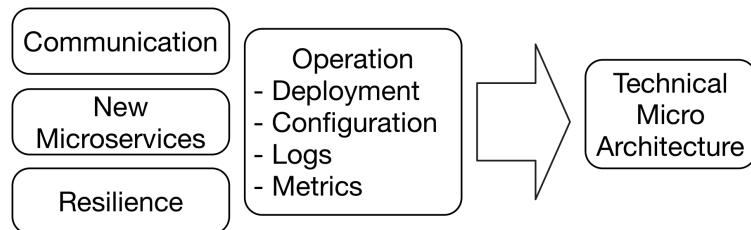


Fig. 5-1: Influencing Factors for the Technical Micro Architecture

### Communication

Microservices have to communicate with other microservices. This requires a UI integration in the web UI or protocols such as REST or messaging. It is a macro architecture decision in which communication protocol is used (see [section 2.2](#)). However, the microservices have to support the chosen communication mechanism. Therefore, the macro architecture decision influences the

micro architecture. The technology choices at the micro architecture level have to ensure that the communication protocol defined by the macro architecture can really be implemented in each microservice.

In principle, every modern programming technology can support the typical communication protocols. Therefore, this requirement does not represent a real restriction.

## Operation

Operating the microservices should be as easy as possible. Topics in this area are:

- *Deployment*: The microservice has to be installed in an environment and has to run in this environment.
- The application's *configuration*: The microservice has to be adapted to different scenarios. It is possible to use custom code for reading the configuration. However, an existing library can facilitate this task and promote a uniform configuration.
- *Logs*: Writing log files is easy. However, the format should be uniform for all microservices. In addition, [chapter 21](#) shows that a simple log file is not enough when a server has to collect the logs from all microservices and provide them for analysis. Therefore, technologies have to be in place for formatting the log outputs and for sending them to the server where all logs are stored and analyzed.
- *Metrics* have to be delivered to the central monitoring infrastructure. This requires appropriate frameworks and libraries.

In principle, different libraries can be used for implementing a macro architecture rule which, for example, predefines a log format and a log server. In this case, the micro architecture has to choose a library for the microservice. Macro architecture rules can also determine the library. However, this limits the technological freedom of the microservices to those programming languages which can use the chosen library.

## New Microservices

It should be easy to create new microservices. When a project over time accumulates more and more code, there are two options. Either the microservices become larger, or the number of microservices of constant size increases. If the microservices increase in size, at some point they will not *deserve* the name *microservice* anymore. To avoid this, it is important that new microservices can be easily generated to keep the size of the individual microservices constant over time.

## Resilience

Each microservice has to be able to deal with the failure of other microservices. This has to be ensured when microservices are implemented.

## 5.2 Reactive

One way to implement a microservice is *reactive programming*. Oftentimes it is stated that microservices must be implemented with reactive technologies. This section discusses what reactive actually is and determines whether reactive technologies are truly needed for microservices.

Similar to microservices, *reactive* has an ambiguous definition. The [Reactive Manifesto<sup>40</sup>](#) defines the term “reactive” based on the following characteristics:

- *Responsive* means that the system responds as fast as possible.
- Because of *resilience*, the system remains available even if parts fail.
- *Elastic*: The system can deal with different levels of load, for example, by using additional resources. After the load peak, the resources are freed again.
- The system uses asynchronous communication (*message driven*).

These characteristics are useful for microservices. They pretty much correspond to the features discussed in [chapter 1](#) as essential characteristics of microservices.

So at first sight, it seems that microservices in fact must be written with reactive technologies.

### Reactive Programming

However, [reactive programming<sup>41</sup>](#) means something completely different. This programming concept resembles the data flow. When new data comes into the system, it is processed. A spreadsheet is an example. When the user changes a value in a cell, the spreadsheet recalculates all dependent cells.

### Classical Server Applications

A similar approach is possible for server applications. Without reactive programming, a server application typically processes an incoming request in a thread. If the processing of the request requires a call to a database, the thread blocks until the result of this call arrives. In this model, a thread has to be provided for each request that is processed in parallel and for each network connection.

### Reactive Server Applications

Reactive server applications behave very differently. The application only reacts to events. It must not block because it is waiting, as an example, for I/O, or an event such as an incoming HTTP request. If a request arrives, the application executes the logic and then sends a call to the database at some point. However, subsequently, the application does not wait for the result of the call to the

<sup>40</sup><http://www.reactivemanifesto.org/>

<sup>41</sup>[https://en.wikipedia.org/wiki/Reactive\\_programming](https://en.wikipedia.org/wiki/Reactive_programming)

database, but suspends processing the HTTP request. Eventually, the next event arrives, namely the result of the call to the database. The processing of the HTTP request then resumes. In this model, only one thread is needed. It processes the respective current event.

Figure 5-2 shows an overview of this approach. The event loop is a thread and processes one event at a time. Instead of waiting for I/O, the processing of the event is suspended. Once the results of the I/O operation are available, they are part of a new event processed by the event loop. In this way, a single event loop can process a plethora of network connections. However, processing of the event must not block the event loop for longer than is absolutely necessary. Otherwise, processing of all events is stopped.

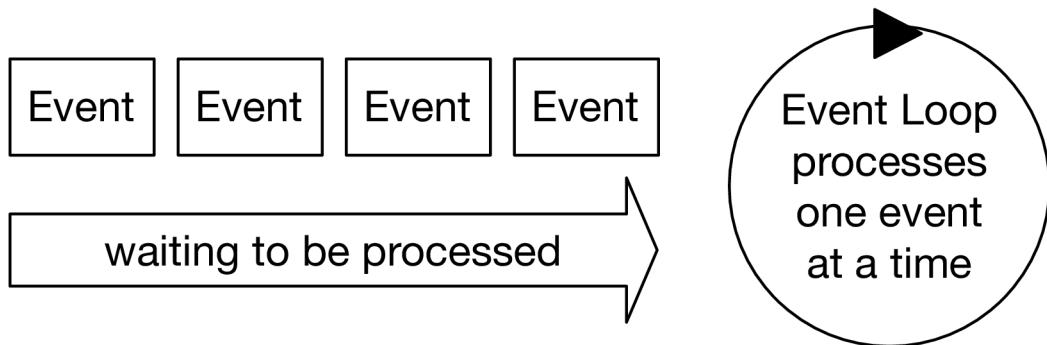


Fig. 5-2: Event Loop

## Reactive Programming and the Reactive Manifesto

Reactive programming can support the goals of the Reactive Manifesto:

- *Responsive*: The model can cause the application to respond faster because fewer threads are blocked. However, whether this really leads to an advantage over a classical application depends on how efficiently the threads are implemented in the system and how efficiently it handles blocked threads.
- *Resilience*: If a service no longer responds, nothing is blocked in reactive programming. This helps with resilience. However, in a classical application for example, a timeout can avoid a blockage by aborting the processing of the request.
- *Elastic*: With a higher load, more and more instances can be started. This is also possible with the classical programming model.
- *Message driven*: Reactive programming does not affect the communication between the services. Therefore, communication may or may not be message driven in reactive programming or in classical applications.

## Reactive Programming Is Not Necessary for Microservices

The Reactive Manifesto is certainly relevant for microservices. But a microservice does not have to be implemented with reactive programming in order to achieve the goals of the Reactive Manifesto.

Whether or not a microservice is implemented with reactive programming can be different for each microservice. This can be a micro architecture decision and, therefore, affects only individual microservices, but not the system as a whole.

It is important to understand the difference, because otherwise the choice of technologies might be limited to reactive programming frameworks, even though that is not necessary. It is perfectly fine to stay with established technologies. In fact, using a technology stack that you are used to might be easier and bring faster results. At the same time, it is possible to try new technologies like reactive programming in one microservice, and then use it in other microservices if it has proven to be useful.

## 5.3 Spring Boot

The Spring Framework has long been part of the Java community. It has a broad set of features covering most of the technical requirements of typical Java applications. [Spring Boot<sup>42</sup>](#) facilitates the use of Spring.

A minimal Spring Boot application can be found in the directory `simplest-spring-boot` of the project <https://github.com/ewolff/spring-boot-demos>.

### Java Code

The Java code from the project shows how Spring Boot can be used.

```
@RestController
@SpringBootApplication
public class ControllerAndMain {

    @RequestMapping("/")
    public String hello() {
        return "hello\n";
    }

    public static void main(String[] args) {
        SpringApplication.run(ControllerAndMain.class, args);
    }
}
```

The annotation `@RestController` means that the class `ControllerAndMain` should process HTTP requests. `@SpringBootApplication` triggers the automatical configuration of the environment. So the application starts an environment with a web server and with the parts of the Spring framework that are fitting for a web application.

---

<sup>42</sup><https://projects.spring.io/spring-boot/>

The method `hello()` is annotated with `@RequestMapping`. Therefore, the method is called upon an HTTP request to the URL `"/"`. The method's return value is returned in the HTTP response.

Finally, the main program `main` starts the application with the help of the class `SpringApplication`. The application can simply be started as a Java application even though it processes HTTP requests. A web server is required for handling HTTP in the Java world. It is included in the application.

## Build

For compiling the project, Spring Boot supports, among others, [Maven<sup>43</sup>](#).

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.ewolff</groupId>
  <artifactId>simplest-spring-boot</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.2.RELEASE</version>
  </parent>

  <properties>
    <java.version>10</java.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
```

---

<sup>43</sup><https://maven.apache.org/>

```
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>

</project>
```

The build configuration inherits settings from the parent configuration `spring-boot-starter-parent`. Maven's parent configuration makes it easy to reuse settings for the build of multiple projects. The version of the parent determines which version of Spring Boot is used. The Spring Boot version defines the version of the Spring framework and the versions of all other libraries. Thus, the developer does not have to define a stack with compatible versions of all frameworks, which is otherwise often a challenge.

## Spring Boot Starter Web as Single Dependency

The application has a dependency on the library `spring-boot-starter-web`. This dependency integrates the Spring framework, the Spring web framework, and an environment for the processing of HTTP requests. The default for the processing of the HTTP requests is a Tomcat server, which runs embedded as part of the application.

Thus, the dependency on `spring-boot-starter-web` would be enough as sole dependency for the application. The dependency on `spring-boot-starter-test` is necessary for tests. The code for the test is not shown in this chapter.

## Spring Cloud

[Spring Cloud](#)<sup>44</sup> is a collection of extensions for Spring Boot, which are useful for cloud applications and for microservices. Spring Cloud contains additional starters. To be able to use the Spring Cloud starters, an entry has to be inserted into the `dependency-management` section in the `pom.xml` for importing the information about the Spring Cloud starter. The `pom.xml` files in the examples already contain the required import for this.

## Maven Plug In

The Maven plug-in `spring-boot-maven-plugin` is necessary to build a Java JAR that starts an environment with the Tomcat server and the application. `mvn clean package` deletes the old build results and builds a new JAR. JARs are a Java file format which contains all the code for an application. Maven gives this file a name derived from the project name. It can be started with `java -jar simplest-spring-boot-0.0.1-SNAPSHOT.jar`. Spring Boot can also generate WARs (web archives) which can be deployed on a Java web server like Tomcat or a Java application server.

---

<sup>44</sup><http://projects.spring.io/spring-cloud/>

## Spring Boot for Microservices?

The suitability of Spring Boot for the implementation of microservices can be decided according to the criteria of [section 5.1](#).

## Communication

For communication, Spring Boot supports REST, as the previous listing shows. The listing uses the Spring MVC API. Spring Boot also supports the JAX RS API. For JAX RS, Spring Boot uses the library Jersey. JAX RS is standardized as part of the Java Community Process (JCP).

For messaging, Spring Boot supports the Java Messaging Service (JMS). This is a standardized API that can be used to address different messaging solutions from Java. Spring Boot has starters for the JMS implementations [HornetQ<sup>45</sup>](#), [ActiveMQ<sup>46</sup>](#), and [ActiveMQ Artemis<sup>47</sup>](#). In addition, there is a Spring Boot starter for [AMQP<sup>48</sup>](#). This protocol is also a standard, but on the network protocol level. The AMQP starter uses [RabbitMQ<sup>49</sup>](#) as an implementation of the protocol.

For AMQP as well as for JMS, Spring offers an API that makes it easier to send messages. In addition, simple Java objects (Plain Old Java Objects, POJOs) with no dependencies on any of the APIs can process AMQP and JMS messages with Spring and also return responses to messages.

Spring Cloud offers [Spring Cloud Streams<sup>50</sup>](#) for implementing applications for the processing of data streams. This library supports messaging systems such as Kafka (see also [chapter 11](#)), RabbitMQ (see above), and [Redis<sup>51</sup>](#). Spring Cloud Streams builds on these technologies and extends them with concepts such as streams, and therefore goes beyond just simplifying the use of the technology's APIs.

The integration of technologies in Spring Boot with Spring Boot starters has the advantage that Spring Boot provides the configuration of the environment. The example Spring Boot application in this chapter uses an infrastructure such as a Tomcat server to handle HTTP requests. This does not require a separate configuration, nor any additional dependencies. Spring Boot starters also offer such simplifications for messaging and other REST technologies.

Spring Boot applications can also use technologies without a Spring Boot starter. A Spring Boot application can use any technology that supports Java. In the end, a Spring Boot project is a Java project and can be extended with Java libraries. However, it is possible that the configuration can be more complex than when using a Spring Boot starter.

## Operation

Spring Boot also has some interesting approaches for operation.

<sup>45</sup><http://hornetq.jboss.org/>

<sup>46</sup><http://activemq.apache.org/>

<sup>47</sup><https://activemq.apache.org/artemis/>

<sup>48</sup><https://www.amqp.org/>

<sup>49</sup><https://www.rabbitmq.com/>

<sup>50</sup><https://cloud.spring.io/spring-cloud-stream/>

<sup>51</sup><https://redis.io/>

- To deploy a Spring Boot application, it is enough to just copy the JAR file to the server and start it. Deploying a Java application can't be further simplified.
- Spring Boot offers numerous options for the [configuration<sup>52</sup>](#). For example, a Spring Boot application can read the configuration from a configuration file or from an environment variable. Spring Cloud offers support for Consul (see [chapter 15](#)) as server for configurations. The examples in this book use `application.properties` files for configuration because they are relatively easy to handle.
- Spring Boot applications can generate [logs<sup>53</sup>](#) in many different ways. Usually, a Spring Boot application displays the logs in the console. Output to a file is also possible. [Chapter 21](#) shows a Spring Boot application that sends the logs as JSON data to a central server instead of using a simple human-readable text format. JSON facilitates the processing of log data on this server.
- For *metrics*, Spring Boot offers a special starter, namely the [Actuator<sup>54</sup>](#). After adding a dependency to `spring-boot-starter-actuator`, the application collects metrics, for example, about the HTTP requests. In addition, Spring Boot Actuator provides REST endpoints under which the metrics are available as JSON documents. The example in [chapter 20](#) is a Spring Boot application that exports the data to the monitoring tool Prometheus based on Actuator. Spring Boot Actuator does not support Prometheus, but custom code can extend Actuator in such a way that other monitoring systems can also be integrated.

## New Microservices

Creating a new microservice is very easy with Spring Boot. A build script and a main class are enough, as shown in the example [simplest-spring-boot<sup>55</sup>](#). To further simplify the creation of a new microservice, a template can be created. The template only needs to be adapted for a new microservice; the template can define settings for the configuration of the microservices or for logging. Thus, a template simplifies the creation of new microservices and facilitates compliance with macro architecture rules.

A particularly easy way to create a new Spring Boot project is to use <http://start.spring.io/>. The developer must select the build tool, the programming language, and a Spring Boot version. In addition, he or she can select different starters. Based on this, the website then creates a project that can be the basis for the implementation of a microservice.

## Resilience

For resilience a library like Hystrix (see [section 14.5](#)) can be useful. Hystrix implements typical resilience patterns such as timeouts in Java. Spring Cloud offers an integration and further simplification for Hystrix.

---

<sup>52</sup><https://docs.spring.io/spring-boot/docs/2.1.2.RELEASE/reference/html/boot-features-external-config.html>

<sup>53</sup><https://docs.spring.io/spring-boot/docs/2.1.2.RELEASE/reference/html/boot-features-logging.html>

<sup>54</sup><https://docs.spring.io/spring-boot/docs/2.1.2.RELEASE/reference/html/production-ready.html>

<sup>55</sup><https://github.com/ewolff/spring-boot-demos/tree/master/simplest-spring-boot>

## 5.4 Go

Go<sup>56</sup> is a programming language that is increasingly being used for microservices. Similar to Java, Go is based on the programming language C. But in many areas, Go is fundamentally different.

### Code

A part of the example in [chapter 9](#) is implemented with Go.

This Go program responds to HTTP requests with HTML code.

```
package main

import (
    "fmt"
    "time"
    "log"
    "net/http"
)

func main() {
    http.Handle("/common/css/",
        http.StripPrefix("/common/css/",
            http.FileServer(http.Dir("/css"))))
    http.HandleFunc("/common/header", Header)
    http.HandleFunc("/common/footer", Footer)
    http.HandleFunc("/common/navbar", Navbar)
    fmt.Println("Starting up on 8180")
    log.Fatal(http.ListenAndServe(":8180", nil))
}

// Header and Navbar left out

func Footer(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintln(w,
        `<script src="/common/css/bootstrap-3.3.7-dist/js/bootstrap.min.js" />`)
}
```

The key word `import` imports some libraries, among others, for HTTP. The main program `main` defines which methods should respond to which URLs. For example, the method `Footer` returns HTML code. On the other hand, for the URL `/common/css`, the application delivers content from files.

---

<sup>56</sup><https://golang.org/>

As you can see, it is also very easy to implement a REST service with Go. In addition, libraries like [Go kit<sup>57</sup>](#) offer many more functionalities to implement microservices.

## Build

Go compilers are particularly well suited for Docker environments because they can create static binaries. Static binaries do not require any further dependencies or a specific Linux distribution. However, the applications must be compiled to Linux binaries. This requires a Go environment that can create Linux binaries.

## Docker Multi Stage Builds

The example in [chapter 9](#) uses Docker multi stage builds. Such a build divides the build process of the Docker image into several stages. The first stage can compile the program in a Docker container with a Go build environment. The second stage can execute the Go program in a Docker container as a runtime environment that contains only the compiled program. Consequently, the runtime environment has no build tools and is therefore much smaller.

Docker multi stage builds are not very complicated, as a look at the `Dockerfile` shows.

```

1 FROM golang:1.8.3-jessie
2 COPY /src/github.com/ewolff/common /go/src/github.com/ewolff/common
3 WORKDIR /go/src/github.com/ewolff/common
4 RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o common .
5
6 FROM scratch
7 COPY bootstrap-3.3.7-dist /css/bootstrap-3.3.7-dist
8 COPY --from=0 /go/src/github.com/ewolff/common/common /
9 ENTRYPOINT ["/common"]
10 CMD []
11 EXPOSE 8180

```

The base image `golang` contains the Go installation (line 1). Into this image, the Go source code is copied (line 2) and compiled (line 3/4). With that, stage 0 of the build is finished.

Stage 1 creates a new Docker image. The image `scratch` (line 6) is an empty Docker image. The `Dockerfile` copies the Bootstrap library (line 7) and the compiled Go binary from stage 0 (line 8) into this image. The option `--from=0` indicates that the file `common` originates from stage 0 of the Docker build.

Finally, `ENTRYPOINT` defines the binary that is supposed to be started (line 9). `CMD` (line 10) indicates that no options are to be passed to the binary at the start. Normally, `ENTRYPOINT` would be a shell which starts the process configured with `CMD`. However, in the `scratch` image there is no shell.

---

<sup>57</sup><https://github.com/go-kit/kit>

Therefore, `ENTRYPOINT` replaces the shell by the Go binary `common`, and `CMD` defines that this binary is to be started without options. `EXPOSE` makes port 8180, on which the process is listening, available to the outside.

## Multi Stage Builds: Advantages

Multi stage builds have a number of advantages.

- Build and runtime are clearly *separated*. In the runtime environment, there are no build tools provided.
- *No Go environment* has to be installed on the local computer. Otherwise, an environment for cross-compiling Go to Linux would have to be installed on the local computer because the target system is a Linux Docker container.
- The image is very small. It is just 5.91 MB. 5.92 MB of this is the Go binary and 984 kB the Bootstrap library.
- The image does not contain a Linux distribution and therefore has a minimal attack surface from a security point of view.

However, a multi stage build creates a new Docker image for each build, which is only needed for the build process and can be removed afterwards. At some point, these images have to be deleted. But these images are stored only on the build system or the developer computer, so that all old Docker images can simply be deleted at a defined point in time, for example, with `docker image prune`.

## Go for Microservices?

The criteria from [section 5.1](#) for the implementation of microservices can serve as a basis to assess Go's suitability as a microservices programming language.

## Communication

Go supports REST in the standard libraries. Libraries are also available for messaging systems such as AMQP – for example, <https://github.com/streadway/amqp>. There is also a library for messaging with [Redis<sup>58</sup>](#). Due to the widespread use of Go, there is hardly any communication infrastructure that does not support Go.

## Operation

Go also offers many options for operation.

- The *deployment* in a Docker container is very easy with Docker multi stage builds, as already illustrated.

---

<sup>58</sup><https://github.com/go-redis/redis>

- Libraries like [Viper<sup>59</sup>](#) support the *configuration* of Go applications. This library supports formats such as YAML or JSON.
- Go itself already offers support for logs. The Go microservices framework Go Kit contains additional features for [logs<sup>60</sup>](#) in more complex scenarios.
- For *metrics*, [Go Kit<sup>61</sup>](#) supports a plethora of tools such as Prometheus (see [chapter 20](#)), but also Graphite or InfluxDB.

## New Microservices

For a new microservice, it is enough to create the Docker build and then write the source code. That is not very elaborate.

## Resilience

Go Kit contains an implementation of resilience patterns such as [Circuit Breaker<sup>62</sup>](#). In addition, there is a port of the [Hystrix library<sup>63</sup>](#) for Go.

## 5.5 Variations

The technical micro architecture decisions can be made differently for each microservice. But a connection with the macro architecture exists. The uniformity of the operational aspects can be enforced by the macro architecture. If you want to implement a microservice with other technologies in a Spring Boot microservices architecture, this can lead to a lot of effort.

A macro architecture decision could be to read out configurations from an `application.properties` file. This decision does not restrict the choice of implementation technologies. But for a Spring Boot application, the implementation is very simple because this mechanism is built into Spring Boot and is the default for Spring Boot applications. A Go application, on the other hand, would have to be adapted to this requirement.

This effect supports a uniform choice of technology for the microservices because implementing a microservice with Spring Boot is easier. Therefore, developers will prefer Spring Boot. A uniform choice of technology has further advantages. For example, developers are more likely to find their way around in other microservices, and developers of different microservices can help each other out with technology issues.

In order to really treat other technologies on an equal footing, a different macro architecture decision should be made. Spring Boot offers [many more options<sup>64</sup>](#). For example, the configuration can be stored in environment variables, transferred via the command line, or read from a configuration server.

---

<sup>59</sup><https://github.com/spf13/viper>

<sup>60</sup><https://godoc.org/github.com/go-kit/kit/log>

<sup>61</sup><https://godoc.org/github.com/go-kit/kit>

<sup>62</sup><https://godoc.org/github.com/go-kit/kit/circuitbreaker>

<sup>63</sup><https://github.com/afex/hystrix-go>

<sup>64</sup><https://docs.spring.io/spring-boot/docs/1.5.6.RELEASE/reference/html/boot-features-external-config.html>

## Alternatives to Spring Boot

In the Java area, there are some alternatives to Spring Boot.

- A classic Java EE application with an application server or a web server is also conceivable as an implementation for a microservice. However, in this case, deployment is more complex because the application server also has to be installed. In addition, application servers and applications must be configured, in some cases even with two different technologies. Any any event, there are doubts about the [usefulness of application servers<sup>65</sup>](#).
- [Wildfly Swarm<sup>66</sup>](#) provides a simple JAR deployment. However, instead of Spring APIs, it implements the standardized Java EE APIs and supplements them with technologies from the microservices area such as Hystrix.
- [Dropwizard<sup>67</sup>](#) has long been offering the possibility of developing Java REST services and deploying them as JARs.

Of course, a lot of other possible choices for the programming language apart from Java or Go are also available. It is impossible to even list them in this book. Actually, the point this book makes is that the technologies for the implementation of each microservice are not that important. It is easily possible to implement each microservice with a different programming language and framework, and so the decision can easily be changed. However, it is much harder to change the technologies for communication, integration and operations that this book focuses on.

The criteria from [section 5.1](#) are a yardstick to check the technologies for their suitability for microservices, as the [section 5.3](#) does for Spring Boot and [section 5.4](#) for Go. Such an assessment is recommended for each technology used. The examples in [chapter 8](#) are mostly implemented with Node.js and JavaScript. This shows that microservices can also be implemented with completely different technologies.

## 5.6 Conclusion

Individual microservices can differ greatly in their technical micro architecture. This freedom is an important advantage of microservices architectures.

- The macro architecture and the challenges associated with implementing microservices can be used to derive requirements for micro architecture and microservices technologies.
- Reactive programming can be used to implement microservices, but this is not mandatory to meet the requirements.
- Spring Boot and Java meet the requirements, just like Go does, with the appropriate libraries.
- There are also many other alternatives.

---

<sup>65</sup><https://jaxenter.com/java-application-servers-dead-112186.html>

<sup>66</sup><http://wildfly-swarm.io/>

<sup>67</sup><http://www.dropwizard.io/>

Since each microservice can use a different micro architecture and other technologies, the technical decisions at this level are not so important. They can be revised in any microservice. The rest of this book mainly discusses technologies that have an impact on the macro architecture and thus the entire system because these technologies have much wider implications.

# 6 Self-contained Systems

A Self-contained System (SCS) is a type of microservice that specifies elements of a macro architecture. SCSs do not represent a complete macro architecture. The area of operation, for instance, is completely missing.

The idea behind SCS is to provide microservices that truly provide everything needed for the implementation of a part of the domain logic, and are therefore self-contained. This means an SCS includes logic, data, and a UI. That also means if a change impacts all technical layers, it can still be contained in one SCS, making it easier to perform the change and put it into production. So an SCS for payment would store all information relevant to payment – that is, it would implement a bounded context. But it would also implement the UI – web pages to show the payment history or do a payment. Data about customers or ordered items would need to be replicated from other SCSs.

This chapter covers the following:

- First, it describes the reasons for using SCSs.
- SCSs determine different macro architecture decisions. This chapter explains the reasons for these macro architecture decisions and discusses which advantages result from the different decisions.
- SCSs are only a variation of microservices. This chapter addresses the differences between the terms “SCS” and “microservice.”
- This chapter then discusses the benefits of the SCS approach.
- Finally, challenges connected to the development of an SCS system are described and potential solutions are discussed.

SCS include a UI and focus on UI integration. Therefore, SCS are a good motivation for the UI integration techniques the next chapters explain.

## 6.1 Reasons for the Term Self-contained Systems

There is no uniform definition for microservices. Self-contained systems, however, are very precisely defined. You can read the definition of SCSs on the <http://scs-architecture.org> website. The content of the site is provided under a creative commons license, so that the material on the site may be reused by anyone if the source is named and the materials are distributed under the same license terms.

The content of the website is available as source code at <https://github.com/innoq/SCS> so that everyone can make changes. The website contains links to many articles about experiences with SCSs from different projects and companies.

Self-contained Systems are best practices that have proven their usefulness in various projects. Whereas microservices do not provide many rules about how systems should be built, SCSs have precise rules based on proven patterns. Thus, SCSs give a point of reference as to how a microservices architecture can look like.

Although SCSs are a collection of best practices, the SCS approach is not the best architecture for every situation. Therefore, it is important to understand the reasons for the SCS rules. By doing so, teams can choose variations of this approach or even completely different approaches that might be better adapted to the respective project.

## 6.2 Definition

Self-contained Systems make different macro architecture decisions.

- Each Self-contained System is an *autonomous web application*. Thus, SCSs contain a web UI.
- There is *no common UI*. An SCS can have HTML links to other SCSs or can integrate itself by different means into the UI of other SCSs. But every part of the UI belongs to an SCS. [Chapter 7](#) describes different options for the integration of frontends. This means each SCS generates a part of the UI. There is no separate microservice that generates all of the UI and then calls logic in the other microservices.
- The SCSs can have an *optional API*. This API can, for example, be useful if mobile clients or other systems have to use the logic in the SCS.
- The *complete logic* and *all data* for the domain are contained in the SCS. This is what SCSs were named for. An SCS is self-contained because it contains UI, logic, and data.

These rules ensure that an SCS completely implements a domain. This means that a new feature causes changes to only one SCS even if the logic, data, and also the UI need to be changed. These changes can be rolled out with a single deployment.

If several SCSs shared a UI, many changes would affect not only the SCSs but also the UI. Then two well-coordinated deployments and a closely coordinated development would be necessary.

### Rules for Communication

The communication between SCSs has to follow a number of rules.

- Integration at the *UI level* is ideal. The coupling is then very loose. The other SCS can display its UI as required. Even if the UI is changed, other SCSs are not affected. For example, if HTML links are used, the integrated SCS does not even have to be available. The link is displayed even if the system is unavailable; there will just be an error if the user clicks on the link. This helps with resilience, because the failure of an integrated SCS does not affect other SCSs.

- The next option is *asynchronous* communication. The advantage of this is that if the integrated SCS fails, the requests only take longer until the failed SCS is available again. However, the calling SCS will not fail because it has to be able to deal with longer latency times anyway.
- Finally, integration with *synchronous* communication is also possible. In such cases, precautions must be taken to deal with the potential failure and slow responses of the integrated SCSs. In addition, the response times add up if you have to wait for the responses of all synchronous services.

These rules focus on a very loose coupling between the SCSs to avoid error cascades in which, when one system fails, the dependent systems as a consequence also fail.

From these rules follows that an SCS replicates data. The SCS has its own database. Since it is primarily intended to communicate asynchronously with other SCSs, it is a challenge if the SCS must use data from another SCS to process a request. If the data is requested during the processing of the request, the communication is synchronous. For asynchronous communication, the data must be replicated beforehand so that it is available in the SCS when processing the request.

Consequently, SCSs are not always consistent. If a change to the dataset has not yet been passed on to all SCSs, then the SCSs have different datasets. This may not be acceptable in some situations. For particularly high-consistency requirements, the SCSs must use synchronous communication. In this scenario, one SCS receives the current state of the data from the other SCS.

## Rules for the Organization

Also for the organization, SCSs provide macro architecture rules. An SCS belongs to *one team*. The team does not necessarily have to make all changes to the code, but it must at least review, accept, or reject changes. This enables the team to direct and control the development of the SCS.

A team can handle several SCSs. However, it isn't a good idea for an SCS to be changed by more than one team on an equal footing.

Thus, Self-contained Systems use the strong architectural decoupling to achieve organizational advantages. The teams do not need to coordinate very much and can work in parallel on their tasks. Requirements can usually be implemented in one SCS by one team because an SCS implements a domain. Coordination in this respect is therefore hardly necessary. Because the technical decisions mostly concern only one SCS, there are hardly any arrangements necessary either.

For SCSs, as well as for microservices and other types of modules, the division into modules is aimed at enabling independent development. Having several teams change the same module is obviously not a very good idea. The modules allow independent development, but when several teams are working on one module, close coordination is still necessary. The advantage of modularization is not exploited then. This is the reason why equal joint development of an SCS by several teams is not allowed.

## Rule: Minimal Common Basis

Because Self-contained Systems aim at enabling a high degree of independence, the common basis should be minimal.

- Business logic must not be implemented in code used by more than one SCS. *Shared business logic* leads to a close coupling of the SCSs. That should be avoided. Otherwise, a change to one SCS could require changes to the shared code. Such changes must be coordinated with other users of the code. This creates a tight coupling that SCSs should avoid. In addition, common business code indicates a bad domain macro architecture. Business logic should be implemented in a single SCS. The business logic in an SCS can of course be called and used by another SCS via the optional interface of the SCS.
- *Common infrastructure* should be avoided. SCSs should not share a database, for example. Otherwise, the failure of the database would lead to a failure of all SCSs. However, a separate database for each SCS requires a considerable effort. For this reason, compromises are conceivable if the robustness of the system is not quite so important. The SCSs could have a separate schema in a common database. A shared schema would violate the rule that each SCS should have its own data.

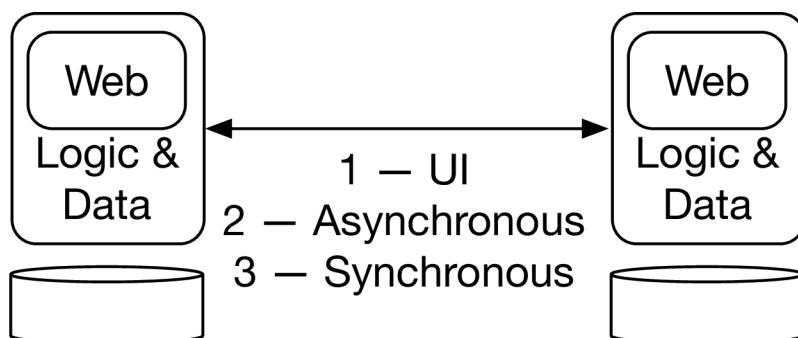


Fig. 6-1: SCS: Concept

[Figure 6-1](#) provides an overview of the most important features of the concept of Self-contained Systems.

- Each SCS contains its own *web UI*.
- In addition, the SCS contains the *data* and the *logic*.
- Integration is *prioritized*. UI integration has the highest priority, followed by asynchronous and finally synchronous integration.
- Each SCS ideally has its *own database* to avoid a common infrastructure.

## 6.3 An Example

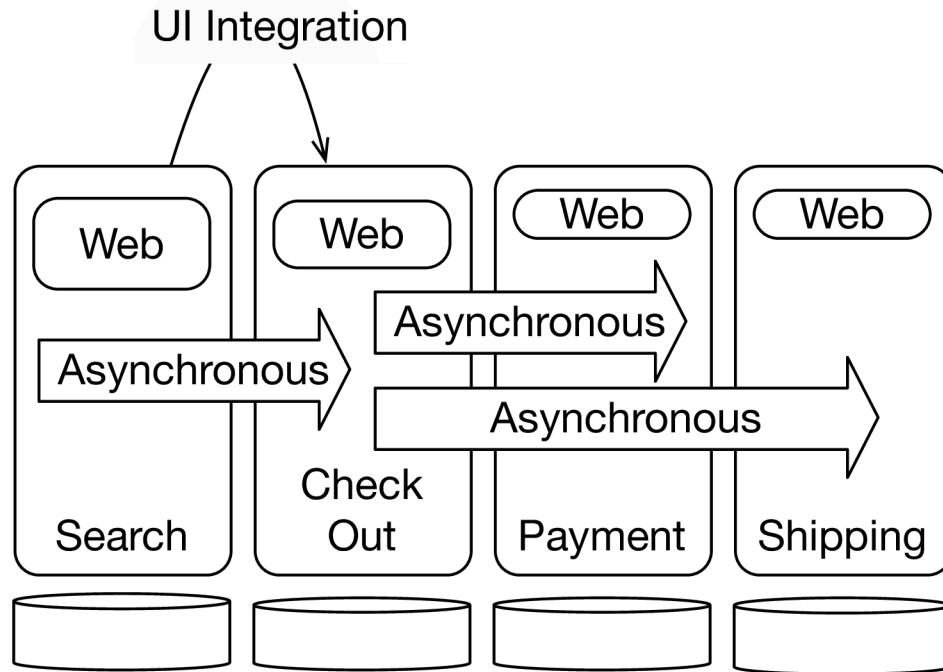


Fig. 6-2: Example of an SCS Architecture

Figure 6-2 demonstrates how the example from section 2.1 can be implemented with SCSs.

- One SCS implements the *search* for products.
- With *check out*, a shopping cart which was filled during searching is turned into an order.
- *Payment* ensures that the order is paid and provides information about the payment.
- *Shipping* sends the goods to the customer and offers the customer the possibility to obtain information regarding the state of the delivery.

Each SCS implements a bounded context with its own database, or, at least, one schema in a common database. In addition to the logic, each SCS also contains the web UI for the respective functionalities. An SCS does not necessarily have to implement a bounded context, but this achieves a high degree of independence of the domain logic.

### Communication

A user sends an HTTP request to the system. The HTTP request is then usually processed by a single SCS without another SCS being involved. This is possible because all the needed data is available in the SCS. This is good for performance and also for resilience. The failure of one SCS does not lead to the failure of another SCS, because the SCS does not call any other systems when

processing the requests. Slow calls of other SCSs via the network are thus also avoided, which promotes performance.

Of course, the SCSs must still communicate with each other. After all, they are part of an overall system. One reason for communication is the lifecycle of an order. The customer searches for products in *search*, orders them in the *check out*, pays them in *payment*, and then tracks the *shipping*. When moving from one step to the next, the information about the order must be exchanged between the systems. This can be done asynchronously. The information does not have to be available in the other SCSs until the next HTTP request; therefore temporary inconsistencies are acceptable.

In some places, close integration seems to be necessary. This means that *search*, for example, must display not only the search results, but also the contents of the shopping cart. However, the shopping cart is managed by *check out*. UI integration can be a solution to such challenges. In that case, the *check out* SCS can decide how the shopping cart is displayed, and the representation will be integrated in the other SCSs. Even changes to the logic rendering the shopping cart will be implemented only in one SCS, although many SCSs will display the shopping cart as part of their web pages.

## 6.4 SCSs and Microservices

SCSs stand out from deployment monoliths in the same way as microservices. A deployment monolith would implement the entire application in a single deployable artifact. SCSs divide the system into several independent web applications.

Because, an SCS is a separate web application, it can be deployed independently of the other SCSs. SCSs are also modules of an overall system. Therefore, SCS are independently deployable modules and consequently correspond to the definition of microservices.

However, an SCS may be split into several microservices. If the payment in the *check out* SCS of an e-commerce system causes a particularly high load, then it can be implemented as a separate microservice, which can be scaled separately from the rest of the *check out* SCS. So the *check out* SCS consists of a microservice for payment and contains the rest of the functionality in another microservice.

Another reason for separating a microservice from an SCS is the security advantage offered by greater isolation. In addition, domain services, which calculate the price of a product or quotation for all SCSs, can also be a useful shared microservice. The microservice should be assigned to a team like an SCS in order to avoid too much coordination.

In summary, microservices and SCSs differ in the following ways:

- Typically, microservices are *smaller* than SCSs. An SCS can be so large that an entire team is busy working on it. Microservices can potentially have only a few hundred lines of code.
- SCSs focus on a *loose coupling*. There is no such rule for microservices, although tightly coupled microservices have many disadvantages and therefore should be avoided.

- In any case, an SCS must have a *UI*. Many microservices offer only a technical interface for other microservices, but no user interface.
- SCSs recommend UI integration or asynchronous communication; synchronous communication is allowed, but not recommended. Large microservices systems such as Netflix also focus on *synchronous communication*.

## 6.5 Challenges

SCS describes an architectural approach which is more narrowly defined than, for example, microservices. Therefore, SCS cannot be the approach for solving all problems.

### Limitation to Web Applications

The first limitation of SCSs is that they are web applications. Thus, SCSs are no solution when no web UI is required.

However, some aspects of SCSs can still be implemented in a scenario that does not involve web applications: The clear separation of domains and the focus on asynchronous communication. If a system is to be developed that, for example, offers only an API, an architecture can be created that provides at least some of the advantages of SCSs.

### Single-Page App (SPA)

A single-page app (SPA) is usually an application written in JavaScript that runs in the browser. SPAs often implement complex UI logic. Applications such as Google Maps or Gmail are examples of applications that are highly complex and must be very interactive. SPAs are ideal for these cases.

However, SPAs also have disadvantages:

- Since logic can be implemented in an SPA, in practice *business logic often also leaks into the UI*. This makes further development more difficult, because logic is now implemented on the server and the client and thus at two different points in two, often different, programming languages.
- The *load times* of an SPA can be higher than the load times of a simple website. Not only HTML must be displayed, but also the JavaScript code must be loaded and started. Loading times are very important in some areas such as e-commerce, because the user behavior of customers depends on loading times.

With SCSs, these problems are complemented by further challenges such as:

- An *SPA for the entire system* would mean that there is a common UI. This is forbidden in SCSs.

- An *SPA per SCS* is one possibility for providing each SCS with its own UI. In this case, switching from one SCS to another means starting and loading a new SPA, which can take some time. Moreover, it is not so easy to split one SCS into multiple SCSs, because the SPA also needs to be split up. However, a further division can be important for the further development of the system in order to adapt the architecture.

Due to the high popularity of SPAs among developers, the contradictions between SPAs and SCSs in practice are a major reason for difficulties in implementing SCSs. An alternative is ROCA (see [section 7.3](#)). It establishes rules that stand for classic web applications and are much easier to use with the SCS idea.

## Mobile Applications

SCSs are not suitable as just a backend for mobile applications. The mobile application is a separate UI from the backends, so that UI and logic are separated and the concepts of SCS are violated. Of course, it is still possible to implement an SCS with a web UI that also provides a backend for a mobile application.

There are different alternatives:

- A *web application* can be developed instead of a mobile application. It can be responsive so that the layout adapts to desktop, tablet, or smartphone. Compared to a native app, the advantage is that there is no need to download and install an app from the app store. The number of apps that a typical mobile user has is quite low. Therefore, the installation of an app can be a hurdle. Thanks to HTML5, JavaScript applications can now use many of the mobile phone features. Websites such as <https://caniuse.com/> show which features are provided by which browser. When the decision is made in favor of a web interface, real SCSs can be implemented.
- A *web application* can be implemented with a framework such as [Cordova](#)<sup>68</sup> that takes advantage of the smartphone's specific features. There are other solutions based on Cordova. Therefore, you can still create a real SCS, but have the app be downloaded from the app store, use all features of the smartphone and behave like a native app. Of course, it is possible to implement parts of an app with such a framework and implement the rest as a truly native app.
- Finally, a *native app* can be created. Then, for example, the backends will offer a REST interface. If the backends do not also implement a web interface, they are not SCSs. Still, even in this case, it is of course possible to divide the logic into largely independent services and use asynchronous communication between the services in order to achieve many advantages of the SCS architecture.

## Look and Feel

Dividing a system into several web applications quickly raises the question of a uniform look and feel. [Section 2.2](#) has already shown that a uniform look and feel can only be achieved with a macro architecture decision. This also applies to SCSs, of course.

---

<sup>68</sup><https://cordova.apache.org/>

## 6.6 Benefits

SCS are a logical extension of the idea of bounded context presented in [section 2.1](#). A bounded context contains the logic on a specific part of the domain. SCSs make sure that also the UI and the persistence are part of one single microservice. So the split by domain that domain-driven design advocates is further improved by SCS.

Ideally, one feature should lead to one change. In an SCS system, the chance that this actually happens is quite high: The change will likely be in one domain. Then it can be contained in one SCS even if the UI and the persistence need to change. And the change can be put into production with one deployment. Therefore, SCSs provide a better changeability.

Also, testing is easier because all required code for the domain are in one domain. So it is easier to do meaningful tests of the logic together with the UI.

Due to the focus on the UI, SCSs make it easier to implement integration in the UI. So SCSs open up additional integration options. This is particularly useful for heterogeneous UI technology stacks. With the rate of innovation in UI technologies, it is unrealistic to assume a uniform UI technology stack.

## 6.7 Variations

Instead of implementing an SCS with UI, logic, and data there are different variations. These architectures are no SCS architectures. However, they might still be sensible in certain contexts.

- Domain microservices with logic and data but *without UI* can make sense when an API or a backend are to be implemented.
- Microservices without logic or data, which consequently implement *only a UI*, can be a good approach for implementing a portal or a different kind of frontend.

Both of these variations are sensible when the project is a pure frontend or backend and does not allow the full implementation of an SCS.

### Typical Changes

A good architecture should limit a change to one microservice. The basic assumption of the SCS architecture is that a change typically goes through all layers and is limited to a domain implemented in one microservice. This assumption has been confirmed in many projects. Still, if a change in a project usually affects only the UI or only the logic, it may be better to implement a division by layers. Then a new look and feel or new colors can be implemented in the UI layer, while logic changes can be implemented in the logic layer – of course, only if they do not affect the UI.

## Possibilities for Combinations

Self-contained Systems are easy to combine with other recipes.

- Frontend integration (see [chapter 7](#)) is preferred when integrating SCSs.
- SCSs focus on asynchronous communication ([chapter 10](#)). SCSs are web applications so that the use of Atom ([chapter 12](#)) for asynchronous REST communication is especially easy because it also builds on HTTP. Kafka ([chapter 11](#)), on the other hand, has a different technical foundation. Even though it can still be combined with SCSs, it then represents an additional technical effort.
- Synchronous communication ([chapter 13](#)), even though possible, should be avoided.

Thus, UI integration, asynchronous communication and synchronous communication between SCSs are possible, but a clear prioritization exists. Of course, SCSs can also communicate with microservices or other systems via these mechanisms.

## 6.8 Conclusion

Self-contained Systems represent an approach for the implementation of microservices that has proven its usefulness in numerous projects. They provide best practices on how to implement successful architectures with microservices. However, they also restrict the general approaches of microservices and are therefore more specialized. Nevertheless, at least aspects such as asynchronous communication and division into different domains are a helpful approach in many situations.

# 7 Concept: Frontend Integration

This chapter explains how microservices can be integrated in the web frontend.

- First, this chapter discusses *why* the web frontend of a microservices architecture should be modularized.
- *Single-page apps (SPAs)* are difficult to modularize.
- *ROCA (resource-oriented client architecture)* is an approach for web UIs which supports modularization.
- Furthermore, this chapter covers the available *options* for integration.
- Finally, *benefits* and *disadvantages* of the different integration options are discussed.

Thus, this chapter demonstrates how frontend integration can be implemented and for which scenarios this is the best approach.

## 7.1 Frontend: Monolith or Modular?

Figure 7-1 shows a frontend monolith which serves as frontend for multiple microservices.

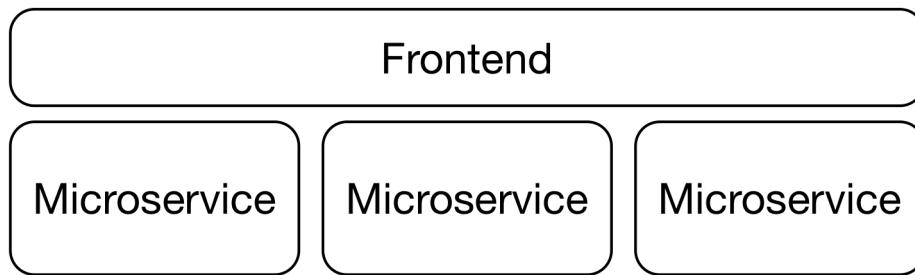


Fig. 7-1: Frontend Monolith with Microservices Backend

### Option: Monolithic Frontend and Backend

Doing without modularization in the frontend is inconsistent. On the one hand, a backend is modularized into microservices; this involves a great deal of effort, for example, for operation. On the other side is a monolithic frontend. Such an architecture must be questioned. The result does not necessarily have to be a modularized frontend. It may also turn out that a monolithic frontend together with a monolithic backend also meets the requirements. Then you can save the effort for the modularization of the backend into microservices. This can be the case, for example, if the advantage of separate deployment and further decoupling in other areas (see section 1.2) is not so important after all.

Figure 7-2 shows a frontend monolith which serves as frontend for a monolithic backend.

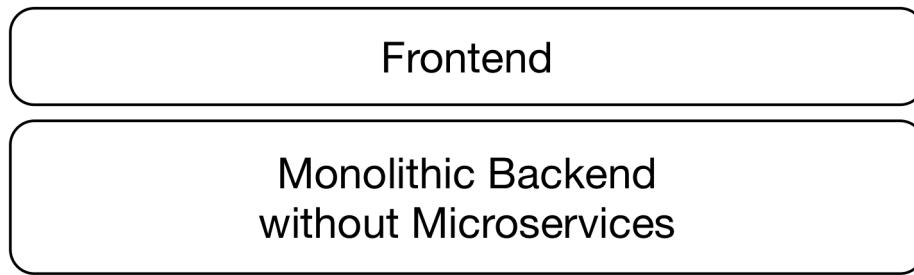


Fig. 7-2: Frontend Monolith with Backend Monolith

### Option: Modularly Developed Frontend

Of course, a frontend deployed as a monolith can be developed modularly. Unfortunately, experience shows that modular development often still leads to an unmaintainable, non-modular system in the end, because the boundaries between the modules dissolve over time. The boundaries between modules that can be deployed separately, such as microservices, are not so easy to circumvent so that modularization is secured in the long term.

Nevertheless, each microservice can, for example, be assigned a module in the frontend in order to decouple and parallelize the development. However, in that case, the frontend monolith still has to be deployed as a whole. Separate deployment is the advantage of microservices over other modularizations.

Figure 7-3 shows a frontend monolith that is divided into modules.

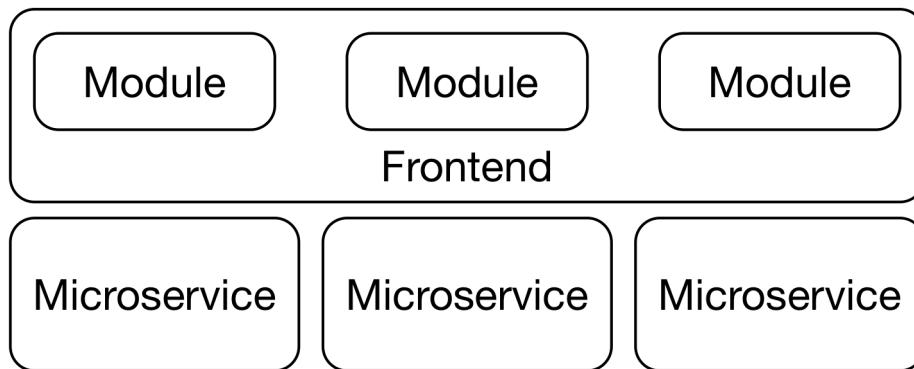


Fig. 7-3: Frontend Monolith Divided into Modules

### Reasons for a Frontend Monolith

A frontend monolith can be a good choice in some circumstances.

- *Native mobile applications* or *rich client applications* are always deployment monoliths. They can be delivered only as a whole. Mobile applications even need to pass a review process in the

app store in case of an update, so deployment takes even longer. However, this mechanism can be “undermined” to a certain extent. An app can display web pages. These can be provided by a microservices system that uses web frontend integration. Frameworks such as [Cordova](#)<sup>69</sup> even allow web applications to take advantage of proprietary mobile phone features or to offer a web application for download from the app store. This means that compromises can be implemented between native applications and web applications.

- *Single-page apps (SPAs)* can also be deployed only as a whole. There are enough alternatives to SPAs for web applications to completely modularize an application in the frontend. The boundaries are fluid. SPAs can contain links to other sites or other SPAs and display HTML generated by a different system. In this way, SPAs can be integrated in the frontend. But in practice, SPAs typically lead to a frontend deployment monolith despite these theoretical possibilities.
- If there is *a team* that deals with frontend development, this can be a reason to develop a monolithic frontend. Each team should be responsible for one component. If you do not want to break up the frontend team because it is too big an organizational change, or because the team is working in a different location than the other teams, a monolithic frontend can be the best approach.
- Finally, the *migration* of an existing system might be particularly easy if the monolithic frontend remains intact.

## Modularized Frontend

The alternative to a monolithic frontend is a fully modularized frontend. [Figure 7-4](#) shows a modularized frontend, where each microservice has its own frontend. Like the backend, the frontend is part of the separately deployable microservices. Such a modularization of the frontend has many advantages. [Section 2.1](#) has shown that microservices and Self-contained Systems can be independent concerning their domain logic. If the microservices contain a part of the modularized frontend, then a change in a domain can be implemented by modifying and deploying just one microservice, even if changes are necessary at the frontend. If the UI is a monolith, on the other hand, many changes to the domain logic require modifications to the UI monolith so that the UI monolith becomes a main focus of change.

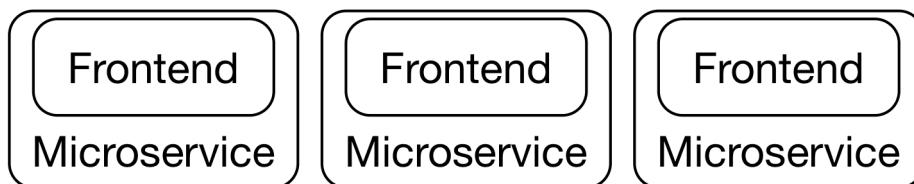


Fig. 7-4: Modularized Frontend

## Modularized Frontend and Frontend Integration

To combine the separately deployed frontends into a complete system, the frontends must be integrated. Modularization is intended to decouple development. Nevertheless, an integrated system

<sup>69</sup><https://cordova.apache.org/>

must be created. In other words, a frontend modularized into different microservices is only possible if an approach for frontend integration has been chosen. For this, different technical approaches are possible, which are the focus of this and the following chapters.

## 7.2 Options

There are different options for frontend integration.

- The easiest option are *links*. One frontend displays a link that another frontend handles. The World Wide Web (WWW) is based on precisely this mechanism. A system creates a link to another system.
- *Redirects* represent another option. For example, OAuth2 uses this approach. A website provides a link to an OAuth2 provider such as Facebook or Google. The user enters his or her password and confirms that the website is allowed to access certain information. The user is then redirected back to the original website by another redirect. Behind the scenes, the website receives the user's data. Redirects can therefore combine frontend integration with data transfer in the background.
- Finally, there are various kinds of *transclusion*. This involves combining the content of a website with the content of another website. This can be done either on the server or on the client. Chapter 8 shows an example where transclusion is implemented on the client with JavaScript. Chapter 9, on the other hand, shows transclusion on the server side with ESI (Edge Side Includes). The blog article at <https://www.innoq.com/en/blog/transclusion/> gives an overview of further possibilities.

These options can, of course, all be combined with each other. However, this leads to a high level of technical complexity. Therefore, you should first try to get by with just links because they have a very low complexity, and only add more options if needed.

## 7.3 Resource-oriented Client Architecture (ROCA)

Modularization and integration in the frontend have an impact on the architecture and the technologies in the frontend. SPAs (single-page apps) are not very well suited for integration in the frontend. Therefore, the question arises as to which frontend architecture is better suited for integration.

ROCA ([Resource-oriented Client Architecture<sup>70</sup>](#)) is an approach for implementing web applications. It focuses on established technologies such as HTML, and leads to an architecture which comprises many benefits for frontend modularization and integration.

---

<sup>70</sup><http://roca-style.org/>

## ROCA Principles

ROCA has the following principles:

- The server adheres to the *REST* principles. All resources have an unambiguous URL. Links to web pages can be sent by e-mail and then accessed from any browser if the necessary authorizations are given. HTTP methods are used correctly. For example, GETs do not change data. The server is stateless.
- Resources identified by URLs can have *other representations* besides HTML, such as JSON or XML. This means that the data is not only available to people, but applications can also access the information.
- All *logic* is on the *server*. Accordingly, JavaScript on the client only serves to optimize the user interface. Not just browser but also other clients should be able to access the system. Logic on the server is desirable for security reasons because the client could be manipulated. Changing the logic is also easier because it is implemented in one location making it unnecessary to update a large number of clients.
- The *authentication information* is included in the *HTTP request*. HTTP basic, digest, client certificates, or cookies can be used for this purpose. In this way, authentication and authorization can take place solely on the basis of information contained in the HTTP request. Therefore, no server-side session is required for the authentication information.
- *Cookies* may be used only for authentication, tracking, or as protection against cross site request forgery. Therefore, they may not contain business information.
- There must *not* be any *server-side session*. The use of sessions contradicts the ideas of HTTP because the communication is then no longer stateless. This condition makes it difficult to implement failover and load balancing. The only exception is data required for authentication purposes in addition to the authentication information in the HTTP request.
- The *browser controls* such as the back, forward, or refresh button should work. This should be a matter of course, but many web applications with JavaScript logic have difficulties with it, or have to do a lot of work in order to ensure the functioning of these buttons.
- The *HTML* may *not* contain *layout information* and should be accessible via a screen reader. The layout is defined by CSS so that layout and content are separate.
- JavaScript can be used, however, only in the form of *progressive enhancement*. The application can even be used without JavaScript – just not as easily and comfortably. The goal is not to avoid JavaScript completely, but to still rely on the fundamental architecture and technologies of the web – HTTP, HTML and CSS.
- *Logic* must *not* be implemented *redundantly* on client and server. Because the business logic is implemented on the server, it thus must not be implemented again on the client.

In the end, ROCA applications are perfectly normal web applications. They use the web principles as they were originally intended.

## Benefits of the ROCA Architecture

ROCA has a number of advantages:

- The applications have a *clean architecture*. The logic is on the server. Changes to the logic can easily be rolled out by a new server version.
- All *features of the web* can also be used. URLs can be sent to other people – for example, by e-mail – because they uniquely identify resources. HTTP caches can be used because, for example, HTTP GETs are not allowed to change data. Optimizations in the browsers are exploited. Browsers implement many tricks to show users the first parts of a website and to allow interactions as quickly as possible.
- As a result of usually having to transfer no more than HTML – and only for the web pages that are actually visited – the applications get by with *little bandwidth*. In an SPA, often the entire application must be transferred before any interaction is possible at all. Modern SPA technologies optimize this by loading separate modules rather than all the code. However, initializing the application and making it react to user interaction still takes some time. This is typically easier with a simple web application. Modern browsers are optimized to make user interaction with simple web applications as fast and responsive as possible.
- In addition, the solution is *fast*. Especially for mobile devices, the speed of JavaScript implementations often leaves much to be desired. ROCA applications require a minimum of JavaScript.
- Finally, an *error in JavaScript* or in the transfer of the JavaScript code due to a problem on the network only leads to hard-to-use, though still available application. If logic were implemented in JavaScript, this would not be the case and the application would have less resilience.
- Users having *JavaScript switched off* can still use the application. Nowadays, however, such users are practically non-existent so that this advantage is irrelevant.

## ROCA versus SPAs

ROCA is an alternative to SPAs (single-page apps) where the browser is used only to execute JavaScript code. ROCA takes advantage of the browser's optimizations for HTML and HTTP, and also of the other advantages of a real web application. Of course, a ROCA system can also use JavaScript as discussed previously. The goal of ROCA is not to avoid JavaScript completely, but to limit it to areas where it makes sense or where there are no alternatives.

SPAs might be a better fit for complex UI – for example, games or map applications on the web. However, for other types of applications, such as e-commerce applications, SPAs not only provide unneeded flexibility, but, in fact, make it harder to achieve some features easily, like search engine optimizations (SEO). So for a lot of applications, SPAs are actually not that great a fit.

SPAs seduce developers to implement more logic on the client. Although this might make the system more responsive, it can also lead to redundant implementation on the client and server that is harder to maintain. Also, it is hard to export the logic in the SPA as a REST service so that other clients can use it.

[Chapter 8](#) shows an example of a ROCA application that is quite comfortable to use.

## Integration Options

ROCA applies to good UI layer design generally. It makes sense as a guideline for implementing browser applications, monolithic or modularized. However, ROCA does facilitate a simplified integration. HTML is the focus in the frontend. JavaScript serves only to improve usability. Thus, the system can be modularized by providing HTML pages with links to other HTML pages that may originate from other microservices. Or, the HTML pages can even be composed of several parts. Each component can come from a different microservice, making it easy to modularize a ROCA UI. Because ROCA supports all options for frontend integration, ROCA is a good basis for frontend integration and for a microservices architecture.

## 7.4 Challenges

Frontend integration means that the frontend is composed of different systems. This causes some challenges.

### UI Infrastructure

For UI integration to work, some infrastructure has to be provided. That might just be common CSS or some JavaScript code, but it can also include server infrastructure for server-side transclusion. Also, there might be a need for UI parts that do not really belong to any specific microservices – for example, the home page or a navigation bar. These have to be developed and maintained.

It is important not to put too much into the UI infrastructure. If the microservice relies too much on the generic UI infrastructure, it will depend heavily on it and that contradicts the goal of independent development. UI integration leads to a certain degree of dependencies on the code level, which usually leads to tight coupling; therefore, the dependencies should be limited. For example, it makes little sense to provide all the styling, UI frameworks, and UI code for all microservices in one single component. This would be a problem in particular for a migration to new technology stacks. The stack is the same for all microservices, and it is therefore hard to migrate stepwise to a new stack.

### Uniform Look and Feel

For example, it is not quite so easy to achieve a uniform look and feel and design of the overall application. A uniform look and feel is usually associated with the sharing of artifacts. Multiple microservices must share the CSS, fonts, or JavaScript code required to implement the design. [Section 2.2](#) has already shown a solution based on macro architecture.

### Interfaces in Frontend Integration

Transclusion generates a subtle type of interface definition. Normally, an interface is defined by data types and operations. This is obviously not the case for frontend integration. Still, there is a

kind of interface definition. For example, HTML code has to integrate itself into another page if it is displayed in another frontend. To do this, it may be necessary to have common CSS classes, and so all the frontends must possess the same CSS selectors. If JavaScript is used in the displayed HTML, the common JavaScript code and the JavaScript libraries used must be available in the other web pages. Overall, these requirements form a kind of interface definition that ensures that HTML can actually be displayed. When only links or redirects are used, this challenge does not exist; only the URL must be known. The linked page can use completely different CSS and JavaScript. Transclusion therefore couples the systems more strongly than links do.

## UI Changes Impact Multiple Modules

If the changes to a system typically only require changes to the UI, this can be more complicated with frontend modularization. The code for the UI is distributed across the different frontends, so that a change means that all frontends have to be modified. Therefore, if the UI is constantly redesigned or if the CSS is constantly being changed, then frontend modularization can increase the effort. However, frontend modularization has an advantage, even in case of such changes. The change can be made step by step by changing only one frontend at a time. This can minimize the risk associated with changes.

In addition, changes that take place only in the UI should be far less frequent than domain-based changes that affect all layers. So, whereas it is harder to change all of the UI, those kinds of changes should be not too frequent. Making them harder and making more frequent changes easier at the same time seems like a good trade off.

## 7.5 Benefits

Frontend integration offers a number of benefits which make the approach attractive.

### Loose Coupling

Integration in the frontend creates loose coupling. For example, if links are used for integration, only a URL must be known to integration. What is behind the URL and how the information is displayed doesn't matter and can be changed without impacting other frontends. So a change can be limited to one frontend, even if the page looks completely different.

### Logic and UI in One Microservice

This is advantageous from the architecture's point of view. All logic for a certain functionality is implemented in a single microservice. For example, a microservice can be responsible for maintaining and displaying a to-do list, even if the list is displayed integrated in the UI of another microservice. If you want to display additional information in the to-do list, such as a priority, you can implement the logic, persistence, and UI by changing only one microservice, even if another microservice integrates the rendered to-do list.

## Free Choice of Frontend Technologies

Another influencing factor for frontend integration are the frontend technologies. Especially in frontend technologies, there are many innovations. There are constantly new JavaScript frameworks and new ways to create beautiful and easy-to-use interfaces. An important advantage of microservices is the freedom of technology. Each microservice can choose its own technologies. If technology freedom should also apply to the frontend, then each microservice must have its own frontend that can use its own technology. For this, the frontends must be integrated accordingly. In particular, care must be taken to ensure that integration does not restrict the use of frontend technologies. For example, if the integration forces a certain JavaScript library, this can limit the technology selection because, with JavaScript, it is not possible to use another version of this library in parallel. For example, the client-side integration in [chapter 8](#) requires that each frontend uses a certain jQuery version or provides its own custom code.

## 7.6 Variations

Self-contained Systems (SCS) (see [chapter 3](#)) particularly focus on frontend integration.

The next chapters discuss the different variations for frontend integration. [Chapter 8](#) shows how links and transclusion work on the client side. [Chapter 9](#) demonstrates the integration on the server side with ESI (Edge Side Includes).

Typically, a microservices system also uses synchronous communication ([chapter 13](#)) or asynchronous communication ([chapter 10](#)) in addition to frontend integration. The use of several integration approaches is common and easily possible.

Ultimately, the browser only needs to access the various backends via HTTP. This means that frontend integration puts few constraints on the technologies used.

## 7.7 Conclusion

In general, integration at the frontend level should always be considered when possible. Too often, microservices are implemented only as an approach to the backend, although by focusing on the frontend, the coupling can be looser, and the system can be simpler and more flexible. It can also be ensured that the entire logic, including the logic for the UI, is actually implemented in the microservice.

Frontend integration allows loose coupling of microservices. Already the use of links and of some JavaScript code can be enough to integrate the frontends of different microservices. It is therefore important not to immediately define a complex technology stack, but to first find out what can be achieved by simple means. This leads to another advantage: The technical complexity of the solution is not particularly high; only web applications are used. There are more of them than usual, but there are no new technologies.

But even when using server-side integration, the integration in the frontend is still not particularly complex and still leads to a loose coupling.

# 8 Recipe: Links and Client-side Integration

This chapter provides an example for a frontend integration with links and JavaScript.

The reader learns in this chapter:

- For which scenarios a simple approach based on links and JavaScript for frontend integration is feasible.
- The example in this chapter is not a Self-contained System (SCS). This shows that frontend integration does not only make sense in the context of SCS, but also in other scenarios.
- The example shows how an integration with links and JavaScript can be implemented.
- With which mechanisms a uniform look and feel can be implemented with such an integration.

## 8.1 Overview

The example in this chapter is a classic assurance application designed to help office workers interact with customers. It was created as a prototype for a real insurance company. The example shows how such a rather classic application can be implemented as a web application with frontend integration.

The example shows the division of a system into several web applications and the integration of these applications. ROCA is used as basis for this. This is an approach for implementing web applications (see [section 7.3](#)), which has a number of fundamental advantages especially for frontend integration.

The example was created as a prototype showing how a web application can be implemented with ROCA and what advantages it offers. The INNOQ employees Lucas Dohmen and Marc Jansing implemented the example.

### Search

The assurance application is available at <https://crimson-portal.herokuapp.com/>. It can also be run on a local computer as a Docker container (see [section 8.2](#)).

The application is in German. However, nowadays browsers can translate web pages into other languages. The screenshots were done with the Firefox browser and Google Translator's translation from German to English.

An input line appears on the main page there to search for customers (see [figure 8-1](#)). While the user enters the customer's name, matching names are suggested. For this, the frontend uses jQuery.

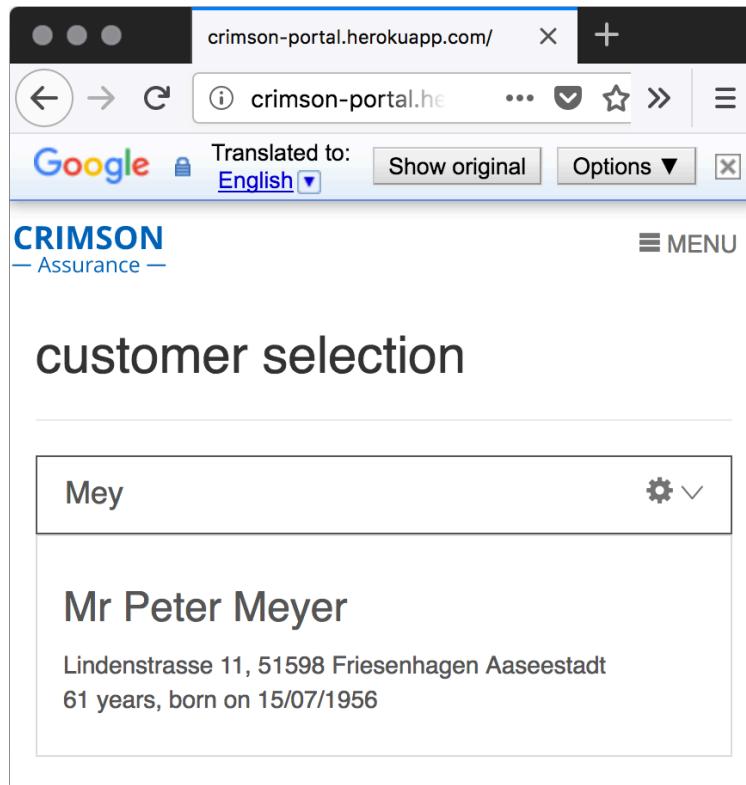


Fig. 8-1: Screenshot Insurance Application

## Postbox

Another functionality is the *postbox*. With a click on the postbox icon on the main page, the user gets an overview of the current news. The overview is displayed in the current main page with JavaScript (see [Figure 8-2](#)).

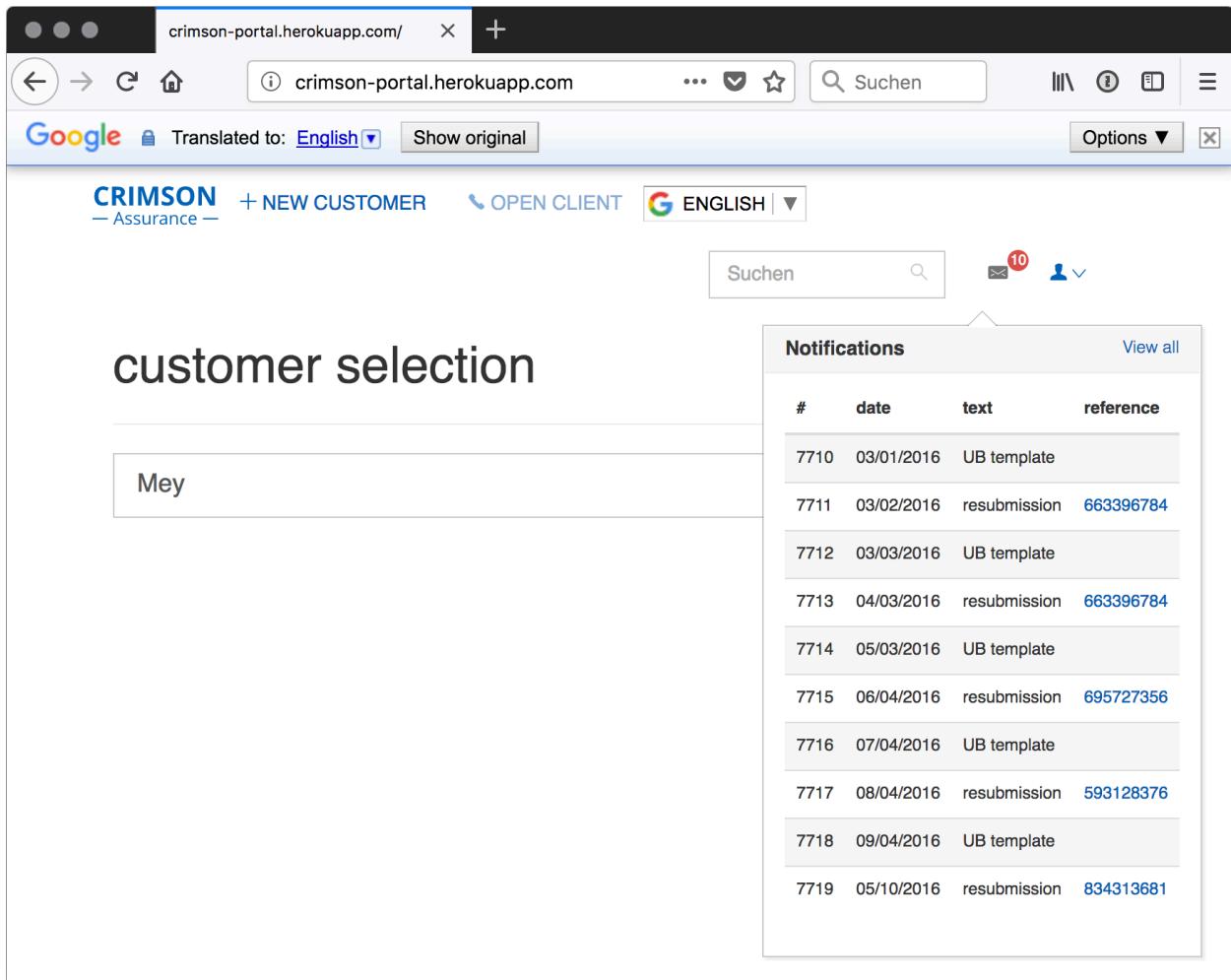


Fig. 8-2: Overlaid Postbox

The entire application uses a uniform frontend. However, if you look at the address line, you will notice that several web applications are used. There is one web application each for the main application (<https://crimson-portal.herokuapp.com/>), for reporting damages (*damage* application) (<https://crimson-damage.herokuapp.com/>), for writing letters (*letter* application) (<https://crimson-letter.herokuapp.com/>) and for the postbox (*postbox* application) (<https://crimson-postbox.herokuapp.com/>). Nevertheless, the frontend has the same look and feel for all these applications.

## Structure of the Application

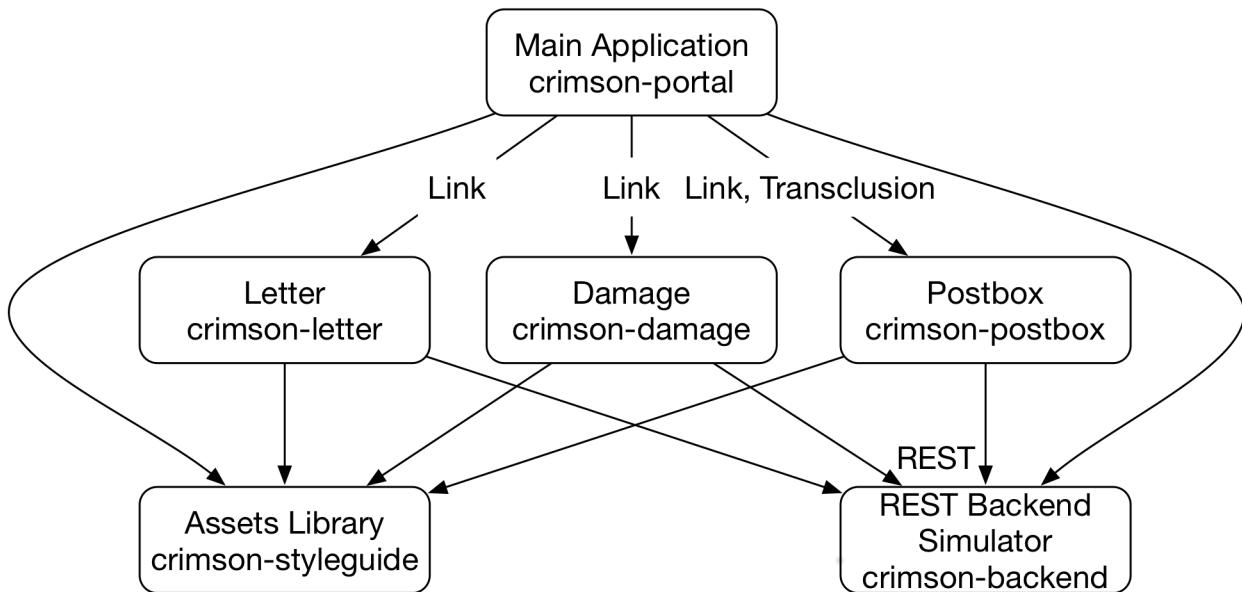


Fig. 8-3: Overview of the Insurance Example

Figure 8-3 shows how the different applications are integrated.

- The *main* application is used to search for customers and to display the basic data of a customer. The code is available at <https://github.com/ewolff/crimson-portal>. There you will also find instructions on how to compile and start the application. The application is written in Node.js.
- The *assets* at <https://github.com/ewolff/crimson-styleguide> contain artifacts used by all applications to achieve a consistent look and feel. The other projects refer to this project in the package.json. This allows the npm build to use the artifacts from this project. npm is a build tool specialized in JavaScript. The asset project includes CSS, fonts, images, and JavaScript code. Before the assets are used in the other projects, the build optimizes them in the asset project. For example, the JavaScript code is minified.
- The *damage* application for reporting a damage is also written in Node.js. The code can be found at <https://github.com/ewolff/crimson-damage>.
- The *letter* application is written in Node.js as well. The code is available at <https://github.com/ewolff/crimson-letter>.
- The code for the *postbox* can be accessed at <https://github.com/ewolff/crimson-postbox>. The postbox is implemented with Java and Spring Boot. To be able to use the shared asset project, the build is divided into two parts. The Maven build compiles the Java code, whereas npm is responsible for integrating the assets. npm then copies the assets into the Maven build.
- Finally, the *backend simulator* can be found at <https://github.com/ewolff/crimson-backend>. It receives REST calls and returns data regarding customers, contracts, and so on. This simulator is also written in Node.js.

## Why Monolithic Backend?

This application uses a monolithic backend and frontend microservices. Because the microservices lack logic, they are not Self-contained Systems (see [chapter 6](#)). However, this architecture can still make sense. The frontend, at least, consists of microservices, and so independent development of the microservices is possible. Also, it is possible to use different technologies in each frontend microservice. With the large number of available UI frameworks and the high speed of innovation, that is a clear advantage.

Also, it is probably not possible to migrate the backend into microservices. Another team might be responsible for it. Therefore, if the scope of the project is just to improve the frontend, there is just no way to change the architecture of the backend.

While SCS are generally a great idea, this example shows one exception from the rule.

## Integration with Redirects

If a user in the *damage* application enters a claim for a car, the user is sent back to the overview of the car displayed on the portal. The transition from the *damage* application to the portal is implemented with a redirect. The *damage* application sends an HTTP redirect after reporting the claim, leading to the web page of the *main* application.

A redirect is a very simple integration. The *damage* application needs to know only the URL for the redirect. The portal could even pass this URL to the *damage* application to further decouple the two applications.

Such an integration is also used if a user registers with his or her Google account on a web page. After the user agrees to register on the Google web page, the Google page sends a redirect back to the initial web page.

## Integration with Links

The integration of the applications is mainly done via links such as <https://crimson-letter.herokuapp.com/template?contractId=996315077&partnerId=4711> to display a web page for writing a letter.

They contain all the essential information necessary for the web page to write the letter: The contract number and the partner number. In this way, the *letter* application can retrieve the data from the backend simulator and render it in the web page. The coupling between the main application and the letter application is very loose; it's just a link with two parameters. The main application does not need to know what is behind the link. As a result, the *letter* application can change its UI at any time without any impact on the *portal* application. However, all applications use a common database from the backend simulator and are consequently tightly coupled, because a change to the data would affect the backend and the respective microservice.

## Integration with JavaScript

In the example, the integration of the frontends is practically always done via links. However, the *postbox* displays an overview of the current messages in the *main* application. For this, a simple link is not enough. Still, this integration is also a link. A look into the HTML code <sup>71</sup> shows:

```
<a href="https://crimson-postbox.herokuapp.com/m50000/messages"
    class="preview" data-preview="enabled"
    data-preview-title="Notifications"
    data-preview-selector="table.messages-overview"
    data-preview-error-msg="Postbox unreachable!"
    data-preview-count="tbody>tr" data-preview-window>
```

The link contains additional attributes. They ensure that the information of the *postbox* is displayed in the current web page and define how exactly this happens. This information is interpreted by less than 60 lines of JavaScript code from the asset project (see <https://github.com/ewolff/crimson-styleguide/tree/master/components/preview>). The code uses jQuery, so every application in the system now needs to use a compatible version of jQuery when using this code from the asset project. However, this is not mandatory. Alternatively, any microservice that integrates the *postbox* with such a link can write its own code to interpret the link. After all, every microservice writes its own code to read data from JSON that other microservices provide, for example. Therefore, code that integrates other projects' HTML should be fine, too.

This type of integration is called *transclusion* because it includes HTML from more than one microservice in a web page. In this example, the transclusion is implemented with JavaScript code that integrates the different backends (see figure 8-4). The JavaScript code runs in the browser, loads HTML fragments of the other web applications, and displays them in the current web page.

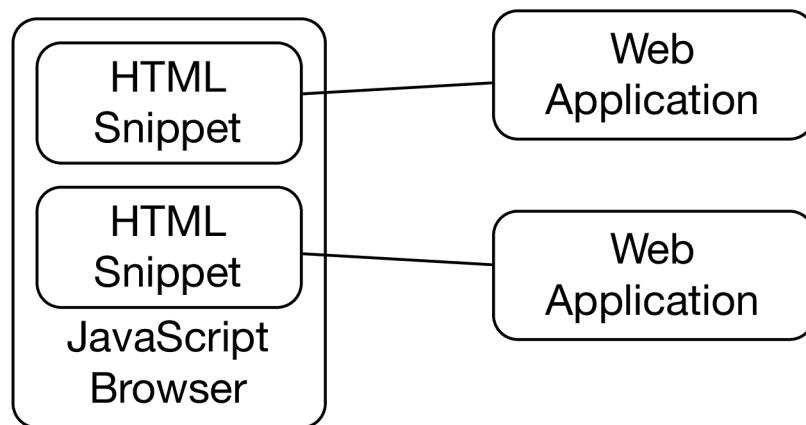


Fig. 8-4: Integration with JavaScript

<sup>71</sup>Actually, when running the application the HTML code contains German expressions. They can be translated by Google Translate. However, the HTML code remains unchanged and still includes German expressions. For convenience, the English expressions are shown in the HTML code in the listing here.

## Presentation Logic in the Postbox

With transclusion, *the postbox* retains control over how messages are displayed, even if the messages are shown as previews in another service. This leads to a clean architecture. The code for displaying the postbox is located in the postbox service, even if the postbox is displayed in another service. This shows how frontend integration can contribute to an elegant solution and architecture.

The approach of dynamically including content from other URLs is not only used for the *postbox*. For each customer there is also an overview of the offers, applications, claims, and contracts. The links for this are located below the postbox icon, and are repeated further down in the inventory. Just as with the postbox, this information is also referenced with links whose contents JavaScript code displays on the web page. In this case, the URLs are in the same microservice, but still provide a better modularization. It also shows that the code solves a general technical problem and is reusable beyond the integration between microservices.

## Assets with Integrated HTML

Transclusion embeds HTML fragments into other web pages. The HTML can require assets such as CSS or JavaScript. There are different approaches for ensuring that these assets are present.

- The HTML can be designed to not require any assets at all. Thus, no assets have to be shared between the microservices.
- The HTML uses only the assets from the shared asset library – in the example, `crimson-styleguide`. Then no special measures are necessary because the assets are present in all microservices.
- The HTML can also bring along its own assets or link to them. However, in this case you have to be careful not to load the assets more than once if several contents are transcluded in one web page.

Till Schulte-Coerne has written a [blog post<sup>72</sup>](#) about this topic.

## Resilience

The application achieves a high degree of resilience and reliability through the consistent use of links. If one microservice fails, the other microservices continue to work; they can still display the links. The JavaScript code contacts the *postbox* to transclude an overview of the messages in the web page. If this doesn't work, the code displays an exclamation mark, – but the application still works.

Thanks to JavaScript, loading the transcluded HTML is carried out in the background, so that the failure of one system does not affect the transclusion of the other contents and high performance is achieved.

---

<sup>72</sup><https://www.innoq.com/en/blog/transclusion/>

## With and Without JavaScript

The user can also use the applications if JavaScript is disabled. In this case, for example, the automatic completion of customer names does not work, nor does displaying the overview of messages in the *postbox*. Nevertheless, the application remains usable. The start page is rendered completely as HTML on the server and does not require client-side templates. The *postbox* simply offers a link instead of the displayed overview. If the user clicks on this link, he or she gets to the *postbox*.

## 8.2 Example

[Section 0.4](#) describes which software has to be installed to start the example.

This example is not only provided in the Heroku cloud, but also as a collection of [Docker containers](#)<sup>73</sup> so that you can run it on a local computer. To do so, you first have to download the code with `git clone https://github.com/ewolff/crimson-assurance-demo`. Then you need to change into the newly created directory `crimson-assurance-demo` with `cd crimson-assurance-demo`. `docker-compose up -d` generates all necessary Docker images with the help of Docker Compose (see [section 4.6](#)) and starts them. This takes quite a while because all the Docker containers are built, and also all the dependencies are downloaded from the Internet. If the Docker containers do not run on `localhost`, the hostname of the server has to be assigned to the environment variable `CRIMSON_SERVER`. This is necessary to make the links work. See [appendix C](#) for more details on Docker and Docker Compose and how to troubleshoot them.

A more detailed description of how the example can be started is available at <https://github.com/ewolff/crimson-assurance-demo/blob/master/HOW-TO-RUN.md>.

## Network Ports

The application is available at port 3000 on the Docker host – that is, at <http://localhost:3000>. *Postbox* has the port 3001, *letter* the port 3002, and *damage* the port 3003 (see [figure 8-5](#)). The frontend services communicate with the backend, which runs in a separate Docker container.

---

<sup>73</sup><https://github.com/ewolff/crimson-assurance-demo>

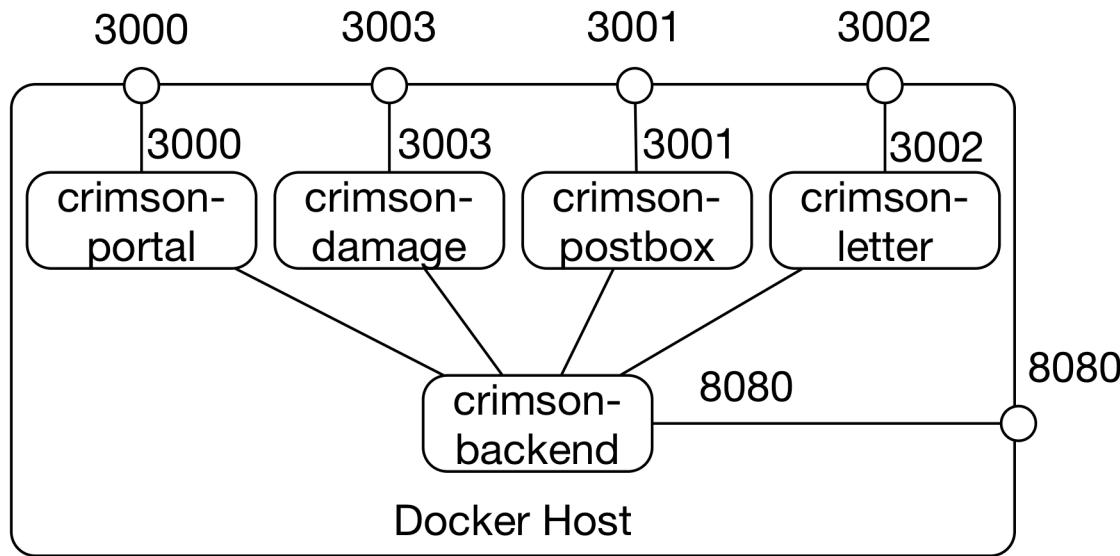


Fig. 8-5: Docker Containers in the Example

Of course, the ports of the Docker containers could be redirected to any other ports of the Docker host. Likewise, all applications in the Docker containers can use the same ports in each Docker container. However, to save confusion, in this example the port numbers of the containers are identical to those of the hosts .

You can also try the system directly on the web at [Heroku<sup>74</sup>](#). The links then point to the separate applications for *postbox*, *letter*, and *damage* also deployed at Heroku. Heroku is a PaaS (Platform as a Service, see [chapter 18](#)) available in the public cloud.

## 8.3 Variations

The example uses a Node project for the shared assets. An alternative option is an asset server which stores the assets. Since the assets are static files loaded via HTTP, an asset server can just be a simple web server.

Over time, the assets will change. For the asset project from the example, a new version of the asset project would have to be created. The new version must be integrated in every microservice. This sounds like an unnecessary work, but this way each application can be tested with a new version of the assets.

Therefore, even with an asset server, a new version of the assets should not simply be put into production, but the applications should be adjusted to the new version and also tested with the assets. The version of the assets can be included in the URL path. Thus, version 3.3.7 of Bootstrap might be found under `/css/bootstrap-3.3.7-dist/css/bootstrap.min.css`. A new version would be available under a different path – for example, `/css/bootstrap-4.0.0-dist/css/bootstrap.min.css`.

<sup>74</sup><https://crimson-portal.herokuapp.com/>

## Simpler JavaScript Code

The JavaScript code in the example is quite flexible and can also deal with the failure of a service. A primitive alternative is shown in the [SCS jQuery Project<sup>75</sup>](#). In essence, it uses the following JavaScript code:

```
$(document).ready(function() {
  $("a.embeddable").each(function(i, link) {
    $("<div />").load(link.href, function(data, status, xhr) {
      $(link).replaceWith(this);
    });
  });
});
```

The code uses jQuery to search for hyperlinks (`<a ...>`) with the CSS class `embeddable`, and then replaces the link with the content the link refers to.

This demonstrates how simple it is to implement a client integration with JavaScript. At the minimum, it is just seven lines of jQuery code.

Of course it is also possible that each microservice has its own code for transclusion. This seems like a redundant implementation at first sight. However, it is no uncommon for each microservice to have its own code to parse e.g. a JSON data structure. So custom code per microservice for transcluding HTML can also be an option.

## Integration using other Frontend Integrations

Of course, client-side integration and links with server-side integration (see [chapter 9](#)) can be combined. Both approaches have different benefits:

- Web pages with server-side integration originate entirely from the server. Thus, the integration makes sense when the web page can be correctly displayed only if all the content is integrated.
- With client-side integration, transclusion is not executed when the other server is not available. The web page is still displayed, just without transclusion. This can be the better option because it improves resilience. However, the webpage would need to be usable without the transcluded content.

However, a server-side integration requires additional infrastructure. For client-side integration, this is not necessary. Therefore, it makes sense to start with client-side integration and to supplement server-side integration when necessary.

---

<sup>75</sup><https://github.com/ewolff/SCS-jQuery/>

## Other Integrations

Synchronous communication ([chapter 13](#)) or asynchronous communication ([chapter 10](#)) enable the communication of the backend system and therefore can be combined, of course, with client-side frontend integration.

## 8.4 Experiments

- Start the example and disable JavaScript in the browser. Is the example still usable? Specifically, does the *postbox* integration still work?
- Analyze the JavaScript code for transclusion at <https://github.com/ewolff/crimson-styleguide/tree/master/components/preview>. How difficult is it to replace this code by an implementation with a different JavaScript library?
- Supplement the system with an additional microservice.
  - A microservice which generates a note for a meeting with a client can serve as example.
  - Of course, to add the service you can copy and modify one of the existing Node.js or the Spring Boot microservice.
  - The microservice has to be accessible by the *portal* microservice. To achieve this, you have to integrate a link to the new microservice into the portal.
  - The link can provide the partner ID to the new microservice. This ID identifies the customer and might be useful to figure out which customer the note belongs to.
  - After entering the note, the microservice can trigger a redirect back to the portal.
  - For a uniform look and feel, you have to use the assets from the styleguide project. The Spring Boot project for the *postbox* shows the integration for Spring/Java and the portal for Node.js. Of course, you can also use other technologies for the implementation of the new microservice.
  - The microservice can store the data concerning the meeting in a separate database.
  - Package the microservice in a Docker image.
  - Reference the Docker image in `docker-compose.yml`.

## 8.5 Conclusion

This example system is deliberately presented in the first chapter on frontend integration. It shows how much is already possible with a simple integration via links. Only for the *postbox* do you need some JavaScript. Before using the advanced technologies for frontend integration, one should first understand what is already possible with such a simple approach.

The example integrates different technologies. In addition to the Node.js systems, there is also a Java/Spring Boot application which seamlessly integrates into the system. This demonstrates that a frontend integration results in only a few limitations regarding the technology choice.

## ROCA

ROCA helps especially with this type of integration. The microservices can be accessed via links, making integration very easy. At the same time, the applications are largely decoupled in terms of deployments and technology. An application can easily be deployed in a new version and not affect the other applications. The applications can also be implemented in different technologies.

At the same time, the ROCA UI is comfortable and easy to use. Compared to a single-page app (SPA), there are no compromises in user comfort.

## Assets

Finally, the application shows how to handle assets, in this case by using a common Node.js project. As a result, each application can decide for itself when to adopt a new version of the assets. This is important because otherwise a change of assets is automatically rolled out to all applications and might cause problems in the applications. However, several versions of the assets should be used only temporarily on the web page. After all, the design and the look and feel should be uniform. Dealing with the asset project in such a way that not all services always use the current version is only meant to minimize the risk of an update, but must not lead to long-term inconsistencies.

However, the asset project also ensures that all web pages contain jQuery in the version that the asset project uses. Thus, the asset project limits the freedom of individual projects with regards to JavaScript libraries.

## Self-contained Systems

Unlike Self-contained Systems (see [section 3](#)), this solution uses a common backend. With an SCS, the logic should also be part of the respective SCS and not be implemented in another system. However, it is still possible to use some SCS ideas even if all systems share a common backend. The systems do not have to deal with logic and storing data as much as an SCS, because they are in the backend. Thus, this system shows how a well-modularized portal for a monolithic backend can be implemented.

But this approach also presents challenges. Any changes to the system will probably affect one of the frontend applications and also the backend. Therefore, the development and deployment of the two components must be coordinated.

## Benefits

- Loose coupling
- Resilience
- No additional server components
- Low technical complexity
- Links often enough

## Challenges

- Uniform look and feel

# 9 Recipe: Server-side Integration using Edge Side Includes (ESI)

Servers can also integrate multiple frontends. This chapter focuses on [ESI<sup>76</sup>](#) (Edge Side Includes). Readers discover the following:

- How the web cache Varnish implements ESI.
- How applications can implement an integration using ESI.
- What benefits and disadvantages ESI has and what alternatives to ESI exist for implementing server-side frontend integration.

## 9.1 ESI: Concepts

ESI (Edge Side Includes) enables web applications to integrate HTML fragments of another web application (see [figure 9-1](#)). To do so, the web application sends HTML containing ESI tags. The ESI implementation analyzes the ESI tags and integrates HTML fragments of other web applications at the right positions.

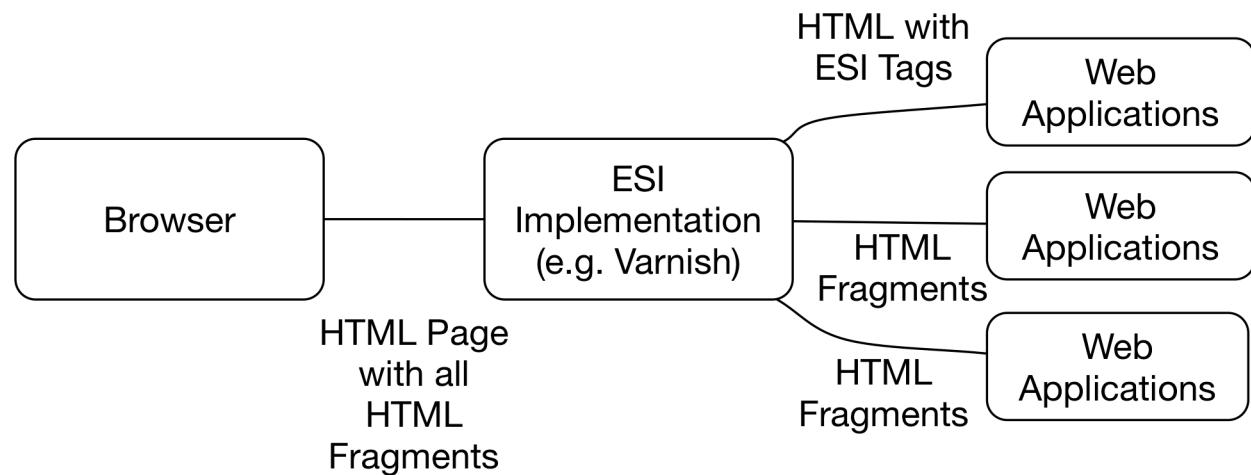


Fig. 9-1: Integration with ESI

<sup>76</sup><https://www.w3.org/TR/esi-lang>

## Caches Implement ESI

In the example, the web cache Varnish<sup>77</sup> serves as ESI implementation. Other caches, such as Squid<sup>78</sup>, also support ESI. Websites use these caches to deliver web pages out of the cache upon incoming requests. The servers handle requests only for cache-misses. This speeds up the website and decreases the load of the web servers.

## CDNs Implement ESI

Content Delivery Networks (CDNs) such as Akamai<sup>79</sup> also implement the ESI standard. In principle, CDNs serve to deliver static HTML pages and images. To do so, CDNs run servers at several Internet nodes so that every user can load web pages and images from a nearby server, and thus reducing loading times. Via the support of ESI, the assembling of HTML fragments can be done on a server that is close to the user.

CDNs and caches implement ESI to be able to assemble web pages from different fragments. Static parts can be cached, even if other parts have to be dynamically generated. This makes it possible to at least partially cache dynamic web pages which otherwise would have to be excluded from caches completely. This improves performance. Therefore, ESI not only offers features for frontend integration, but also features especially useful for caching.

## 9.2 Example

The [example<sup>80</sup>](#) shows how Edge Side Includes (ESI) can be used to assemble HTML fragments from different sources and how the entire HTML can be sent to the browser. For this, the HTML contains special ESI tags which are replaced by HTML fragments.

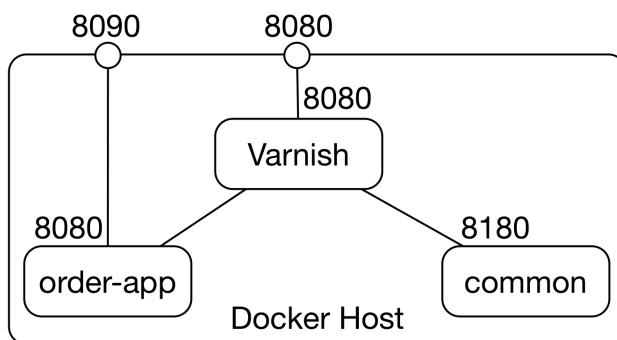


Fig. 9-2: Overview of the ESI Example

[Figure 9-2](#) shows an overview of the structure of the example. The Varnish cache directs the HTTP request to the *order* microservice or the *common* service. The *order* microservice contains the

<sup>77</sup><https://varnish-cache.org/>

<sup>78</sup><http://www.squid-cache.org/>

<sup>79</sup><http://www.akamai.com/html/support/esi.html>

<sup>80</sup><https://github.com/ewolff/SCS-ESI>

logic for processing orders. The common service offers CSS assets and HTML fragments which microservices have to integrate in their HTML pages. The example therefore shows a typical scenario. The applications like order deliver content that is displayed in a frame. The frame is provided by common so that all applications can uniformly integrate it.

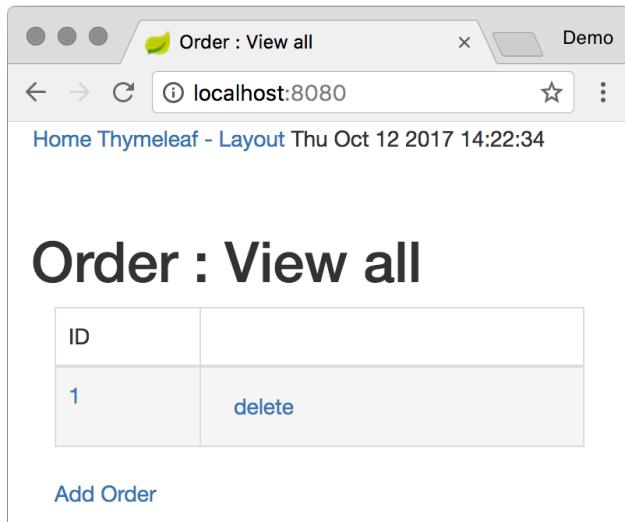


Fig. 9-3: Screenshot of the ESI Example

Figure 9-3 shows one page of the ESI example. The links to the home page and Thymeleaf and also the date are provided by the common service. Also the CSS and therefore the layout originate from the common service. The order service only provides the list of orders. Thus, when additional microservices have to be integrated into the system, they only have to return the respective information in the middle. The frame and the layout is added by the common service.

During a reload, the time is updated but only every 30 seconds because the data is cached for that long. The cache works only if no cookies were sent in the request.

## Running the Example

Section 0.4 describes what software has to be installed for the example.

To start the example, download the code with `git clone https://github.com/ewolff/SCS-ESI.git`. Afterwards, you have to compile the Java application in sub directory `scs-demo-esi` with `./mvnw clean package` (macOS, Linux) or `mvnw.cmd clean package` (Windows). See [appendix B](#) for more details on Maven and how to troubleshoot the build. Finally, you can build the Docker images in the directory `docker` with `docker-compose build` and start the example with `docker-compose up -d`. See [appendix C](#) for more details on Docker, Docker Compose, and how to troubleshoot them.

<https://github.com/ewolff/SCS-ESI/blob/master/HOW-TO-RUN.md> explains the installation procedure and how to start the example in more detail.

Varnish, which is available n the Docker host at port 8080, receives the HTTP requests and processes the ESI tags. If the Docker containers are running on the local computer, you can reach Varnish at

<http://localhost:8080>. The web pages of the order microservice can be accessed at <http://localhost:8090>. These web pages contain the ESI tags and therefore appear broken if displayed in a web browser.

## 9.3 Varnish

Varnish<sup>81</sup> is a web cache and is used as ESI implementation in the example.

Varnish is mainly used for optimizing web servers. Varnish intercepts HTTP requests to web servers, caches the responses, and forwards only those requests to the web servers that are not in the cache. This improves the performance.

### Licence and Support

Varnish is licensed under a [BSD license<sup>82</sup>](#). The cache is mainly developed by [Varnish Software<sup>83</sup>](#), which also provides commercial support.

### Caching with HTTP and HTTP Headers

Caching data correctly is not a trivial matter. Above all, the questions are when content can be retrieved from the cache and when the data must be retrieved from the web server, because the data in the cache no longer represents the current state. Varnish uses its own HTTP headers for this. The HTTP protocol has very good support for caching through its HTTP headers. The control over whether data is cached rests with the web server, which informs the cache of the settings via HTTP headers. Only the web server can decide whether a page can be cached because that depends on the domain logic.

### Varnish Docker Containers

In the example, Varnish runs in a Docker container which contains a Ubuntu 14.04 LTS. On the Ubuntu image, the Varnish version from the official Varnish repository will be installed.

### Varnish Configuration

Varnish offers a powerful configuration language. For the example, Varnish is installed in its own Docker container. The configuration can be found in the file `default.vcl` in the directory `docker/varnish/`. Here is an extract with the essential settings.

---

<sup>81</sup><https://varnish-cache.org/>

<sup>82</sup><https://github.com/varnishcache/varnish-cache/blob/master/LICENSE>

<sup>83</sup><https://www.varnish-software.com/>

```

vcl 4.0;

backend default {
    .host = "order";
    .port = "8080";
}

backend common {
    .host = "common";
    .port = "8180";
}

sub vcl_recv vcl_recv {
    if (req.url ~ "^/common") {
        set req.backend_hint = common;
    }
}

sub vcl_backend_response{
    set beresp.do_esi = true;
    set beresp.ttl = 30s;
    set beresp.grace = 15m;
}

```

- `vcl 4.0;` chooses version 4 of the Varnish configuration language.
- The first backend has the name `default`. Each request arriving at the Varnish cache is passed on to the web server if there is no other configuration. The hostname `order` is resolved by Docker Compose. The `default` backend is the `order` microservice, which implements all functions to accept and display orders.
- The second backend has the name `common` and is provided by the host of the same name. Also in this case, Docker Compose resolves the name to a Docker container. The `common` service provides headers and footers for the HTML pages of the microservices and `Bootstrap84` as a shared library for the UI of the microservices.
- When an HTTP request arrives for a URL where the path starts with `/common`, the HTTP request is redirected to the `common` backend. The code of the subroutine `vcl_recv` is responsible for this. Varnish automatically calls this routine to determine the routes for the requests.
- The subroutine `vcl_backend_response` configures Varnish with `beresp.do_esi` so that Varnish interprets ESI tags. `beresp.ttl` turns on the caching. Each page is cached for 30 seconds. Finally, `beresp.grace` ensures that if a backend fails, the web pages are cached for 15 minutes. This can temporarily compensate for a backend failure. Of course, this works only if the web page is already in cache because it has been accessed before the failure. Therefore, if the page is not in

---

<sup>84</sup><http://getbootstrap.com/>

the cache or not cacheable at all, this feature does not help. However, this caching is very simple: If a new order has been created, it is not displayed until the cache has been invalidated. This can take up to 30 seconds. It would, of course, be better if a new order leads to invalidation of the cache. This is still relatively easy to implement in the example, but in a complex application, it can be difficult to invalidate the correct pages. For example, goods are displayed on product pages and order pages, so several pages need to be invalidated if the data for the goods is changed. Thus, simple time-based caching can be the better solution, being easy to implement and possibly doing a good enough job. There is a chapter<sup>85</sup> about cache invalidation in the Varnish book.

The present configuration not only enables ESIs, but also implements the functionality of a reverse proxy and redirects requests to specific microservices.

## Load Balancing

You can add [load balancing<sup>86</sup>](#) to the Varnish configuration, thereby splitting the requests to the microservices can be split across multiple microservices by defining multiple backends in the configuration. However, you can of course also rely on an external load balancer. In that case, Varnish would do only ESI and caching.

## Evaluation of VCL

As you can see, VCL is a very powerful language that has many possibilities for manipulating HTTP requests. That is essential, for example, in a case where you can only be sure a request can be cached if it does not contain cookies (because cookies could change the response): therefore, VCL must be able to remove cookies.

A comprehensive documentation of Varnish and VCL can be found in the free [Varnish book<sup>87</sup>](#).

## Order Microservice

The *order* microservice offers a normal web interface, which, however, has been supplemented with ESI tags in some places.

A typical HTML page of the order microservice looks like this:

---

<sup>85</sup>[https://book.varnish-software.com/4.0/chapters/Cache\\_Invalidation.html](https://book.varnish-software.com/4.0/chapters/Cache_Invalidation.html)

<sup>86</sup><https://varnish-cache.org/trac/wiki/LoadBalancing>

<sup>87</sup><http://book.varnish-software.com/>

```
<html>
<head>
  ...
  <esi:include src="/common/header"></esi:include>
</head>

<body>
  <div class="container">
    <esi:include src="/common/navbar"></esi:include>
    ...
  </div>
  <esi:include src="/common/footer"></esi:include>
</body>
</html>
```

## HTML with ESI Tags in the Example

The order microservice is available at port 8090 of the Docker host. The output goes past the Varnish and still contains the ESI tags. At <http://localhost:8090/> the HTML with the ESI tags can be viewed.

The ESI tags look like normal HTML tags. They only have an `esi` prefix. Of course, a web browser cannot interpret them.

### ESI Tags in the HTML Head

In the head the ESI tags are used to integrate common assets like Bootstrap into all pages. Changing the header under `"/common/header"` causes all pages to get new versions of Bootstrap or other libraries. If the pages with a new version are no longer displayed correctly, such a change will cause problems. Therefore, the pages themselves should be responsible for using new versions. For this, a version number can be encoded in the URL in the ESI tag, for example.

### ESI Tags in the Remaining HTML

The ESI Include for `"/common/navbar"` ensures that each web page has the same navigation bar. Finally, `"/common/footer"` can contain scripts or a footer for the web page.

### Result: HTML at the Browser

Varnish collects these HTML snippets from the common service so that the browser receives the following HTML:

```

<html>
<head>
...
<link rel="stylesheet"
      href="/common/css/bootstrap-3.3.7-dist/css/bootstrap.min.css" />
<link rel="stylesheet"
      href="/common/css/bootstrap-3.3.7-dist/css/bootstrap-theme.min.css" />
</head>

<body>
<div class="container">
<a class="brand"
   href="https://github.com/ultraq/thymeleaf-layout-dialect">
  Thymeleaf - Layout </a>
  Mon Sep 18 2017 17:52:01 </div></div>
...
</div>
<script src="/common/css/bootstrap-3.3.7-dist/js/bootstrap.min.js" />
</body>
</html>

```

The ESI tags have thus been replaced by suitable HTML snippets.

ESI offers many other features, for example, for securing a system against the failure of a web server, or for integrating HTML fragments only under certain conditions.

## No Tests without ESI Infrastructure

A problem with the ESI approach is that individual services cannot be tested without an ESI infrastructure. At the very least, they do not display any pages with meaningful content, because the ESI tags would have to be interpreted for that. This works only if the HTTP requests are routed through Varnish. Therefore, suitable environments containing a Varnish must be provided for the development.

## Effects on the Application

The application is a normal Spring Boot web application without any dependencies on Spring Cloud or ESI. This shows that pure frontend integration leads to a very loose coupling and has little impact on the applications.

## Common Microservice

In the example, the `common` service is a very simple Go<sup>88</sup> application. It handles the three URLs `"/common/header"`, `"/common/navbar"`, and `"/common/footer"`. For these URLs, the Go code

---

<sup>88</sup><https://golang.org/>

generates suitable HTML fragments.

## Asset Server

The Go code also contains a web server that provides static resources under "/common/css/" – the Bootstrap framework. In this way, the common microservice assumes the function of an asset server. Such a server offers CSS, images, or JavaScript code to applications. The ESI example shows an alternative for the integration of shared assets. In [chapter 8](#), a common asset project has ensured that all applications can use the same assets. In the example in this chapter, an asset server is used for this purpose.

The application displays the current time in the navigation bar. This shows that dynamic content can also be displayed with ESI includes. [Section 5.4](#) already explains how this part of the system functions and can be built.

## 9.4 Recipe Variations

Of course, instead of Varnish, a different ESI implementation could be used, for example, by Squid<sup>89</sup> or by a CDN like Akamai<sup>90</sup>.

### SSI

Another option for server-side frontend integration is SSI<sup>91</sup> (Server Side Includes). This is a feature that most web servers offer. <https://scs-commerce.github.io/> is an example of a system that uses SSI with the nginx web server to integrate the frontends.

SSI and ESI have different benefits and disadvantages.

- Web servers are often already available in the infrastructure for SSL/TLS termination or for other reasons. Because web servers can implement SSI, no additional infrastructure such as a Varnish is necessary.
- Caches not only speed up applications, but also compensate for web server failures for some time, thus improving resilience. This speaks for ESI and a cache like Varnish. ESI also has more features for further optimizing caching. However, correct caching can also be difficult to implement. For example, changing the data of a single data record can trigger a cascade of invalidations. Finally, every page and every HTML fragment containing information about the goods must be regenerated.

---

<sup>89</sup><http://www.squid-cache.org/>

<sup>90</sup><https://www.akamai.com/us/en/support/esi.jsp>

<sup>91</sup>[https://en.wikipedia.org/wiki/Server\\_Side\\_Includes](https://en.wikipedia.org/wiki/Server_Side_Includes)

## Tailor

Tailor<sup>92</sup> is a system for server-side frontend integration that Zalando implemented as part of Mosaic<sup>93</sup>. It is optimized for showing the user the first parts of the HTML page as quickly as possible. For e-commerce the rapid display of a web page is very important to keep users and can increase sales. To achieve this, Tailor implements a BigPipe<sup>94</sup>. First, very simple HTML code is transferred to the user in order to be able to display a simple page very quickly. JavaScript is used to load more details step by step. Tailor implements this with asynchronous I/O using Node.js streams.

## Client-side Integration

Client-side integration does not use any additional infrastructure and can be the simpler option for frontend integration. Therefore, it makes sense to find out how far you can go with client-side integration before using server-side frontend integration.

For an integration of a header and footer as in the example for this chapter, server-side integration is the better choice because a page cannot be displayed without these elements. The pages should be delivered to the user in such a way that they can be displayed to the user without loading any additional content.

Therefore, client-side integration for optional elements makes sense. Dealing with failed services is then a task for the client code. That might simplify the server implementation and the server setup.

## Shared Library

The example from chapter 8 uses a library to deliver assets. Theoretically, the HTML fragments that are integrated with ESI in this example could also be delivered as a library. But then all systems would have to be rebuilt and deployed for an additional link in the navigation. With the ESI solution, all you have to do is change the HTML fragment on the server.

## Additional Integration

Pure frontend integration is rarely enough. Therefore, a system will combine backend integration with synchronous (see chapter 13) or asynchronous (see chapter 10) communication mechanisms with frontend integration. An exception is a scenario like in chapter 8, where a complex portal application is implemented. The parts of the portal can communicate through frontend integration. Backend integration is not necessary because the services do not implement a lot of logic and have no database, but only create a web interface.

---

<sup>92</sup><https://github.com/zalando/tailor>

<sup>93</sup><https://www.mosaic9.org/>

<sup>94</sup><https://de-de.facebook.com/notes/facebook-engineering/bigpipe-pipelining-web-pages-for-highperformance/389414033919/>

## 9.5 Experiments

- Supplement the system with an additional microservice.
  - A microservice which simply displays a static HTML page can serve as example.
  - Of course, you can simply copy and modify the existing Go or the Spring Boot microservice.
  - The new microservice should integrate header, navigation bar, and footer of the common microservice via ESI tags.
  - Package the microservice as a Docker image and reference it in `docker-compose.yml`. There you can also determine the name of the Docker container.
  - The microservice has to be accessible via Varnish. To achieve that you have to integrate a new backend with the name of the Docker container in `default.vcl` and adapt the routing in `vcl_recv()`.
  - Now you should be able to access the new microservice at, for example, <http://localhost:8080/name> if the Docker containers run on the local computer and the new service is configured in Varnish's routing with `name`. The ESI tags should have been replaced by HTML code.
- The Go application returns only a dynamic HTML fragment – the navigation bar with the current time. Instead of the Go application, a web server can also deliver static pages. For example, replace the Go application with an Apache httpd server that delivers the HTML fragments and the Bootstrap library. The time does not necessarily have to be displayed in the navigation bar, so a static HTML fragment is all that is needed.
- Change the caching so that the pages are immediately invalidated when new orders are received. You can, for example, use matching HTTP headers as explained in a [chapter<sup>95</sup>](#) of the Varnish book. An alternative is to remove objects directly from the cache. The Varnish book also contains a [chapter<sup>96</sup>](#) on this option.
- Replace the Varnish cache in the example with [Squid<sup>97</sup>](#), which also implements ESI.
- Replace ESI with SSI and replace the Varnish cache with an Apache httpd or nginx.
- What happens if the web servers fail? Simulate the failure with `docker-compose up --scale common=0` or `docker-compose up --scale order=0`. Which parts of the web page still work? Is it still possible to place orders, for example?

## 9.6 Conclusion

ESIs are a possible implementation of frontend integration and lead to loose coupling. The applications are simple web applications that, apart from the ESI tag, have no dependencies on the infrastructure.

<sup>95</sup><http://book.varnish-software.com/4.0/chapters/HTTP.html>

<sup>96</sup>[http://book.varnish-software.com/4.0/chapters/Cache\\_Invalidation.html](http://book.varnish-software.com/4.0/chapters/Cache_Invalidation.html)

<sup>97</sup><http://www.squid-cache.org/>

The integration with ESIs has the advantage that the web pages are completely assembled by the cache and can be displayed directly in the browser. Therefore, the page isn't delivered unusable in any way due to fragments that still have to be loaded.

Using a cache together with ESI has the advantage that fragments can be cached. This means that not only static pages but also static parts of dynamic pages can be cached, which improves performance. The pages can even be cached and assembled from a CDN that supports ESI, further improving performance.

The cache can also be used to achieve a certain degree of resilience. If a web server fails, the cache can return the old data. In this way, the web page remains available but might potentially return invalid information. However, for this the cache must hold the pages a long time. In addition, the cache must also check the availability of the services.

## Benefits

- Web page is always delivered in entirety
- Resilience via cache
- Higher performance via cache
- No code in the browser

## Challenges

- Uniform look and feel
- Additional server infrastructure necessary

# 10 Concept: Asynchronous Microservices

Asynchronous microservices have advantages over synchronous microservices. This chapter discusses:

- How microservices can communicate asynchronously.
- Which protocols can be used for asynchronous communication.
- How events and asynchronous communication are linked.
- The advantages and disadvantages of asynchronous communication.

## 10.1 Definition

Asynchronous microservices distinguish themselves from synchronous microservices. [Chapter 13](#) describes synchronous microservices in detail. The term “synchronous microservices” stands for the following:

A microservice is synchronous if it makes a request to other microservices while processing requests and waits for the result.

The logic to handle a request in the microservice might therefore not depend on the result of a request to a different microservice. So, a definition of asynchronous microservices would be:

A microservice is asynchronous (a) if does not makes a request to other microservices while processing requests or (b) makes a request to other microservices while processing requests and does not wait for the result.

So there are two cases here:

- (a) The microservice does not communicate at all with other systems while processing a request. In that case, the microservice will typically communicate with the other systems at a different time (see [figure 10-1](#)). For example, the microservice can replicate data that is used when processing a request. In this way, customer data can be replicated so that when processing

an order, the microservice can access the locally available customer data, instead of having to load the necessary customer data for each request via a request to another system.

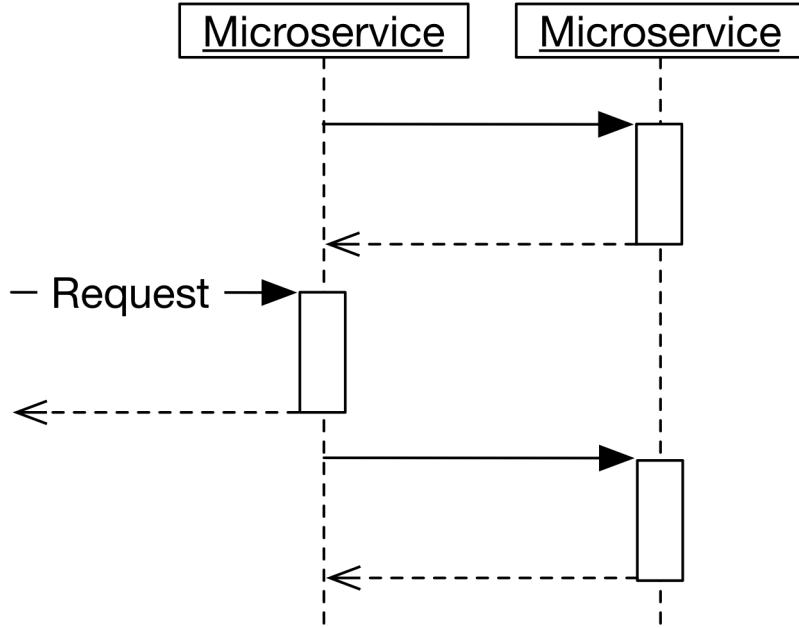


Fig. 10-1: Communication Only Outside of Requests

- (b) The microservice sends a request to another microservice, but does not wait for a response (see [figure 10-2](#)). A microservice responsible for processing an order can send a request to another microservice which generates the invoice. A response to this request is not necessary for processing the order so there is no need to wait for it.

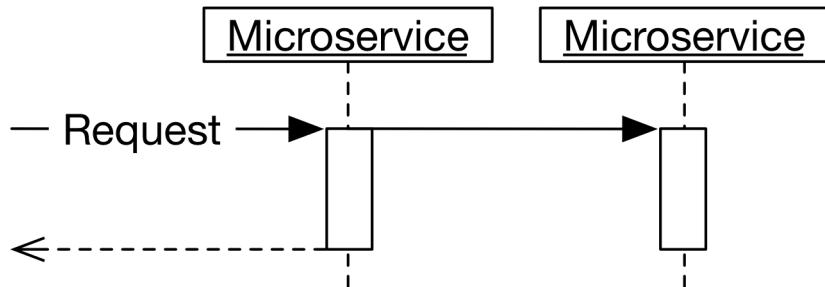


Fig. 10-2: Communication Without Waiting for a Response (Fire-and-Forget)

[Figure 10-3](#) shows an example for a more complex asynchronous architecture. In this e-commerce system, orders are processed. Via the catalog, customers can choose goods for an order. The *order process* generates the orders. An *invoice* and a *shipping* data record is produced for the order. The *registration* microservice adds new customers to the system. The *listing* microservice is responsible for new goods.

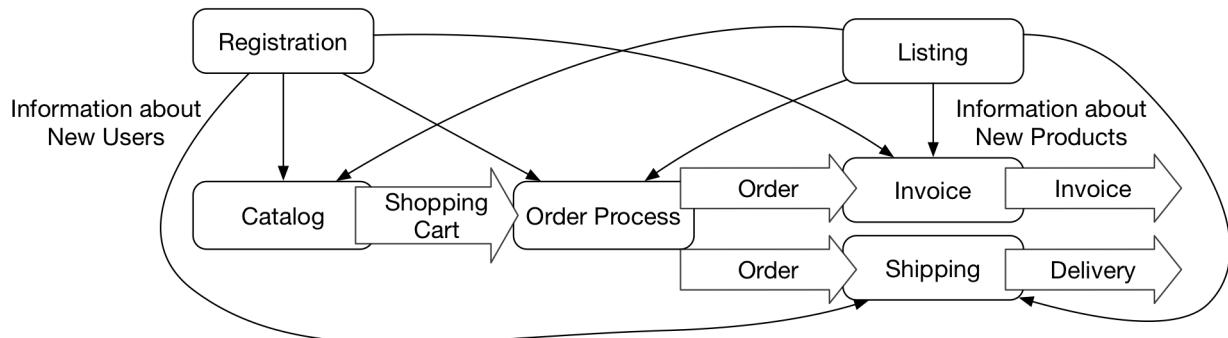


Fig. 10-3: Architecture for an Asynchronous System

## Asynchronous Communication with No Response

The four systems *catalog*, *order process*, *invoice*, and *shipping* send asynchronous notifications for processing the orders. The catalog collects goods in the shopping cart. If the user orders the shopping cart, the catalog transfers the cart to the order process. The order process turns the shopping cart into an order. The order then becomes an invoice and a delivery. Such requests can be executed asynchronously. No data has to flow back. The responsibility for the order is transferred to the next step in the process.

## Data Replication and Bounded Context

This becomes more complicated if data is required to execute a request. For example, in the catalog, in the order process, and for the invoice, data about products and customers has to be available. So each of the systems stores a part of the information about these business objects. The *catalog* must display the products, so it has pictures and descriptions of the products. For invoices, prices and tax rates are important. This corresponds to the bounded contexts from [section 2.1](#), each of which has its own domain model.

Each bounded context has its own domain model. That means that all data for the bounded context is represented in its domain model. Therefore, the data specific for the bounded context should be stored in the bounded context in its own database schema. Other bounded contexts should not access that data directly, which would compromise encapsulation. Instead, the data should be accessed only by the logic in the bounded context and its interface.

Although it would be possible to have a system that contains all information about, for example, a product, this would not make a lot of sense. The model of the system would be very complicated. Also it means that the domain model would be split across, say, a system for an order process and a system for the product data. That would lead to a very tight coupling.

A third system, such as *registration* for customer data or *listing* for product data, must accept all the data and transfer the needed parts of the data to the respective systems. This can also be done asynchronously. The other bounded contexts then store the information about products and customers in their local databases, making replication just a result of events being processed. An

event such as “new product added” makes each bounded context add some data to its domain model. *Registration* or *listing* do not need to store the data. After they have sent the data to the other microservices, their job is done.

It is also possible to do an extract-transform-load approach. In that case, a batch would *extract* the data from one bounded context, transform it to a different format, and load it into the other bounded context. This is useful if a bounded context should be loaded with an initial set of data, or if inconsistencies in the data require a fresh start.

## Synchronous Communication Protocols

Asynchronous communication as defined previously does not make any assumptions about the communication protocol used. For synchronous communication, the server must respond to each request. Examples are REST and HTTP. A request leads to a response that contains a status code and optionally additional data. It is possible to implement asynchronous communication with a synchronous communication protocol. [Chapter 12](#) explains this in more detail.

## Asynchronous Communication Protocols

It is more natural to implement asynchronous communication with an asynchronous communication protocol. An asynchronous communication protocol sends messages and does not expect responses. Messaging systems like Kafka (see [chapter 11](#)) implement this approach.

There is also a [presentation<sup>98</sup>](#) which explains the difference between REST and messaging, and highlights that both technologies can be used to implement synchronous communication like *request/reply*, but also asynchronous communication like *fire & forget* or *events*.

## 10.2 Events

With asynchronous communication, the coupling of systems can be driven to different lengths. As already mentioned, the system for order processing could inform the invoice system asynchronously that an invoice has to be written. The ordering system thus determines exactly what the invoicing system has to do: generate an invoice. It also sends a message to the bounded context shipping to trigger the delivery.

The system can also be set up differently. The focus will then be on an event like “There is a new order.” Every microservice can react appropriately to this. The invoicing system can write an invoice and the shipping system can prepare the goods for delivery. So each microservice decides for itself how it reacts to the events (see [figure 10-4](#)).

---

<sup>98</sup><https://www.slideshare.net/ewolff/rest-vs-messaging-for-microservices>

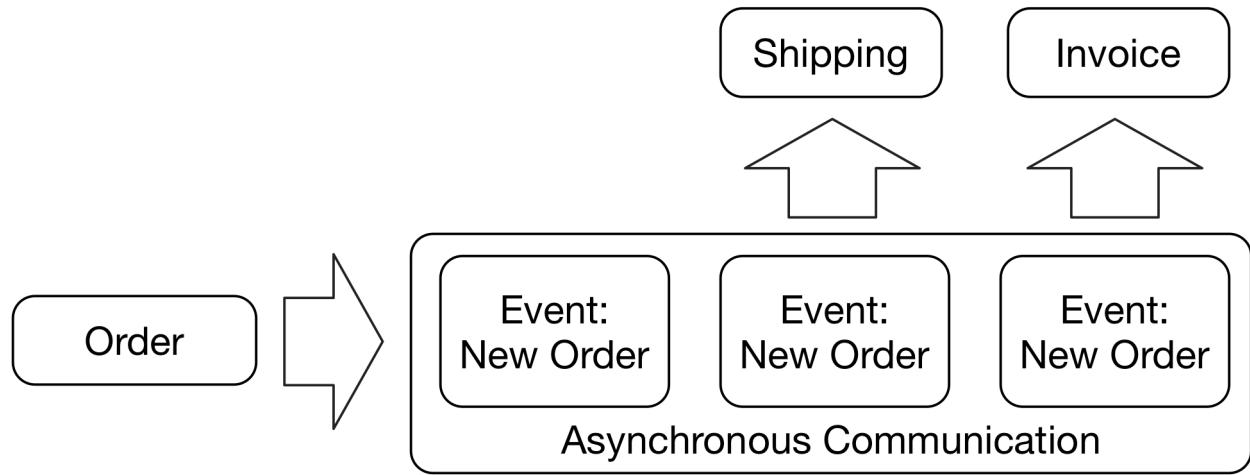


Fig. 10-4: Asynchronous Events

This leads to a better decoupling. If a microservice has to react differently to a new order, the microservice can implement this change on its own. It is also possible to add a new microservice, which generates statistics, for example, when a new order is placed.

## Events and DDD

However, this approach is not quite as easy to implement. The crucial question is what data is transferred with the event. If the data is to be used for such different purposes as the writing of an invoice, statistics, or recommendations, then a large number of different attributes must be stored in the event.

This is problematic because domain-driven design shows that each domain model is valid only in a bounded context (see [section 2.1](#)). For invoicing, prices and tax rates have to be known. For shipping, size and weight of the goods are needed to organize a suitable transport.

## Patterns from Strategic Design

The views of invoice and shipping on the data of an order thus represent two bounded contexts, which can receive data from a third bounded context, the order process. Domain-driven design defines patterns for this (see [section 2.1](#)). For example, with customer/supplier, the team for invoicing and shipping can define what data it needs to receive. The team that provides the data for the order must meet these requirements. This pattern defines the interaction of the teams that develop the bounded contexts that participate in the communication relationship. Such coordination is necessary regardless of whether events are sent, or whether communication between components takes place by a synchronous call.

In other words: Events may seem to decouple the system, but coordination regarding the necessary data must still take place. This means that events do not necessarily lead to a truly decoupled system. In extreme cases, events can even lead to hidden dependencies. Who reacts to an event? If this

question can no longer be answered, the system is hardly changeable anymore because changes to the events have had unforeseeable consequences.

A solution might be to provide specific types of events for each receiver. Each type of event would just contain the information that this receiver needs. So, if a new order appears in the system, an event is sent to the invoicing system with the data it needs. Another event is sent to shipping with the data for that system. The two systems are truly decoupled: A change in the interface to one of the systems does not influence the other system. This can be the result of a customer/supplier relationship.

Another solution would be to use a published language (see [section 2.1](#)). In that case, a common data structure exists that contains all information for all receivers. This might make it hard to understand which receivers use what: changes to the data structure might lead to unforeseen problems. However, there is just one data structure so it is somewhat easier to implement the system.

A very important matter is the difference in information that each receiver needs. If it is mostly the same, a published language might be better. If it is very different, it might make more sense to have separate data structures. For the case of invoicing and shipping, there is probably not too much overlap, so two separate data structures might be the better alternative.

## Sending Minimal Data in an Event

There is yet another way to deal with this problem. In the event, just an ID number is sent along, for example, the number of a new order. Afterwards every bounded context can decide for itself how to get the necessary data. There can be a special interface for each bounded context that provides the appropriate data for that specific bounded context.

## Event Sourcing

An architecture focus on events can also entail other advantages. The state each microservice has in its database is the result of the events it has received. So the state of a microservice can be restored by resending all events it has received so far. The microservice can even change its internal domain model and then process the events again to rebuild its database with the new version of the domain model. That facilitates database schema migration. Thus, each microservice can have its own domain model according to the bounded context pattern, but all microservices are still connected by the events they send to each other. An overall state of the system no longer exists, but when all events are saved and can be retrieved, the state of each microservice can be reconstructed. These ideas form the basis for event sourcing.

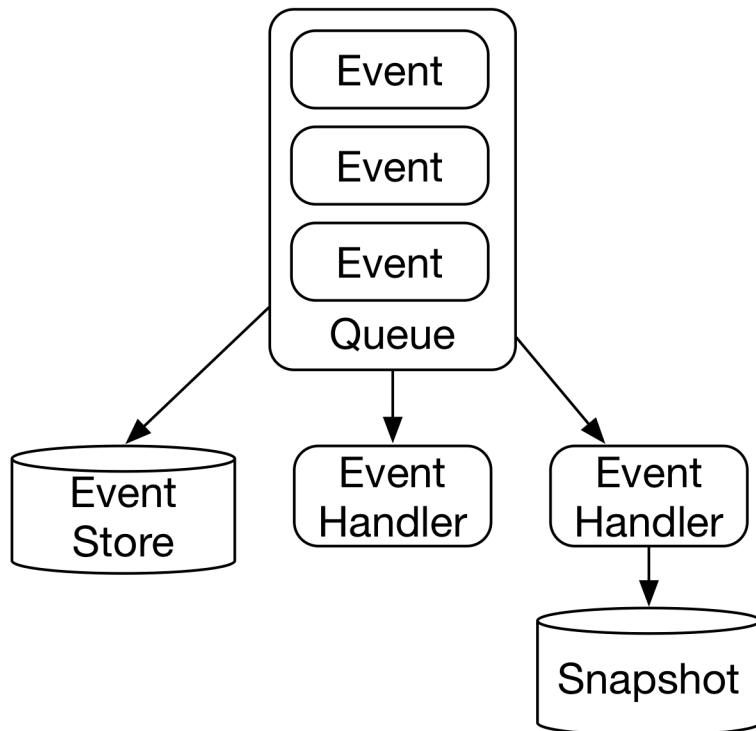


Fig. 10-5: Event Sourcing

The elements of an event sourcing implementation are shown in figure 10-5:

- The *event queue* sends the events to the recipients.
- The *event store* saves the events.
- *Event handlers* process the events. They can save their state as a *snapshot* in a database.

The event handler can read the current state from the snapshot. The snapshot can be deleted. The snapshot can be restored on the basis of the events, which can be retrieved from the event store. As an optimization, an event handler can also reconstruct its state from an older version of the snapshot.

There is a difference between the events for event sourcing and domain events; see [Christian Stettler's blog post<sup>99</sup>](#).

## Individual or Shared Event Store?

The event store can be part of the microservice that receives events and stores them in its own event store. Alternatively, the infrastructure not only sends the events, but also stores them. At first glance, it seems better if the infrastructure stores the events because it simplifies the implementation of the microservices. In such a case, the event store would be implemented in the event queue.

If each microservice stores the events in its own event store, the microservice can store all relevant data in the event, which the microservice may have collected from different sources. When storing

<sup>99</sup><https://www.innoq.com/en/blog/domain-events-versus-event-sourcing/>

events in the infrastructure, it is necessary to find a model of the event that satisfies all microservices. Such a model for the events can be a challenge because of the concept of bounded context (see [section 2.1](#)). After all, every microservice is a separate bounded context with its own domain model, so finding a common model is difficult.

## 10.3 Challenges

If the communication infrastructure for event sourcing has to store old events, it has to handle considerable amounts of data. Consequently, if old events are missing, the state of a microservice can no longer be reconstructed from the events. As an optimization, it would be possible to delete events that are no longer relevant. If a customer has moved to a different address several times, the last address is probably the only relevant one. The others can then be deleted.

In addition, it must also be possible to continue processing old events. If the schema of the events changes in the meantime, old events have to be migrated. Otherwise, every microservice has to be able to handle events in all old data formats. This is particularly difficult if new data has to be contained in events that have not yet been saved in old events.

### Inconsistency

Due to asynchronous communication, the system is not consistent. Some microservices already have certain information, others do not. For example, order process might already have information about an order, but invoicing or shipping do not know about the order yet. This problem cannot be solved. It takes time for asynchronous communication to reach all systems.

### CAP Theorem

These inconsistencies are not only practical problems, but cannot even be solved in theory. According to the [CAP theorem<sup>100</sup>](#), three characteristics exist in a distributed system:

- *Consistency* (C) means that all components of the system have the same information.
- *Partition tolerance* (P) means that a system will continue to work in case of arbitrary package loss in the network.
- *Availability* (A) means that no system stops working because another system failed.

The CAP theorem states that a system can have a maximum of two features out of these three. Partition tolerance is a special case. A system must react if the network fails. In fact, not even a complete failure is necessary; the package loss just needs to be high or the response time very long. A system that responds very slowly is indistinguishable from a system that has failed completely.

---

<sup>100</sup>[https://en.wikipedia.org/wiki/CAP\\_theorem](https://en.wikipedia.org/wiki/CAP_theorem)

## Reasons for the CAP Theorem

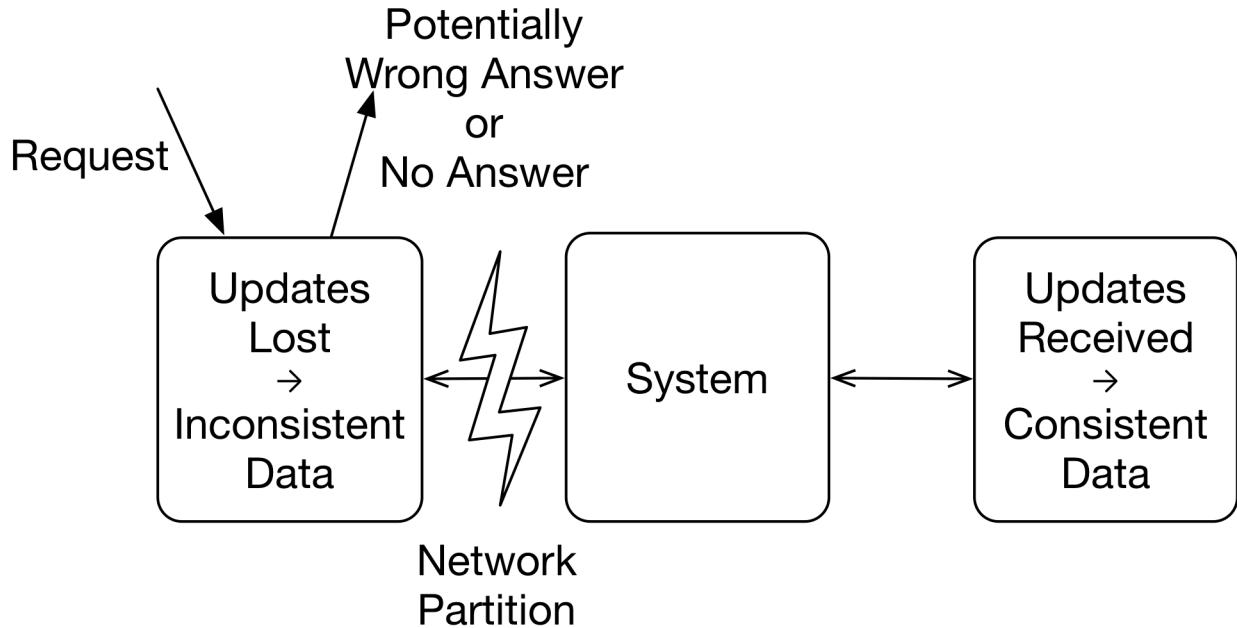


Fig. 10-6: If Communication Fails, a System Can Either Return a Potentially Wrong Response Upon a Request (AP) or None at All (CP).

There are only two options as to how a system can react to a request when the network is partitioned (see [figure 10-6](#)).

- The system provides *a response*. In this instance, the response can be wrong because changes have not reached the system; this is the *AP* case. The system is available. However, it might return a different response from systems that have obtained newer information. Thus, inconsistencies exist.
- Alternatively, the system returns *no response*; this is the *CP* case. On the one hand, the system is not available when there is a problem. On the other hand, all systems always return the same responses and therefore are consistent as long as there is no network partitioning.

## Compromises with CAP

Of course, you can make compromises. Let's take a system with five replicas. When writing, each replica confirms that the data has actually been written. When reading, several systems can be called to find out the latest state of the data, if not all systems have completed the replication yet.

Such a system with five replicas, in which one replica is read and only the confirmation from one replica is waited for, focuses on availability. Up to four nodes can fail without the system failing. However, it does not guarantee high consistency: The data could possibly be written to one node and – due to the time for replication to other nodes to occur – an old value is read from another node.

If the system with five replicas always waits for five nodes to be confirmed and always reads from five nodes, the data is always consistent. However, the failure of a single node causes the system to become unavailable. A compromise can be to wait for confirmation of writes from three nodes and for reads from three nodes. In this way, inconsistencies can still be ruled out and the failure of up to two nodes can be compensated for.

## CAP, Events, and Data Replication

The CAP theorem actually considers data storage like NoSQL databases, which achieve performance and reliability via replication. But similar effects also occur when systems use events or data replication. Ultimately, an event can be seen as a kind of data replication across multiple microservices. However, unlike the full replication of data between nodes of NoSQL databases, each microservice can react differently to the event and may use only parts of the data. A microservice that relies on asynchronous communication, events, and data replication corresponds to an AP system. Microservices may not have received some events yet, so the data may be inconsistent. Nevertheless, the system can process requests using local data and is therefore available even if other systems fail. The CAP theorem says that the only alternative is a CP system. This would be consistent but not available. For example, it could store the data in a central microservice which is accessible by all. As a result, all microservices would receive the latest data. However, if the central microservice fails, all other microservices would no longer be available.

## Are Inconsistencies Acceptable?

Thus, the inconsistency of an asynchronous system is inevitable unless you want to give up availability. It is therefore important to know the requirements for consistency, which requires some skill. Customers want a reliable system. Data inconsistency seems to contradict this. That's why it is important to know what happens when the data is temporarily inconsistent and whether this really causes problems. After all, the inconsistencies should usually disappear after fractions of a second or a few seconds. Besides, certain inconsistencies can even be tolerable from a domain perspective. For example, if goods are listed days before the first sale, inconsistencies are initially acceptable and must only be corrected when the goods are finally being sold.

If inconsistencies are not acceptable at all, asynchronous communication is not an option. This means that synchronous communication must be used with all its disadvantages. If the tolerance for temporarily inconsistent data is not known, this can lead to a wrong decision regarding the communication variant.

## Repairing Inconsistencies

In the simplest case, the inconsistencies disappear as soon as all events have reached all systems. However, there may be exceptions. An example: After registration, a customer receives an initial credit balance. A microservice receives the event for the initial balance, but it has not yet received the registration of the user. The microservice cannot credit the initial balance because the customer

has not been created in the system. If the registration of the customer arrives later on, the initial balance would have to be executed once again.

This problem can be solved when the order of events can be guaranteed. In this case, the problem will not arise in the first place. Unfortunately, many solutions cannot guarantee the order of events.

Event sourcing can also help. This allows the microservice to always reconstruct its state from the events. Therefore, the state could be discarded and recreated from the events as long as those are available without any gaps.

It is also possible to extend the domain logic. In that case, if the event for the initial balance is received before the registration, a new customer object is created, but the object is marked as invalid as long as the data from the registration is missing. The rest of the logic would need to handle such invalid customers. That might make the business logic quite complex. Error might be hard to spot because now there are so many states that an object might have. Therefore, this solution should probably be avoided.

## Guaranteed Delivery

In an asynchronous system, the delivery of messages can be guaranteed if the system is appropriately implemented. The sender has the messaging system confirm that it received the message. Afterwards, the messaging system has the recipient of the message acknowledge the receipt. However, if the recipient never picks up the data and thus prevents delivery, the sender has an acknowledgement, but the message still does not arrive at the recipient.

It is difficult to guarantee delivery when the recipient is anonymous. In this case, it is unclear who is supposed to receive the message and whether there are any recipients at all who should get the message. Therefore, it is also unclear who has to issue receipts.

## Idempotency

If the messages are not acknowledged by the recipient, they are sent again. When the receiving microservice processes the message, but is unable to acknowledge the message due to a problem or a failure, the recipient receives the message a second time although the recipient processed the message already.

This is an *at least once strategy*. The messages are sent at least once, and, in the described failure scenario, more often.

Therefore, typically, one tries to design distributed systems in such a way that the microservices are *idempotent*. This means that a message can be processed more than once, but the state of the service no longer changes. For example, when creating an invoice, the invoice microservice can first check in its own database whether an invoice has already been created. In this manner, an invoice is created only the first time the message is received. If the message is transferred again, it will be ignored.

## One Recipient

In addition, it can be necessary that only one instance of a microservice processes a message. For example, it would be incorrect from a domain perspective when multiple instances of the invoice microservice receive the order and all of them generate an invoice. This would generate multiple invoices rather than one. For this, messaging systems normally have an option to send messages only to a single recipient. This recipient then has to confirm the message and process it. Such a communication type is termed *point to point communication*.

Unfortunately, the rules for processing can be complex. When changes are made to customer data, parallel processing should be carried out as far as possible to ensure high performance. However, changes to the data of a specific customer probably have to follow a sequence. For example, it would not be good if changes to the billing address are processed after the invoice has been written; the invoice would still contain the wrong address. For this reason, it may be important to guarantee the order of messages.

## Test

Also with asynchronous microservices, the continuous delivery pipelines must be independent to enable independent deployment. To do this, the testing of the microservices must be independent of other microservices.

With asynchronous communication, a test can send a message to the microservice and check whether the system behaves as expected. Timing can be difficult because it is not clear when the microservice has processed the message and how long the test should wait for processing. The test can then check whether the microservice sends the correct messages in response. This allows very simple black box tests, – a test based on the interface without knowing about the internal structure of the microservice. In addition, such tests do not place particularly high demands on the test environment. The environment just needs to be able to transmit messages. In particular, it is not necessary to install a large number of other microservices in the test environment. Instead, the messages that other microservices send or expect from the tested microservice can be the basis for the tests.

## 10.4 Advantages

Decoupling via events was presented in [section 10.2](#). Such an architecture achieves a high degree of decoupling.

Especially for distributed systems, asynchronous communication has a number of decisive advantages:

- When a communication partner fails, the message is sent later when the communication partner is available again. In this manner, asynchronous communication offers *resilience*, – that is, a protection against the failure of parts of the system.

- Delivery and also processing of a message can nearly always be *guaranteed*. The messages are stored for a long time. Processing is assured, for example, by the recipients *acknowledging* the message.

In this manner, asynchronous communication solves challenges caused by distributed systems.

## 10.5 Variations

The two following chapters introduce concrete technologies for implementing asynchronous communication.

- [Chapter 11](#) shows Apache Kafka as an example for a message-oriented middleware (MOM). Kafka offers the option to store messages for a very long time. This can be helpful for event sourcing. This feature distinguishes Kafka from other MOMs which are also good options for microservices.
- [Chapter 12](#) demonstrates the implementation of asynchronous communication with REST and the Atom data format. This can be helpful when MOMs are too much of an effort as additional infrastructure.

Asynchronous communication is easy to combine with frontend integration (see [chapter 7](#)) because both of these integrations focus on different levels: frontend and logic. However, inconsistencies easily occur during UI integration as when two microservices simultaneously present their state on one web page. When the microservices implement things of different domains, they use different data and therefore are rarely inconsistent.

However, a combination of asynchronous and synchronous communication (see [section 13](#)) should be avoided because synchronous and asynchronous communication both start at the logic level. However, even this combination might be sensible in special scenarios. For example, synchronous communication can be necessary if a response of a microservice is required immediately.

## 10.6 Conclusions

Asynchronous communication should be preferred over synchronous communication between microservices due to the advantages concerning resilience and decoupling. The only reason arguing against this is inconsistency. Therefore, it is important to know exactly what the requirements are, especially concerning consistency, in order to make the technically correct decision. Choosing asynchronous communication has the potential to elegantly solve the essential challenges of the microservices architecture and should therefore be considered in any case.

# 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<sup>101</sup>](#) (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<sup>102</sup>](#) or [IBM MQ<sup>103</sup>](#), which was previously known as IBM MQSeries. However, many more JMS products<sup>104</sup> are available. 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<sup>105</sup>](#) (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<sup>106</sup>](#), [Apache ActiveMQ<sup>107</sup>](#), and [Apache Qpid<sup>108</sup>](#) are the best known implementations of the AMQP standard. There are also a lot more [implementations<sup>109</sup>](#).

In addition, there is [ZeroMQ<sup>110</sup>](#), which does not comply with any of the standards. And [MQTT<sup>111</sup>](#) 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 is 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<sup>112</sup>](#) 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 (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.

<sup>101</sup><https://jcp.org/aboutJava/communityprocess/final/jsr914/index.html>

<sup>102</sup><http://activemq.apache.org/>

<sup>103</sup><http://www-03.ibm.com/software/products/en/ibm-mq>

<sup>104</sup>[https://en.wikipedia.org/wiki/Java\\_Message\\_Service#Provider\\_implementations](https://en.wikipedia.org/wiki/Java_Message_Service#Provider_implementations)

<sup>105</sup><https://www.amqp.org/>

<sup>106</sup><https://www.rabbitmq.com/>

<sup>107</sup><http://activemq.apache.org/>

<sup>108</sup><https://qpid.apache.org/>

<sup>109</sup>[https://en.wikipedia.org/wiki/Advanced\\_Message\\_Queueing\\_Protocol#Implementations](https://en.wikipedia.org/wiki/Advanced_Message_Queueing_Protocol#Implementations)

<sup>110</sup><http://zeromq.org/>

<sup>111</sup><http://mqtt.org/>

<sup>112</sup><https://kafka.apache.org/>

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<sup>113</sup>](#) 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.”

---

<sup>113</sup><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<sup>114</sup>](#). 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.

---

<sup>114</sup><http://www.confluent.io/blog/how-to-choose-the-number-of-topics-partitions-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*<sup>115</sup> – 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

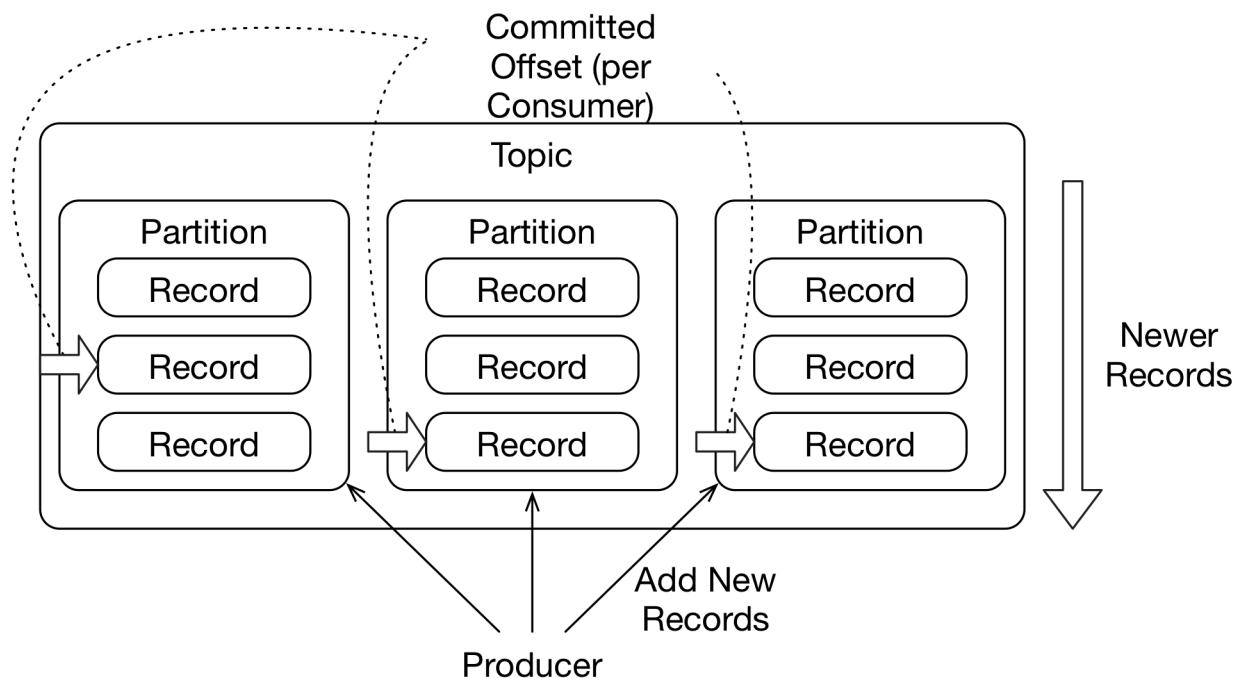


Fig. 11-1: Partitions and Topics in Kafka

<sup>115</sup><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.

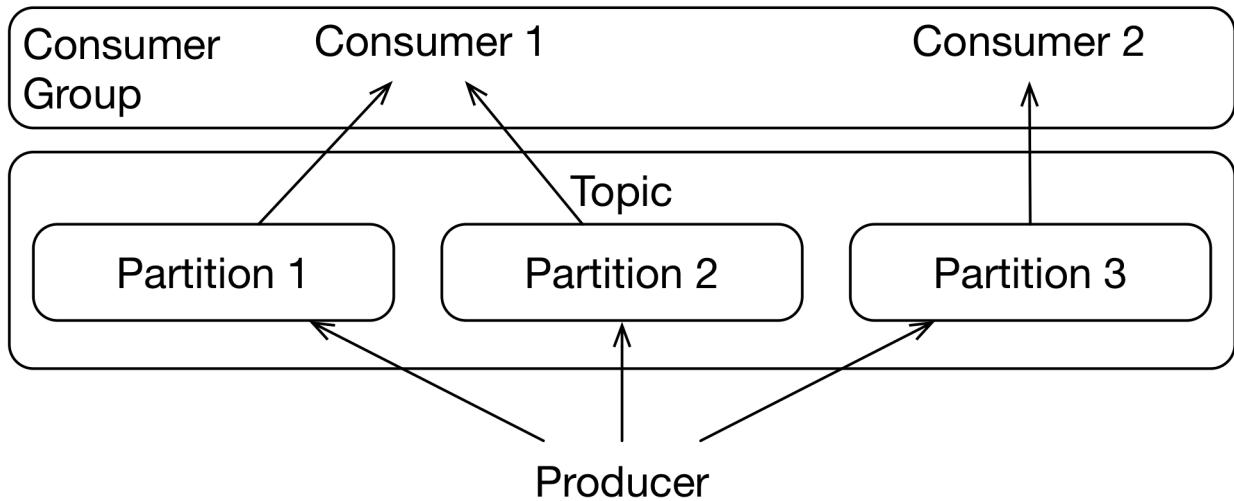


Fig. 11-2: Consumer Groups and Partitions

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

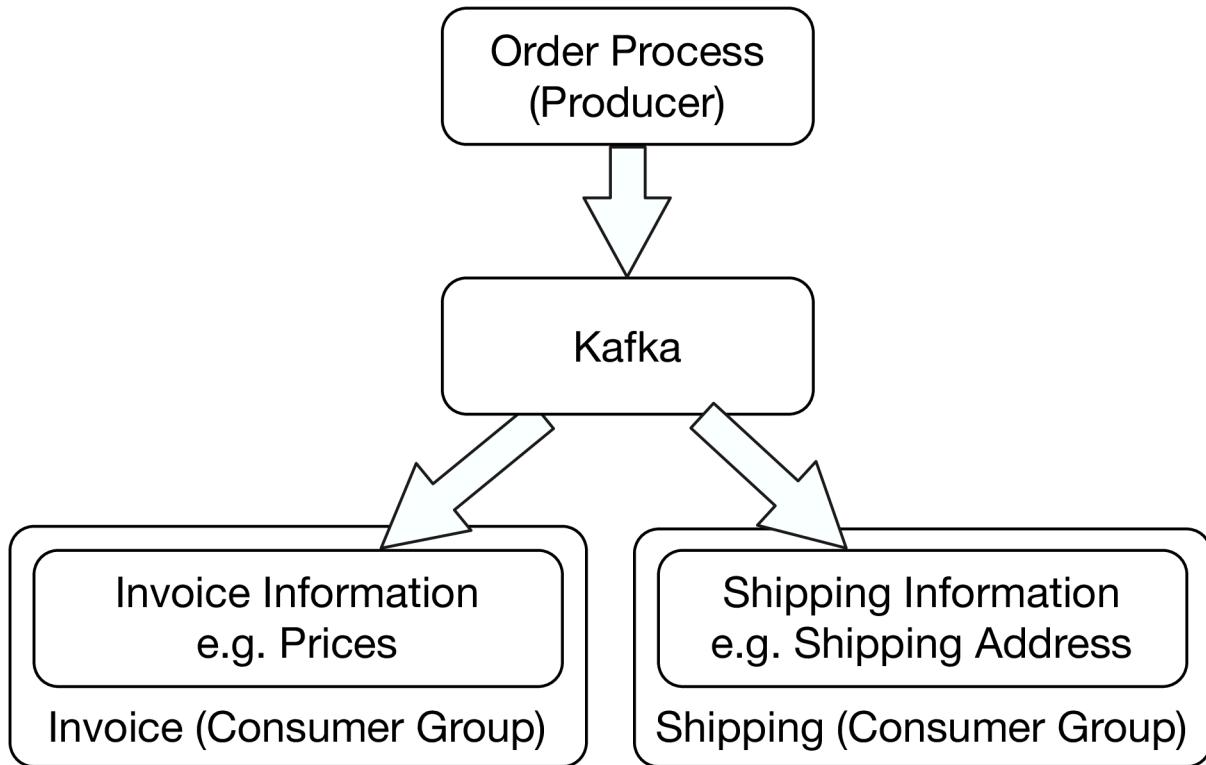


Fig. 11-3: Example for Kafka

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

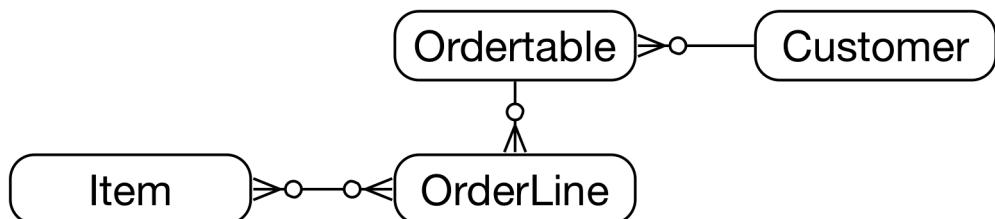


Fig. 11-4: Data Model in the System *microservice-kafka-order*

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.

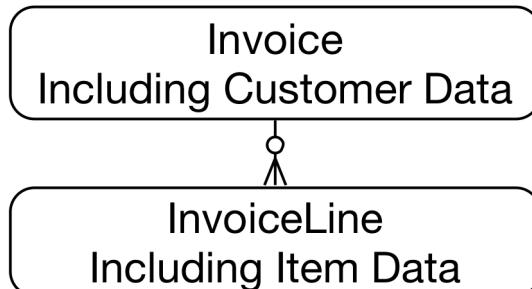


Fig. 11-5: Data Model in the System *microservice-kafka-order*

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

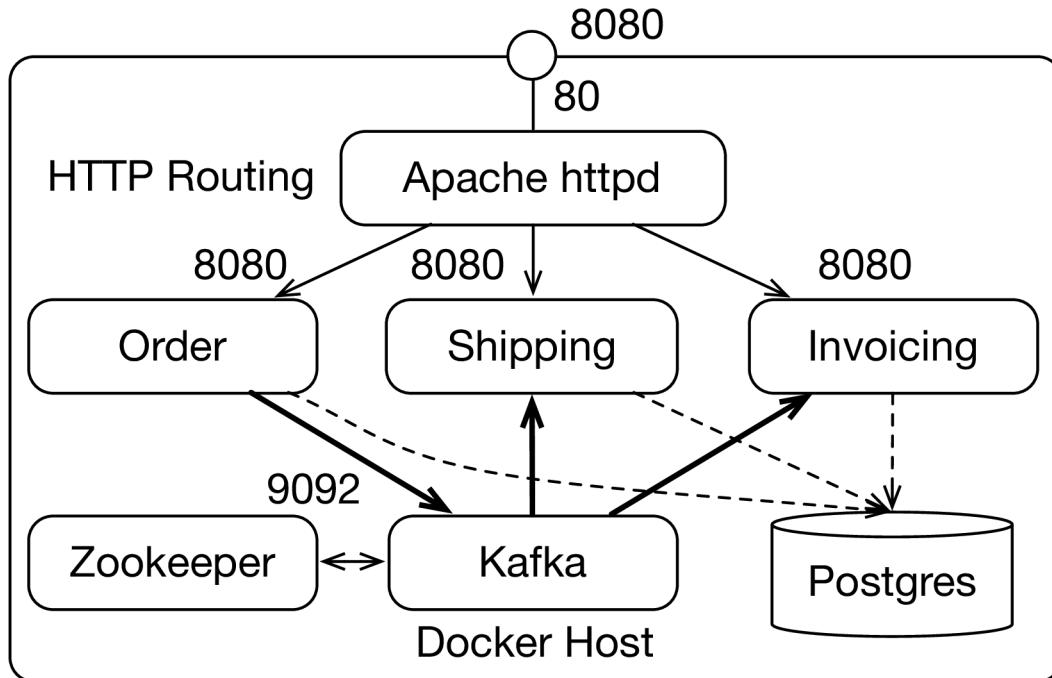


Fig. 11-6: Overview of the Kafka System

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<sup>116</sup>](#). 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

---

<sup>116</sup><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.

```

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`.

```

@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 `Acknowledgment`. 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.

```
@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<sup>117</sup> is a data format quite frequently used together with Kafka<sup>118</sup> 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<sup>119</sup>. 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.

---

<sup>117</sup><http://avro.apache.org/>

<sup>118</sup><https://www.confluent.io/blog/avro-kafka-data/>

<sup>119</sup><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<sup>120</sup>](#). 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<sup>121</sup>](#) 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](#)).

---

<sup>120</sup><http://avro.apache.org/>

<sup>121</sup><https://kafka.apache.org/documentation/#compaction>

# 12 Recipe: Asynchronous Communication with Atom and REST

This chapter deals with the integration of microservices based on the Atom data format. An example<sup>122</sup> is provided which uses a simple scenario to illustrate what an integration with the help of Atom and REST can look like.

The readers learn:

- What the Atom format is, how it functions, and how it can be exploited for the asynchronous communication of microservices.
- What alternatives to Atom exist for the asynchronous communication via REST/HTTP and what advantages and disadvantages the different formats have.
- How HTTP and REST can be efficiently used, not only for asynchronous communication.
- What an implementation of an asynchronous system based on HTTP, REST, and Atom can look like.

## 12.1 The Atom Format

Atom is a data format originally developed to make blogs accessible to readers. An Atom document contains blog posts that subscribers can read in a chronological order. Besides blogs, Atom can also be used for podcasts. Because the protocol is so flexible, it is also suitable for other types of data. For example, a microservice can offer information about new orders to other microservices as an Atom feed. This corresponds to the REST approach, which uses established web protocols such as HTTP for the integration of applications.

Atom<sup>123</sup> is no protocol, but just a data format. A GET request to a new URL such as <http://innoq.com/order/feed> can return a document with orders in the Atom format. This document can contain links to the details of the orders.

### MIME Type

HTTP-based communication indicates the type of content with the help of MIME types (Multi-purpose Internet Mail Extensions). The MIME type for Atom is `application/atom+xml`. Thanks to content negotiation, other data representation can be offered in addition to Atom for the same URL. Content negotiation is built into HTTP. The HTTP client signals – via an accept header – what data

<sup>122</sup><https://github.com/ewolff/microservice-atom>

<sup>123</sup><https://validator.w3.org/feed/docs/atom.html>

formats it can process. The server then sends a response in a suitable format. Thus, accept headers enable a client to request all orders as JSON or the last changes as an Atom feed under the same URL.

## Feed

```
<?xml version="1.0" encoding="UTF 8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>Order</title>
  <link rel="self" href="http://localhost:8080/feed" />
  <author>
    <name>Big Money Online Commerce Inc.</name>
  </author>
  <subtitle>List of all orders</subtitle>
  <id>tag:ewolff.com/microservice-atom/order</id>
  <updated>2017 04 20T15:28:50Z</updated>
  <entry>
    ...
  </entry>
</feed>
```

A document in Atom format is called an Atom feed. An Atom feed contains different components. First, the feed defines metadata. The Atom standard defines that this metadata has to comprise a number of elements.

- *id* is a globally unambiguous and a permanent URI (Uniform Resource Identifier). It has to identify the feed. In the listing, this is done via a [tag URI<sup>124</sup>](#).
- *title* is the title of a feed in a human-readable form.
- *updated* contains the point in time when a feed has been last changed. This information is important for finding out whether new data exists.
- There also has to be an *author* (including *name*, *email*, and *uri*). This is, of course, very helpful for blogs. However, it does not seem to make sense for Atom feeds, which only transport data. However, it can be useful for indicating a contact persons in case of problems.
- Multiple *link* elements are recommended. Each element has an attribute called *rel* for indicating the relationship between the feed and the link. The attribute *href* contains the actual link. A *link* can be used to provide a link to a HTML representation of the data. In addition, the feed can use a *self* link to indicate at which URL it is available.

Additional optional metadata includes:

- *category* narrows down the topic area of the feed, for example, to a field such as sports. Of course, this does not make a lot of sense for data.

---

<sup>124</sup>[https://en.wikipedia.org/wiki/Tag\\_URI\\_scheme](https://en.wikipedia.org/wiki/Tag_URI_scheme)

- Analogous to *author*, *contributor* contains information about people who contribute to the feed.
- *generator* indicates the software which produced the feed.
- *icon* is a small icon, whereas *logo* represents a larger icon. This makes it possible to represent a blog or podcast on the desktop. It is not so useful for a data feed.
- *rights* defines, for example, the copyright. This element is likewise mostly interesting for blogs or podcasts.
- Finally, *subtitle* is a human-readable description of the feed.

As the listing illustrates, the fields *id*, *title*, and *updated* are enough for an Atom feed with data. Having a *subtitle* for documentation is helpful. There should also be a *link* for documenting the URL of the feed. All other elements are not really needed.

## Entry

In the previous example, the most important thing is missing: the feed entries. Such an entry adheres to the following format:

```
<entry>
  <title>Order 1</title>
  <id>tag:ewolff.com/microservice-atom/order/1</id>
  <updated>2017 04 20T15:27:58Z</updated>
  <content type="application/json"
    src="http://localhost:8080/order/1" />
  <summary>This is the order 1</summary>
</entry>
```

An entry contained in the feed must consist of the following data according to the Atom standard:

- *id* is the globally unambiguous ID as URI. Thus, there cannot be a second entry with the id `tag:ewolff.com/microservice-atom/order/1` in this feed. This URI is a [tag URI<sup>125</sup>](#) meant for such globally unambiguous identifiers.
- *title* is a human-readable title for the entry.
- *updated* is the timepoint when the entry has last been changed. It has to be set so that the client can decide whether it already knows the last state of a certain entry.

In addition, the following elements are recommended:

- As described for the feed, *author* names the author of the entry.
- *link* can contain links – for example, *alternate* as link to the actual entry.

---

<sup>125</sup>[https://en.wikipedia.org/wiki/Tag\\_URI\\_scheme](https://en.wikipedia.org/wiki/Tag_URI_scheme)

- *summary* is a summary of the content. It has to be provided if the content is only accessible via a link or if it does not have an XML media type. This is the case in the example.
- *content* can be the complete content of the entries or a link to the actual content. To enable access to the data of the entry, either *content* has to be offered, or a *link* with *alternate*.
- In addition, *category* and *contributor* are optional, analogous to the respective elements of the feed. *published* can indicate the first date of publication. The element *rights* can state the rights. *source* can name the original source if the entry was a copy from another feed.

In the example, the elements *id*, *title*, *summary*, and *updated* are filled. Access to the data is possible via a link in *content*. The data could also directly be contained in the *content* element in the document. However, in that case, the document would be very large. Thanks to the links, the document remains small. Each client has to access the additional data via links, and can limit itself to only the relevant data for a respective client.

## Tools

At [https://validator.w3.org/feed/#validate\\_by\\_input](https://validator.w3.org/feed/#validate_by_input), a validator is provided that can check whether an Atom feed is valid – that is, all necessary elements are present.

There are systems such as [Atom Hopper<sup>126</sup>](#) that offer a server with the Atom format. In this way, an application does not have to generate Atom data, but can post new data to the server. The server then converts the information into Atom. Clients can fetch the Atom data from the server.

## Efficient Polling of Atom Feeds

Asynchronous communication with Atom requires that the client regularly requests the data from the server and processes new data. This is called polling.

The client can periodically read out the feed and check the *updated* field in the feed to see if the data has changed. If this is the case, the client can use the *updated* elements of the entries to find out which entries are new and process them.

This is very time-consuming because the entire feed has to be generated and transmitted. Additionally, only a few entries are read from the feed. Most requests will show that there are actually no new entries. For this purpose, it does not make sense to create and transfer the complete feed.

## HTTP Caching

A very simple way to solve this problem is HTTP caching (see [figure 12-1](#)).

HTTP provides a header with the name *Last-Modified* in the HTTP response. This header indicates when the data was last changed. This header takes over the function of the *updated* field from the feed.

---

<sup>126</sup><http://atomhopper.org/>

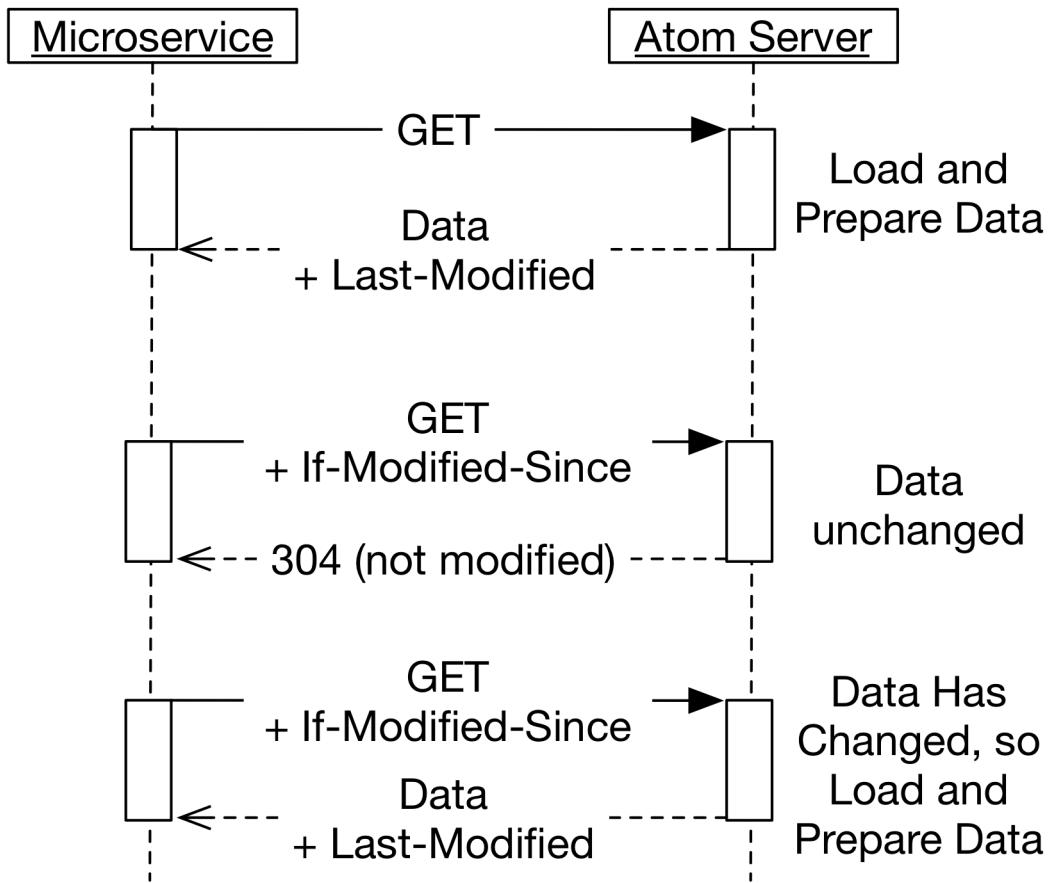


Fig. 12-1: Efficient Pooling with HTTP Caching

The client stores the value of this header. On the next HTTP GET request, the client sends the read value in the *If-Modified-Since* header with the GET request. The server can now respond with an HTTP status of 304 (not modified) if the data has not changed. Then, no data is transferred except the status code.

Whether data has changed can often be determined very easily. For example, you can implement code that determines the time of the last change in the database. This is much more efficient than converting all data into an Atom representation.

If the data has actually changed, the server responds normally with a status of 200 (OK). Also, a new value is sent in the *Last-Modified* header so that the client can use HTTP caching again.

The demo implements such an approach. For a request with a set *If-Modified-Since* header, the database is used to determine the time of the last change. This is compared with the value from the header. An HTTP status 304 is returned if no data has changed. In this case, only one database query is required to respond to the HTTP request.

An alternative would be to create the Atom feed once and store it on a web server as a static resource. In this case, dynamic generation is performed once. This approach also works efficiently for very large feeds. In that case, it would be up to the web server to implement the HTTP caching for the

static resource.

## ETags

Another approach would be HTTP caching with ETags<sup>127</sup>. Here, the server returns an ETag together with the data. The ETag can be compared to a version number or checksum. For any further requests, the client sends the ETag along. The server uses the ETag to determine whether the data has changed in the meantime, and provides data only if new data is available. In the example, however, it is much easier to find out whether new orders have been accepted since a certain point in time. It therefore does not make sense to use ETags in the example.

## Pagination and Filtering

If a client is not interested in all events, it can signal this by setting parameters in the URL. This makes it possible to paginate, for example, with a URL like <http://ewolff.com/order?from=23&to=42>. The events can be filtered as well: <http://ewolff.com/order?name=Wolff>. Of course, pagination and filtering can be combined with caching. However, if each client uses its own pagination and filters, caching on the server side can become inefficient, because too much different data has to be stored for the different clients. Therefore, it may be necessary to restrict the pagination and filtering options in order to increase the efficiency of the cache.

## Push vs. Pull

HTTP optimizations such as conditional GETs can significantly speed up communication. But they remain pull mechanisms. The client queries the server, whereas, in case of a push mechanism, the client is actively notified by the server about changes. The push model seems to be more efficient, but pulls have the advantage that the client requests new data when it can actually process it. This prevents the client from handling requests if it is under too much load.

## Guaranteed Delivery

Atom via HTTP cannot guarantee the delivery of the data. The server only provides the data. Whether it will be read at all is an open question. Monitoring can help to identify problems and remedy them if necessary. So it would be very unusual if the Atom resources are never needed. However, the HTTP protocol does not have any measures for issuing receipt acknowledgements.

## Old Events

In principle, an Atom feed can contain all events that have ever occurred. As mentioned in section 10.2, this might be interesting for event sourcing, in which case a microservice can reconstruct its state by processing all old events again.

---

<sup>127</sup>[https://en.wikipedia.org/wiki/HTTP\\_ETag](https://en.wikipedia.org/wiki/HTTP_ETag)

If the data on old events is already stored in a microservice, the microservice needs only to make it available. For example, if a service already contains all orders, it can offer this information additionally as an Atom feed if necessary. In this case, no additional storage of old events is necessary. Thus, the effort to make the old events accessible is very low.

## 12.2 Example

The example can be found at <https://github.com/ewolff/microservice-atom>.

The example for Atom is analogous to the example in the Kafka chapter ([section 11.4](#)) and is based on the example for events from [section 10.2](#). The order system creates orders. Based on the data in the order, the invoicing microservice creates invoices, and the shipping microservice creates deliveries. The data models and database schemas are identical to the Kafka example. Only the communication is now done via Atom.

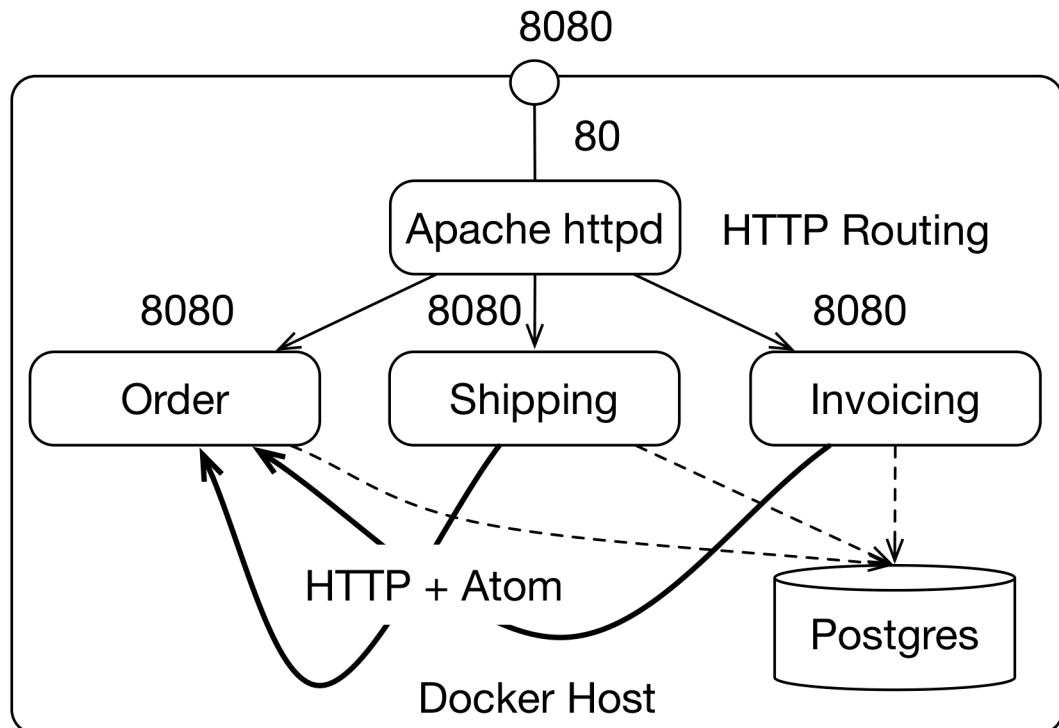


Fig. 12-2: Overview of the Atom System

[Figure 12-2](#) shows how the example is structured:

- The *Apache httpd* distributes calls to the microservices. For this purpose, the *Apache httpd* uses Docker Compose service links. Docker Compose offers simple load balancing. The *Apache httpd* uses the load balancing of Docker Compose to forward external calls to one of the microservice instances.

- The *order microservice* offers an Atom feed from which the invoicing and shipping microservice can read the information about new orders.
- All microservices use the same *Postgres database*. Within the database, each microservice has its own separate database schema. Thus, the microservices are completely independent regarding the database schema. At the same time, one database instance is enough to run all microservices.

Section 0.4 describes what software needs to be installed to start the example. The example can be found at <https://github.com/ewolff/microservice-atom>. You first have to download the code with `git clone https://github.com/ewolff/microservice-atom.git` to start the example. Then you have to execute the command `./mvnw clean package` (macOS, Linux) or `mvnw.cmd clean package` (Windows) in the directory `microservice-atom`. See appendix B for more details on Maven and how to troubleshoot the build. Finally, you have to execute `docker-compose build` in the directory `docker` to create the Docker images and `docker-compose up -d` to start the environment. See appendix C for more details on Docker, Docker Compose, and how to troubleshoot them. Afterwards, the Apache httpd load balancer will be available at port 8080 – that is, at <http://localhost:8080/>, for example. From there you can use the other microservices to add new orders that will eventually also appear in the invoicing and shipping microservice.

<https://github.com/ewolff/microservice-atom/blob/master/HOW-TO-RUN.md> describes the necessary steps in detail for running the example.

## Implementation of the Atom View

The class `OrderAtomFeedView` in the project `microservice-atom-order` implements the Atom feed as a view with the framework Spring MVC. Spring MVC splits the system into MVC (model, view, controller). The *controller* contains the logic, the *model* the data, and the *view* displays the data from the model. Spring MVC offers support for a plethora of view technologies for HTML. For Atom, Spring uses the [Rome Library](#)<sup>128</sup>. It offers different Java objects to display data as feeds and entries in the feeds.

## Implementation of the Controller

---

<sup>128</sup><https://rometools.github.io/rome/>

```
public ModelAndView orderFeed(WebRequest webRequest) {  
    if ((orderRepository.lastUpdate() != null)  
        && (webRequest.checkNotModified(orderRepository.lastUpdate().getTime()))) {  
        return null;  
    }  
    return new ModelAndView(new OrderAtomFeedView(orderRepository),  
                           "orders", orderRepository.findAll());  
}
```

The method `orderFeed()` in the class `OrderController` is responsible for displaying the Atom feed with the help of `OrderAtomFeedView`. As shown in the listing, `OrderAtomFeedView` is returned as the view and a list of the orders as the model. The view then displays the orders from the model in the feed.

## Implementation of HTTP Caching on the Server

Spring provides the `checkNotModified()` method in the `WebRequest` class to simplify the handling of HTTP caching. The time of the last update is passed to the method. The `lastUpdate()` method of the class `OrderRepository` determines this time point with a database query. Each order contains the time at which it was placed. `lastUpdate()` returns the maximum value. `checkNotModified()` compares this passed value with the value from the `If-Modified-Since` header in the request. If no more-recent data needs to be returned, the method returns true. Then, `orderFeed()` returns `null`, so that Spring MVC returns an HTTP status code 304 (Not Modified). So in this case, the server just makes a simple query to the database and returns an HTTP response with a status code. It does not provide any data.

## Implementation of HTTP Caching on the Client

On the client side, HTTP caching must of course also be implemented. The microservices `microservice-order-invoicing` and `microservice-order-shipping` implement the polling of the Atom feed in the method `pollInternal()` of the class `OrderPoller`. They set the `If-Modified-Since` header in the request (see line 4/5 in the listing). The value is determined from the variable `lastModified`. It contains the value of the `Last-Modified` header of the last HTTP response (line 15-17). If no data has been changed in the meantime, the server responds to the GET request directly with an HTTP status 304 and it is clear that no new data exists. Accordingly, data is processed only if the status is not `NOT_MODIFIED` (line 11).

```

1 public void pollInternal() {
2     HttpHeaders requestHeaders = new HttpHeaders();
3     if (lastModified != null) {
4         requestHeaders.set(HttpHeaders.IF_MODIFIED_SINCE,
5             DateUtils.formatDate(lastModified));
6     }
7     HttpEntity<?> requestEntity = new HttpEntity(requestHeaders);
8     ResponseEntity<Feed> response =
9         restTemplate.exchange(url, HttpMethod.GET, requestEntity, Feed.class);
10
11    if (response.getStatusCode() != HttpStatus.NOT_MODIFIED) {
12        Feed feed = response.getBody();
13        ...          // evaluate feed data
14        if (response.getHeaders().getFirst(HttpHeaders.LAST_MODIFIED) != null) {
15            lastModified =
16                DateUtils.parseDate(
17                    response.getHeaders().getFirst(HttpHeaders.LAST_MODIFIED));
18        }
19    }
20 }

```

`pollInternal()` is called by the method `poll()` in the class `OrderPoller`. The user can call this method with a button in the web UI. In addition, the microservice calls the method every 30 seconds because of the `@Scheduled` annotation.

## Data Processing and Scaling

If there are multiple instances of the invoicing and shipping microservices, each instance polls the Atom feed and processes the data. Of course, it is not correct that several instances write an invoice for an order or initiate a delivery because then an order would create multiple invoices or deliveries. Therefore, each instance must determine what orders are already processed and what data is in the database. If another instance has already created the data record for the invoice or delivery of the order, then the entry from the Atom feed for the order must be ignored. To do this, `ShippingService` and `InvoicingService` use a transaction in which a data record is first searched for in the database. A new data record is written only if none yet exists. Therefore, only one instance of the microservices can write the data record. All others read the data and find out that another instance has already written a data record. With a very large number of instances, this can cause a considerable load on the database. In return, the services are idempotent. No matter how often they are called, the state in the database in the end is always the same.

## Atom Cannot Send Data to a Single Recipient

This is a disadvantage of Atom. It is not easily possible to send a message to exactly one instance of a microservice. Instead, the application has to deal with multiple instances reading the message

from the Atom feed. Thus, especially when a lot of point-to-point communication is necessary, the Atom approach can be disadvantageous.

The application must also be able to deal with messages not being processed. If a message has been read, the process can fail before the message has been processed. As a result, no data might be written for some messages. In this case, however, another instance of the microservice would eventually read the message and process it. So retries are actually quite easy to implement with Atom.

## 12.3 Recipe Variations

Instead of Atom, a different format can be used for the communication of changes via HTTP.

### RSS

Atom is just one format for feeds. An alternative is RSS<sup>129</sup>. There are different versions of RSS. RSS is older than Atom. Atom has learnt from RSS and represents the more modern alternative. Blogs and podcasts should offer feeds as RSS and Atom to reach as many clients as possible. Because server and client are under the control of the same developer in case of microservices, it is not necessary to support multiple protocols. Therefore, Atom is the better and more modern alternative.

### JSON Feed

Another alternative is JSON Feed<sup>130</sup>. It also defines a data format for a feed. However, it uses JSON rather than XML, which is used by Atom and RSS.

### Custom Data Format

Atom and RSS are only formats for communicating changes. Some of the elements are not useful for data, but only for blogs or podcasts. The useful part of Atom and RSS is the list of changes and the links to the actual data. Atom and RSS therefore use hypermedia to communicate the changes without delivering all data.

Of course it's feasible to define your own data format, which contains the changes and links to the data. In addition to the links, the data can also be embedded directly into the document.

Compared to Atom and RSS, a custom data format has the disadvantage in that it is not standardized. For a standardized data model, libraries are available and learning about the format is easier. There are also tools such as validators. A user can read and display Atom data with an Atom reader, which is useful, among other things, for troubleshooting.

However, the data to be transported is not very complex, and so a custom data format has no major disadvantages. In essence, even with Atom, the approach simply uses hypermedia as an essential

<sup>129</sup><http://web.resource.org/rss/1.0/spec>

<sup>130</sup><http://jsonfeed.org/>

component of REST. It provides a list of links that clients can use to get more information about the changes to the data. This procedure can also be easily implemented with a custom data format.

The example in [section 23](#) uses a custom data format based on JSON. The data format is not very complex. It contains just a list of links to the details of the individual orders.

## Alternatives to HTTP

HTTP supports features such as scaling or reliability very well. Most applications already use HTTP even without Atom to deliver web pages or provide REST services. The alternative to HTTP would be a messaging system such as Kafka (see [chapter 11](#)), which can also be used to implement asynchronous communication. Such messaging systems must be scalable and have to provide high availability. The messaging systems offer these features in principle, but must be tuned and configured accordingly. This is especially challenging if you have never operated such a messaging system before. In particular, the advantages of HTTP concerning operation argue for using this protocol also for asynchronous communication.

## Including Event Data

Of course, the feed can also include the event data rather than just links. In this way, a client can work with the data without further requests to load the linked data; but the feed gets bigger. The question also arises as to what data should be included in the feed. Each client may need different data, which makes it difficult to model the data. Sending links has the advantage that the client can select the appropriate representation of the data with content negotiation .

## 12.4 Experiments

- Start the system and examine the logs of *microservice-order-invoicing* and *microservice-order-shipping* with `docker logs -f msatom_invoicing_1` respectively `docker logs -f msatom_shipping_1`. The microservices log messages when they poll data from the Atom feed, because there are new orders. If you start additional instances of a microservice with `docker-compose up --scale`, these new instances will also collect orders via the Atom feed and log information about them. In doing so, only one instance writes at a time; the other ones ignore the data. Create orders and notice this behavior based on the log messages. Explore the code to find out what the log messages mean exactly and where they are put out.
- Supplement the system with an additional microservice.
  - As an example, a microservice can be used that credits the customer with a bonus depending on the value of the order or that counts the orders.
  - Of course, you can copy and modify one of the existing microservices.
  - Implement a microservice which polls the URL `http://order:8080/feed`.
  - In addition, the microservice should display an HTML page with some information (customer bonus or number of calls).

- Package the microservice in a Docker image and reference it in `docker-compose.yml`. There you can also determine the name of the Docker container.
  - Create a link from the container `apache` to the container with the new service in `docker-compose.yml` and from the container with the new service to the container `order`.
  - The microservice has to be accessible via the home page. For this purpose, 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 for this. Then, add a link to the new load balancer in `index.html`.
  - Optional: Add HTTP caching.
- Currently, it is only possible to request all orders at once in the Atom feed. Implement paging so that only a subset of the orders is returned.
  - At the moment, the system runs with Docker Compose. However, it could also run on a different infrastructure – for example, on a microservices platform ([chapter 16](#)). [Chapter 17](#) discusses Kubernetes in more detail, and [chapter 18](#) deals with Cloud Foundry. Port the system to one of these platforms.
  - Instead of using the Atom format, you could also deliver your own representation of a feed, for example, as a JSON document. Change the implementation in the example so that it uses its own custom data format.

## 12.5 Conclusion

REST and the Atom format offer an easy way to implement asynchronous communication. HTTP is used as the communication protocol. This has several advantages. HTTP is well understood, has already proven its scalability many times, and most of the time HTTP is already used in projects anyway to transfer other content like JSON via REST or to deliver HTML pages. Therefore, most teams have the necessary experience to implement scalable systems with HTTP. Because HTTP caching is supported, polling of Atom resources can be implemented very efficiently.

Having very old events still available also helps because another microservice with event sourcing can rebuild its state by processing all events again. With an Atom-based system, a microservice must also provide old events. This can be done very easily in some cases. If the microservice has stored the old information anyway, it only needs to provide it as an Atom feed. In this case, access to old events is thus very easy to implement.

The data in the Atom Feed comes from the same source as all other representations. Therefore, the data is consistent with the data that can be queried via other means. All these services represent only different representations of the same data.

However, Atom cannot guarantee the delivery of the message. Therefore, the microservices in which the messages are processed should be implemented idempotently and try to read the message several times to ensure that they are processed.

Atom has no way to limit the reception of a message to just one microservice. Therefore, the instances of the microservice must select an instance that then processes the message, or else you have to rely on the idempotency.

Atom can guarantee the order of the messages because the feed is a linear document, and so the order is fixed. However, this requires that the order of the entries in the feed does not change.

All in all, Atom is a very simple alternative for asynchronous communication.

## Benefits

- Atom does not require additional infrastructure, just HTTP.
- Old events are easily accessible if necessary. This can be advantageous for event sourcing.
- The sequence can be guaranteed.
- The Atom feed is consistent. It is just another representation of the data and contains exactly the same information as all other representations of the data.

## Challenges

- Guaranteed delivery is difficult. The recipient can attempt to read the data several times to guarantee all data is processed. However, the responsibility for this lies with the recipient, not with the infrastructure.
- Messages for only one recipient are difficult. All recipients poll the messages. They must then decide which recipient actually processes the message.
- In part, Atom is not well suited for the representation of events because it was originally designed for blogs.

# 13 Concept: Synchronous Microservices

Synchronous microservices are one way in which microservices can communicate. This chapter shows:

- What is meant by synchronous communication between microservices.
- What the architecture of a synchronous microservices system can look like.
- What advantages and disadvantages are associated with synchronous communication between microservices.

## 13.1 Definition

This chapter deals with the technical options for implementing synchronous microservices. [Chapter 10](#) already introduced the term “synchronous microservices.”

A microservice is synchronous if it makes a request to other microservices while processing requests and waits for the result.

The logic to handle a request in the microservice might therefore not depend on the result of a request to a different microservice.

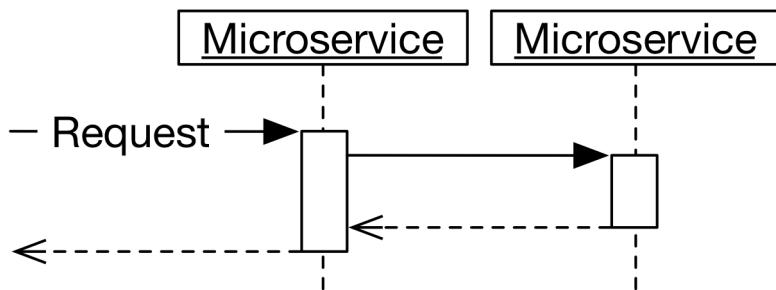


Fig. 13-1: Synchronous Communication

[Figure 13-1](#) illustrates this kind of communication. While the left microservice processes a request, it calls the right microservice and waits for the result of this call.

Synchronous and asynchronous communication according to this definition are independent of the communication protocol. A synchronous communication protocol means that a request returns a

result. For example, a REST or HTTP GET returns a result such as an HTTP status, a JSON document, or an HTML page. If a system processes a REST request, makes a REST request itself to another system and waits for the response, it is synchronous. Asynchronous REST systems are discussed in [chapter 12](#).

Asynchronous communication protocols, on the other hand, send messages to which the recipients react. There is no direct response. Synchronous communication with an asynchronous protocol occurs when one system sends a message with an asynchronous communication protocol to another system and then waits to receive a response with an asynchronous communication protocol.

## An Example

A microservice for orders is synchronous if it calls a microservice for customer data while processing an order and waits for the customer data.

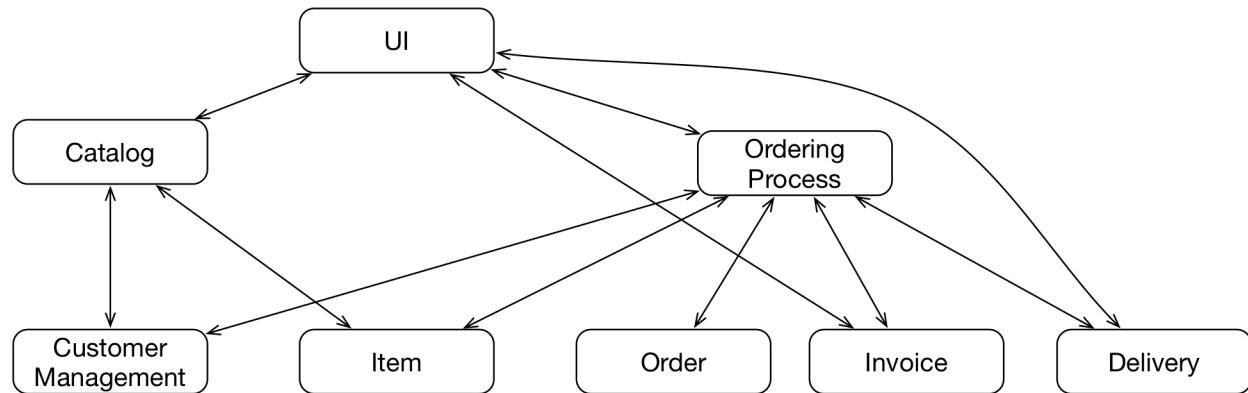


Fig. 13-2: Architecture for a Synchronous System

[Figure 13-2](#) shows an exemplary synchronous architecture, which corresponds to the asynchronous architecture displayed in [figure 10-3](#). It describes an excerpt from an e-commerce system. The microservices *customer management*, *items*, *order*, *invoice*, and *delivery* manage the respective data. The *catalog* displays all information about the goods and takes into account the customer's preferences. Finally, the *order process* serves to order goods, issue the invoice, and deliver the goods. The *UI* accesses *catalog* and *order process* and thus makes the processes implemented in these microservices available to the user. The *UI* can also display invoice and delivery data.

## Consistency

The architecture of this system ensures that all microservices display the same information about a product or order at all times, because they use synchronous calls to access the respective microservices in which the data is stored. There is only one place where each piece of the data is stored. Every microservice uses that data and therefore shows the latest data.

However, this architecture also has disadvantages. Centralized data storage can lead to problems because the data of a customer for displaying the catalog is completely different from that needed

for the order process. The customer's purchasing behavior is important for the catalog in order to display the right products. For orders, on the other hand, the delivery address or the preferred delivery service are relevant.

Storing all this data in a microservice can make the data model complex. Domain-driven design<sup>131</sup> states that a domain model is valid only in a particular bounded context (see section 2.1). Therefore, such a centralized model is problematic.

The illustration also reveals another problem: Most of the functionalities use many microservices. That complicates the system. In addition, the failure of a microservice means that many functionalities are no longer available if no special precautions are taken. Performance may also suffer because the microservices have to wait for results from many other microservices.

## Bounded Context

Such an architecture is often found in synchronous architectures. However, it is not mandatory. Bounded contexts could conceivably communicate synchronously with each other, but they typically exchange events. This is particularly easy with asynchronous communication and not a great fit for synchronous communication.

## Tests

To enable independent deployment, the tests of each microservice must be as independent as possible, and integration tests should be kept to a minimum.

For tests on synchronous systems, the communication partners must be available. These can be the microservices used in production. In this case, setting up an environment is hard because many microservices have to be available, and dependencies arise between the microservices because a new version of a microservice must be made available to all clients in order to enable tests.

## Stubs

An alternative are stubs that simulate microservices and simplify the setup of the test environment. Now the tests no longer depend on other microservices because the tests use stubs rather than microservices.

## Consumer-driven Contract Tests

Finally, consumer-driven contract tests<sup>132</sup> can be a solution. With this pattern, the client writes a test for the server. In this way, the server can test whether it meets the client's expectations. This makes it easier to change the server because changes to the interface can be tested without a client. In addition, the tests then no longer depend on the client microservice.

---

<sup>131</sup>Eric Evans: Domain Driven Design: Tackling Complexity in the Heart of Software, Addison Wesley, 2003, ISBN 978 0 32112 521 7

<sup>132</sup><https://martinfowler.com/articles/consumerDrivenContracts.html>

Consumer-driven contract tests can be written with a testing framework like JUnit, for example. The team that implements the called microservice must then execute the tests. If the tests fail, they have made an incompatible change to the microservice. Either the microservice must then be changed to be compatible, or the team that wrote the consumer-driven contract test must be informed so that they can change their microservice in such a way that the interface is used differently according to the change. Consumer-driven contract tests therefore formalize the definition of the interface.

As part of the macro architecture, all teams must agree on a test framework in which the consumer-driven contract tests are written. This framework does not necessarily have to support the programming language in which the microservices are written when the consumer-driven contract tests use the microservices as black boxes and only access them through the REST interface.

## The Pact Test Framework

One option is the framework [Pact<sup>133</sup>](#). It enables writing tests of a REST interface in a programming language. This results in a JSON file containing the REST requests and the expected responses. With Pact implementations, the JSON file can be interpreted in different programming languages. This increases the technology freedom. An example for Pact with Java can be found at <https://github.com/mvitz/pact-example>.

## 13.2 Benefits

In practice, synchronous microservices are a relatively often-used approach. Many well-known examples for microservices architectures use such a concept. This has the following advantages:

- All services can access the same dataset, so there are *less consistency problems*.
- Synchronous communication is a natural approach if the system is to offer an *API*. Any microservice can implement part of the API. The API can be the product. This is the case, for example, with a provider of payment solutions. Or the API can be used by mobile applications.
- After all, it can be easier to *migrate* into such an architecture. For example, the current architecture can already have such a division into different synchronous communication endpoints, or teams can exist for each of the functionalities.
- Calling methods, procedures, or functions in a program is usually synchronous. Developers are thus *familiar* with this model so that they can understand it more easily.

## 13.3 Challenges

Synchronous microservices create a number of challenges:

---

<sup>133</sup><https://pact.io>

- The communication with other microservices during request processing causes the latencies for responses of other microservices, and the communication times via the network to add up. Therefore, synchronous communication can lead to a performance problem. When a microservice reacts slowly, this can have consequences for a plethora of other microservices. In the example in [figure 13-2](#), the *catalog* uses two other microservices (*item* and *customer management*). Thus the latencies of these three systems can add up. A request to the *catalog microservice* creates requests to the *customer management microservice* and to the *item microservice*. Only when all microservices have responded to the requests, does the user see the result.
- When a synchronous microservice calls a failed microservice, the calling microservice may also crash and thus propagate the failure. This makes the system very vulnerable. The vulnerability of the microservices and the adding up of waiting times can prevent the reliable operation of microservices systems with synchronous communication. Therefore, these problems first have to be largely solved when a microservices system uses synchronous communication.
- In addition, synchronous communication can create a higher level of dependency in the domain logic. Asynchronous communication often focuses on events (see [section 10.2](#)). In this case, a microservice can decide how to react to events. In contrast, synchronous communication typically defines what a microservice is supposed to do. In the example, the *order process* would make the *invoice microservice* generate an invoice. With events, the *invoice microservice* would decide itself how to react to the event. This facilitates the expandability of the system. If the customer is supposed to be credited with bonus points for an order, there only has to be an additional microservice reacting to the already existing event. That event is probably already processed by more than one microservice, so just another receiver needs to be added. In a synchronous architecture, an additional system has to be called.

## Technical Solutions

Implementing a system of synchronous microservices requires some technical solutions:

- The microservices have to know how they can communicate with other microservices. Usually, this requires an IP address and a port. *Service discovery* serves to find the port and IP address of a service. Service discovery should be dynamic. If microservices are scaled, there are new IP addresses at which additional instances of a microservice are available. In addition, a microservice can fail, making it unavailable at the known IP address. Service discovery can be very simple. DNS (Domain Name System), for example, provides IP addresses for servers on the Internet. This technology already provides a simple service discovery.
- When communication is synchronous, microservices have to be prepared for the failure of other microservices. The calling microservice has to be prevented from failing as well. Otherwise, a failure cascade occurs when first one microservice fails, and other microservices that call this microservice also subsequently fail. In the end, the entire system will be down. When in [figure 13-2](#), for example, the service for the items fails, the catalog and order process could fail in turn. This would render a large part of the system unusable. Therefore, there has to be a technical solution to achieve *resilience*.

- Each microservice should be scalable independently of the other microservices. Load has to be distributed between microservices. This not only pertains to access from the outside, but also to internal communication. Therefore, there has to be a *load balancing* for each microservice.
- Finally, every access from the outside should be forwarded to the responsible microservices. This requires *routing*. A user might want to use the catalog and the order process. Although they are separate microservices, they should appear as parts of the same system to the outside.

Consequently, the technologies for synchronous microservices discussed in the following chapters have to offer solutions for service discovery, resilience, load balancing, and routing.

## API Gateways

For complex APIs, complex routing of requests to the microservices might be needed. API Gateways offer additional features. Most of the API gateways can perform user authentication. In addition, they can throttle the network traffic for individual users to support a high number of users at the same time. This can be supplemented, for example, by a centralized logging of all requests or caching. Moreover, API gateways can also solve aspects like monitoring, documentation, or mocking.

The implementations of [Apigee<sup>134</sup>](#), [3scale by Red Hat<sup>135</sup>](#), [apiman<sup>136</sup>](#), or cloud products such as the [Amazon API Gateway<sup>137</sup>](#) and the [Microsoft Azure API Gateway<sup>138</sup>](#) are examples of API gateways.

The examples in this book do not offer public REST interfaces, but only REST interfaces for internal use. The public interfaces are just web pages. Thus, the examples do not use API gateways.

## 13.4 Variations

Frontend integration ([chapter 7](#)) can be a good addition to synchronous communication. Asynchronous communication (see [chapter 10](#)) is rather an alternative. Synchronous and asynchronous communication are both possibilities with which microservices can communicate with each other on the logic level.

One of these options should be enough to build a microservices system. Of course, a combination is also possible. The asynchronous communication with Atom and the synchronous communication with REST use the same infrastructure so that these two communication mechanisms can be used together very easily.

The following chapters show concrete implementations for synchronous communication. The examples all use REST for communication. In today's technology landscape, REST is the preferred architecture for synchronous communication. In principle, other feasible approaches include [SOAP<sup>139</sup>](#) or [Thrift<sup>140</sup>](#).

<sup>134</sup><https://apigee.com/api-management/>

<sup>135</sup><https://www.redhat.com/de/technologies/jboss-middleware/3scale>

<sup>136</sup><http://www.apiman.io/latest/>

<sup>137</sup>[http://docs.aws.amazon.com/de\\_de/apigateway/latest/developerguide/welcome.html](http://docs.aws.amazon.com/de_de/apigateway/latest/developerguide/welcome.html)

<sup>138</sup><https://azure.microsoft.com/de-de/services/api-management/>

<sup>139</sup><https://www.w3.org/TR/soap/>

<sup>140</sup><https://thrift.apache.org/>

## 13.5 Conclusion

At first glance, synchronous microservices are very simple because they correspond to the classic programming model. But synchronous microservices lead to a high degree of technical complexity because microservices have to deal with the failure of other microservices and accumulating latencies. The resulting systems are distributed systems and thus technically complex; this has few advantages. It is advisable to choose this approach only if there are really good reasons for doing so. In general, asynchronous communication makes it easier to deal with the challenges of distributed systems than synchronous communication.

# 14 Recipe: REST with the Netflix Stack

This chapter provides the following content:

- Overview of the Netflix microservices stack
- Details about service discovery with Eureka, routing with Zuul, load balancing with Ribbon, and resilience with Hystrix
- The advantages and disadvantages of the Netflix stack

With this information, the reader can assess the suitability of these technologies for a concrete project and for a microservices using these technologies.

## Where Does the Netflix Stack Come From?

Netflix<sup>141</sup> has developed a new platform for online video streaming to meet the high performance and scalability requirements in this area. The result was one of the first microservices architectures. Later, Netflix released its technologies as open source projects. Therefore, the Netflix stack is one of the first stacks to implement microservices.

## License and Technology

The components of the Netflix stack are open source and are under the very liberal Apache license. The Netflix projects are practically all based on Java. They are integrated into Spring Cloud, making it much easier to use them together with Spring Boot (see section 5.3).

## 14.1 Example

The example for this chapter can be found at <https://github.com/ewolff/microservice>. It consists of three microservices.

- The *catalog* microservice manages the information about the items.
- The *customer* microservice stores the customer data.
- The *order* microservice can accept new orders. It uses the catalog and the customer microservice.

---

<sup>141</sup><https://www.netflix.com/>

## Architecture of the Example

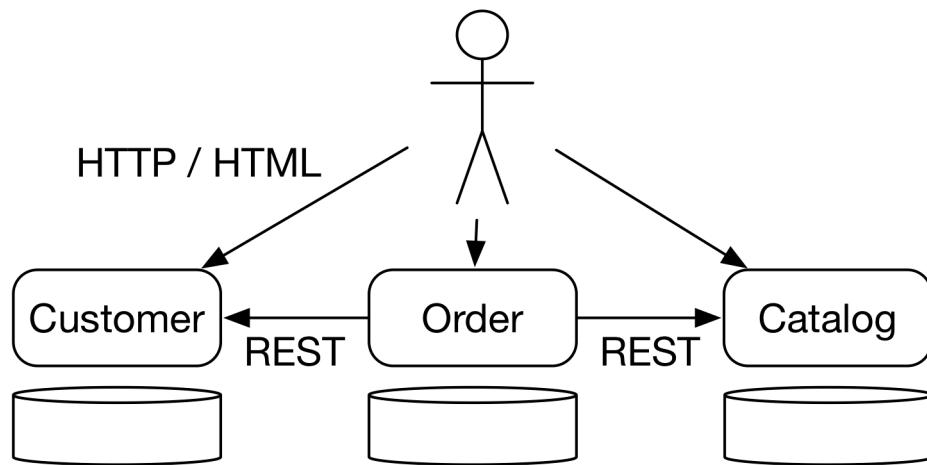


Fig. 14-1: Architecture of the Netflix Example

Each of the microservices has its own web interface with which users can interact. Among each other, the microservices communicate via REST. The order microservice requires information about customers and items from the other two microservices (see [figure 14-1](#)).

In addition to the microservices, a Java application displays the Hystrix dashboard for monitoring the Hystrix circuit breakers. [Figure 14.2](#) shows the entire example at the level of the Docker containers.

## Building the Example

[Section 0.4](#) explains what software has to be installed for starting the example.

First the code has to be downloaded with `git clone https://github.com/ewolff/microservice.git`. Then the code has to be compiled with `./mvnw clean package` (macOS, Linux) or `mvnw.cmd clean package` (Windows) in the directory `microservice-demo`. See [appendix B](#) for more details on Maven and how to troubleshoot the build. Afterwards, the Docker containers can be built with `docker-compose build` in the directory `docker` and started with `docker-compose up -d`. See [appendix C](#) for more details on Docker, Docker Compose and how to troubleshoot them. Subsequently, the Docker containers are available on the Docker host.

<https://github.com/ewolff/microservice/blob/master/HOW-TO-RUN.md> explains in detail the steps necessary to build and run the example.

## Docker Containers and Ports

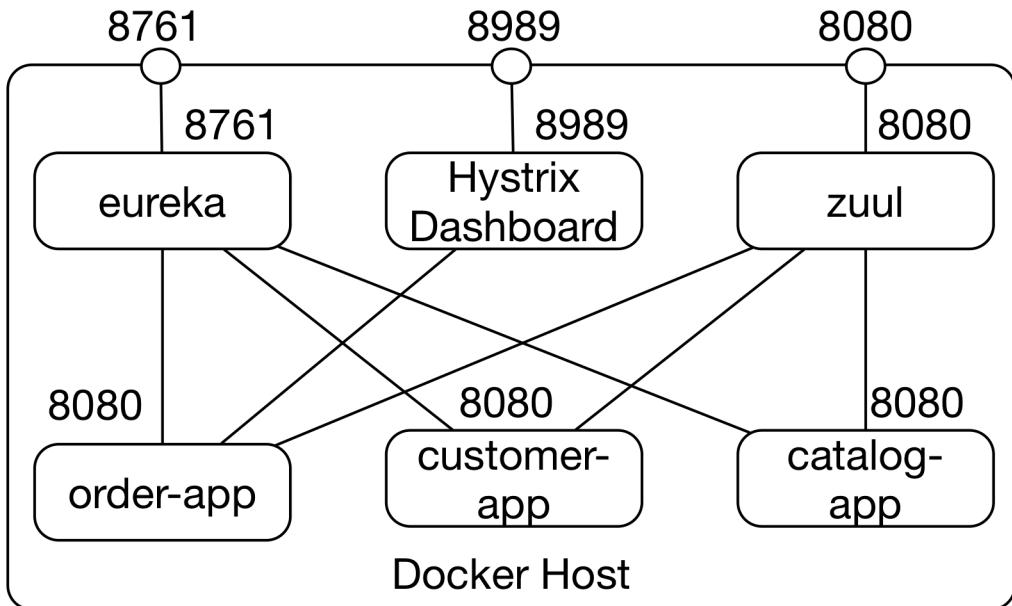


Fig. 14-2: Docker Containers in the Netflix Example

The Docker containers communicate via an internal network. Some Docker containers can also be used via a port on the Docker host. The Docker host is the computer on which the Docker containers run.

- The three microservices *order*, *customer*, and *catalog* each run in their own Docker containers. Access to the Docker containers is only possible within the Docker network.
- To use the services from the outside, *Zuul* provides routing. The *Zuul* container can be accessed from outside under port 8080 and forwards requests to the microservices. If the Docker containers are running locally, the URL is <http://localhost:8080>. At this URL, a web page is available that includes links to all microservices, Eureka, and the Hystrix dashboard.
- *Eureka* serves as service discovery solution. The dashboard is available at port 8761. This port is also accessible at the Docker host. For a local Docker installation, the URL is <http://localhost:8761>.
- Finally, the *Hystrix dashboard* runs in its own Docker container that can also be accessed under port 8989 on the Docker host – for example, at <http://localhost:8989>.

## 14.2 Eureka: Service Discovery

Eureka implements service discovery. As already discussed in [section 13.3](#), for synchronous communication microservices have to find out at which port and IP address other microservices can be accessed.

Essential characteristics of Eureka are:

- Eureka has a *REST interface*. Microservices can use this interface to register or request information about other microservices.
- Eureka supports *replication*. The information from the Eureka servers is distributed to other servers. This enables the system to compensate for the failure of Eureka servers. In a distributed system, service discovery is essential for communication between microservices. Therefore, service discovery must be implemented in such a way that a failure of one server does not cause the entire service discovery to fail.
- Due to *caches on the client*, the performance of Eureka is very good. In addition, availability is improved because the information is stored on the client, thus compensating for server failure. The server sends information only about new or deleted microservices to the client and not information about all registered services, making communication very efficient.
- Netflix supports *AWS* (Amazon Web Services) – that is, the Amazon Cloud. In AWS, servers run in availability zones. These are basically separate data centers. The failure of an availability zone does not affect other availability zones. Several availability zones form a region. A region is located in a geographical zone. For example, the data centers for the region called EU-West-1 is located in Ireland. Eureka can take regions and availability zones into account and, for example, offer a microservice instance from the same availability zone to a client as a result of the service discovery in order to increase speed.
- Eureka expects the microservices to regularly send *heartbeats*. In this way, Eureka detects crashed instances and excludes them from the system. This increases the probability that Eureka will return available service instances. However, a microservice instance that is no longer running could still be returned. But whether a microservice instance is crashed is obvious after it is used so that, if necessary, a different instance can be substituted.

## Servers

The Netflix Eureka project is available for download at [GitHub<sup>142</sup>](#), so you can build the project and get both the server and the client.

The Spring Cloud project also supports Eureka. The Eureka server can even be started as a Spring Boot application. For this, the main class that also has the annotation `@SpringBootApplication` has to be annotated as well with `@EnableEurekaServer`. In `pom.xml` in the `dependencyManagement` section, the Spring Cloud dependencies have to be imported, and a dependency on the library `spring-cloud-starter-eureka-server` has to be inserted. In addition, a configuration in the `application.properties` file is necessary. The project `microservice-demo-eureka-server` provides all of that and implements an Eureka server.

At first glance, it does not seem to make much sense to build the Eureka server by yourself in this way, especially because the implementation essentially consists of an annotation. But the Eureka server can be treated like all the other microservices. The Spring Cloud Eureka server is a JAR file

---

<sup>142</sup><https://github.com/Netflix/eureka/>

and, like all other microservices, can be stored and started in a Docker image. It is also possible to secure it like a Java web application with Spring security, for example, and configure logging and monitoring as with all other microservices.

The screenshot shows a browser window titled "Eureka" with the URL "localhost:8761". The page has a dark header with the "spring Eureka" logo. Below the header, there's a "System Status" section containing various configuration parameters:

Environment	test
Data center	default
Current time	2017-06-22T19:51:30 +0000
Uptime	01:14
Lease expiration enabled	true
Renews threshold	10
Renews (last min)	108

Below the status section is a "DS Replicas" section with a search bar containing "localhost". Underneath is a heading "Instances currently registered with Eureka" followed by a table:

Application	AMIs	Availability Zones	Status
CATALOG	n/a (1)	(1)	UP (1) - 5d50e86620ae:catalog:8080
CUSTOMER	n/a (1)	(1)	UP (1) - e34c51981195:customer:8080
ORDER	n/a (1)	(1)	UP (1) - f513a9945277:order:8080
TURBINE	n/a (1)	(1)	UP (1) - e741763d63e1:turbine:8989
ZUUL	n/a (1)	(1)	UP (1) - 20dec567ae39:zuul:8080

Fig. 14-3: Eureka Dashboard

Eureka provides a dashboard (see [figure 14-3](#)). It displays an overview of the microservices registered with Eureka. This includes the names of the microservices and the URLs at which you can access them. However, the URLs only work in the Docker internal network, so that the links in the

dashboard do not work. The dashboard is accessible on the Docker host at port 8761 – that is, <http://localhost:8761/> if the example runs locally.

## Client

Each microservice is a Eureka client and must register with the Eureka server in order to inform the Eureka server about the name of the microservice, as well as the IP address and port at which it can be reached. Spring Cloud simplifies the configuration for clients.

## Registration

The client has to have a dependency on `spring-cloud-starter-eureka` to add the necessary libraries. The main class which has the annotation `@SpringBootApplication` additionally has to be annotated with `@EnableEurekaClient`. An alternative is `@EnableDiscoveryClient`. In contrast to `@EnableEurekaClient`, the annotation `@EnableDiscoveryClient` is generic. Thus, it also works with Consul (see [chapter 15](#)).

```
spring.application.name=catalog
eureka.client.serviceUrl.defaultZone=http://eureka:8761/eureka/
eureka.instance.leaseRenewalIntervalInSeconds=5
eureka.instance.metadataMap.instanceId=${spring.application.name}:${random.value}
eureka.instance.preferIpAddress=true
```

In the file `application.properties` just shown, appropriate settings must be entered for the application to register.

- `spring.application.name` contains the name under which the application registers at the Eureka server.
- `eureka.client.serviceUrl.defaultZone` defines which Eureka server is to be used.
- The setting `eureka.instance.leaseRenewalIntervalInSeconds` ensures that the registration information is replicated every five seconds and thus is replicated faster than the default setting. This makes new microservice instances usable more quickly. In production, this value should not be set so low so that there is not too much network traffic.
- `eureka.instance.metadataMap.instanceId` provides each microservice instance with a random ID – for example, to be able to discriminate between two instances for load balancing.
- Due to `eureka.instance.preferIpAddress`, the services register with their IP address and not with their host name. This avoids problems that arise because host names cannot be resolved in the Docker environment.

During registration, the name of the microservice is automatically converted to uppercase letters. Thus `order` is turned into `ORDER`.

## Other Programming Languages

For programming languages other than Java, you must use a library to access Eureka. There are some libraries that implement Eureka clients for certain programming languages. Of course, Eureka provides a REST interface that can also be used.

### Sidecars

To use the Netflix infrastructure with other programming languages, it is also possible to use a *sidecar*. A sidecar is an application written in Java that uses the Java libraries to talk to the Netflix infrastructure. The application uses the sidecar to communicate with the Netflix infrastructure, making the sidecar just a helper for the real application. That way, the application can be written in any programming language. Essentially, the sidecar is the interface to the Netflix infrastructure.

In this way, for example, Eureka can support other programming languages, but this requires an additional process that consumes additional resources.

Netflix itself offers [Prana<sup>143</sup>](#) as sidecar. Spring Cloud also provides an implementation of such a sidecar<sup>144</sup>.

### Access to Other Services

In the example, Ribbon (see [section 14.4](#)) implements the access to other services in order to implement load balancing across multiple instances. Thus, the Eureka API is used via Ribbon only to find the information about the other microservices.

## 14.3 Router: Zuul

Zuul is the routing solution that is part of the Netflix stack. Zuul is responsible for forwarding external calls to the correct microservice.

### Zuul versus Reverse Proxy

The routing could also be provided by a Reverse Proxy. This is a web server configured to forward incoming calls to other servers.

Zuul has a feature that a reverse proxy is lacking: dynamic filters. Depending on the properties of the HTTP request or external configurations, Zuul can forward certain calls to specific servers or execute logic for logging, for example. A developer can write custom code for the routing decision. The code can even be dynamically loaded as Groovy code at runtime. In this way, Zuul ensures maximum flexibility.

---

<sup>143</sup><https://github.com/Netflix/Prana/wiki>

<sup>144</sup>[http://projects.spring.io/spring-cloud/spring-cloud.html#\\_polyglot\\_support\\_with\\_sidecar](http://projects.spring.io/spring-cloud/spring-cloud.html#_polyglot_support_with_sidecar)

Zuul filters can also be used to implement central functionalities such as logging of all requests. A Zuul filter can implement the login and send information about the current user with the HTTP requests, thereby implementing authentication. Zuul can thus take over typical functionalities of an API gateway (see [section 13.3](#)).

## Zuul in the Example

In the example, Zuul is configured as a proxy and does not contain any special code. Zuul forwards access to a URL such as <http://localhost:8080/order> to the order microservice under the name “ORDER” in Eureka (see [figure 14-4](#)). All names in Eureka are in uppercase. Such a forwarding works for all microservices registered in Eureka. Zuul reads the names of all microservices from Eureka and forwards the requests.

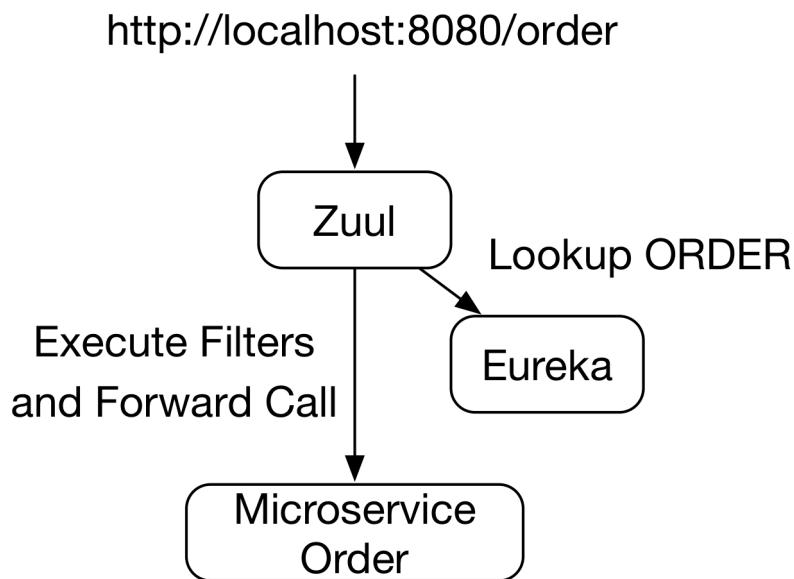


Fig. 14-4: Routing with the Zuul Proxy

Of course, Zuul “reveals” which microservices comprise the microservices system. However, Zuul can be reconfigured by routes and filters in such a way that completely different microservices can be accessed under the same URL.

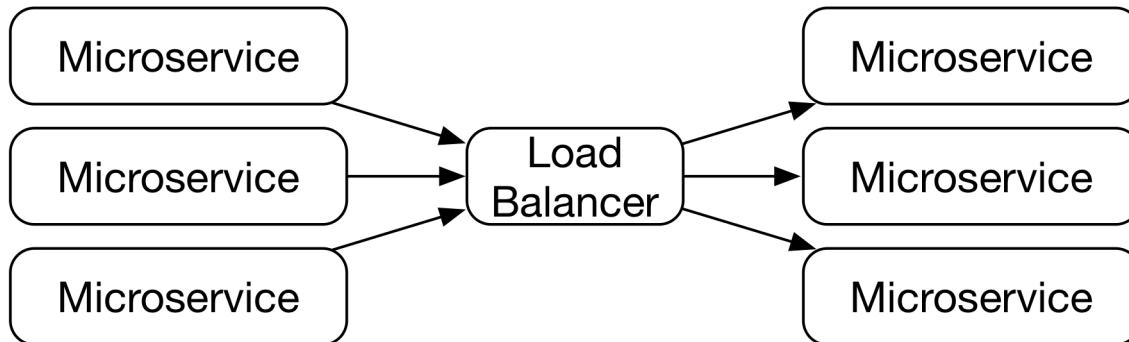
Zuul can also deliver static content. In the example, Zuul provides the web page from which you can access the various microservices.

## 14.4 Load Balancing: Ribbon

Microservices have the advantage that each microservice can be scaled independently of the other microservices. To do this, the call to a microservice must be distributed to several instances by a load balancer.

## Central Load Balancer

Typically, a single load balancer is used for all calls. Therefore, a single load balancer, which processes all requests from all microservices, can also be used for an entire microservices system (see [figure 14-5](#)). However, such an approach leads to a bottleneck because all network traffic must be routed through this single load balancer. The load balancer is also a single point of failure. If the load balancer fails, all network traffic stops functioning and the entire microservices system fails.

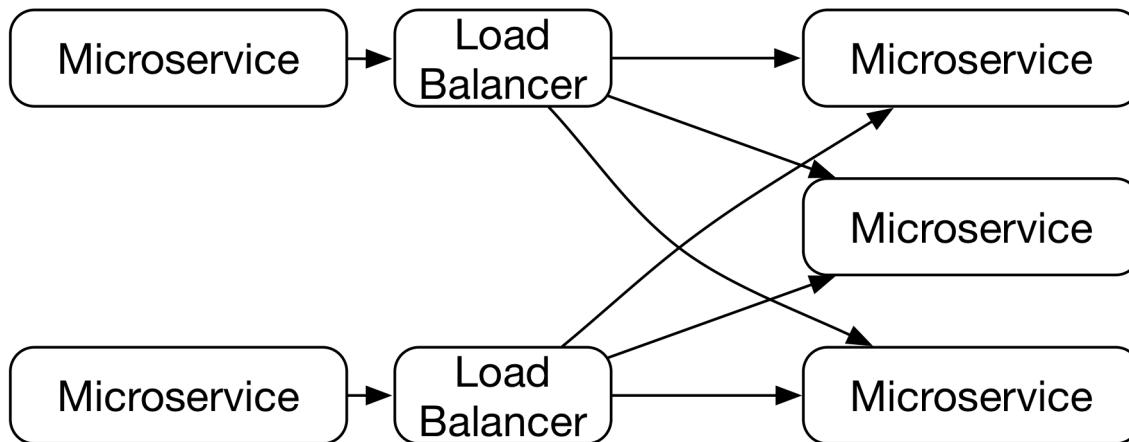


[Fig. 14-5: Central Load Balancer](#)

Decentralized load balancing would be better. For this, each microservice must have its own load balancer. If a load balancer fails, only one microservice will fail.

## Client-side Load Balancing

The idea of client-side load balancing (see [figure 14-6](#)) can be implemented by a “normal” load balancer such as Apache httpd or nginx. A load balancer is deployed for each microservice. The load balancer must obtain the information about the currently available microservices from the service discovery.



[Fig. 14-6: Client-side Load Balancer](#)

It is also possible to write a library that distributes requests to other microservices to different instances. This library must read the currently available microservice instances from the service

discovery and then, for each request, select one of the instances. This is not particularly hard to implement and is the way [Ribbon<sup>145</sup>](#) works.

## Ribbon API

Ribbon offers a relatively simple API for load balancing.

```
private LoadBalancerClient loadBalancer;
// Spring injects a LoadBalancerClient
ServiceInstance instance = loadBalancer.choose("CUSTOMER");
url = String.format("http://%s:%s/customer/",
instance.getHost(), instance.getPort());
```

Spring Cloud injects an implementation of the interface `LoadBalancerClient`. First, a call to the `LoadBalancerClient` selects an instance of a microservice. This information is then used to fill a URL to which the request can be sent.

Ribbon supports various strategies for selecting an instance. Thus, approaches other than a simple round robin are feasible.

## Ribbon with Consul

As part of the Netflix stack, Ribbon supports Eureka as a service discovery tool, but it also supports Consul. In the Consul example (see [chapter 15](#)), access to the microservices is implemented identically to the Netflix example.

## RestTemplate

Spring contains the `RestTemplate` to easily implement REST calls. If a `RestTemplate` is created by Spring and annotated with `@LoadBalanced`, Spring makes sure that a URL like `http://order/` is forwarded to the order microservice. Internally, Ribbon is used for this. <https://spring.io/guides/gs/client-side-load-balancing/> shows how this approach can be implemented with a `RestTemplate`.

## 14.5 Resilience: Hystrix

With synchronous communication between microservices, it is important that the failure of one microservice does not cause other microservices to fail as well. Otherwise, the unavailability of a single microservice can cause further microservices to gradually fail until the entire system is no longer available.

The microservices may, of course, return errors because they cannot deliver reasonable results due to a failed microservice. However, a microservice should never wait for the result of another microservice for an infinite period of time, thereby becoming unavailable itself.

---

<sup>145</sup><https://github.com/Netflix/ribbon/wiki>

## Resilience Patterns

The book “Release It!”<sup>146</sup> describes different patterns with which the resilience of a system can be increased. Hystrix implements some of these patterns.

- A *timeout* prevents a microservice from waiting too long for another microservice. Without a timeout, a thread can block for a very long time because, for example, the thread does not get a response from another microservice. If all threads are blocked, the microservice will fail because no more threads available for new tasks exist. Hystrix executes a request in a separate thread pool. Hystrix controls these threads and can terminate the request to implement the timeout.
- *Fail Fast* describes a similar pattern. It is better to generate an error as quickly as possible. The code can check at the beginning of an operation whether all necessary resources are available. This may include enough disk space, for example. If this is not the case, the request can be terminated immediately with an error. This reduces the time that the caller has to block a thread or other resources.
- Hystrix can use its own thread pool for each type of request. For example, a separate thread pool can be set up for each called microservice. If the call of a particular microservice takes too long, only the thread pool for that particular microservice is emptied, whereas the others still contain threads. This limits the impact of the problem and is called a *bulkhead*. This term was coined in analogy to a watertight bulkhead in a ship, which divides the ship into different segments. If a leak occurs, only part of the ship is flooded with water so that the ship does not sink.
- Finally, Hystrix implements a *circuit breaker*. This is a fuse analogous to the ones used in the electrical system of a house. There, a circuit breaker is used to cut off the current flow in case of a short circuit. This prevents, for example, a fire from breaking out. The Hystrix circuit breaker has a different approach: If a system call results in an error, the circuit breaker is opened and does not allow any further calls to pass through. After some time, a call is allowed to pass through again. Only when this call is successful, does the circuit breaker close again. This prevents a faulty microservice from being called. This saves resources and avoids blocked threads. In addition, the circuit breakers of the different clients are closing one by one so that a failed and recovered microservice only gradually has to handle the full load. This reduces the probability that it will fail again immediately after starting up.

## Implementation

Hystrix<sup>147</sup> offers an implementation of most resilience patterns as a Java library.

The Hystrix API requires command objects rather than simple method calls. These classes supplement the method call with the necessary Hystrix functionalities. When using Hystrix with Spring Cloud, it is not necessary to implement commands. Instead, the methods are annotated

---

<sup>146</sup>Michael T. Nygard: Release It!: Design and Deploy Production-Ready Software, Pragmatic Bookshelf, 2nd Edition, 2017, ISBN 978 1 68050 239 8

<sup>147</sup><https://github.com/Netflix/Hystrix/>

with `@HystrixCommand`, which activates Hystrix for this method. The attributes of the annotation configure Hystrix.

```
@HystrixCommand(
    fallbackMethod = "getItemsCache",
    commandProperties = {
        @HystrixProperty(
            name = "circuitBreaker.requestVolumeThreshold",
            value = "2"))
public Collection<Item> findAll() {
    ...
    this.itemsCache = pagedResources.getContent();
    ...
    return itemsCache;
}
```

The listing shows the access from the order microservice to the catalog microservice. `circuitBreaker.requestVolumeThreshold` specifies how many calls in a time window must cause errors for the circuit breaker to open. In addition, the `fallbackMethod` attribute of the annotation configures the method `getItemsCache()` as a fallback method. `findAll()` stores the data returned by the catalog microservice in the instance variable `itemsCache`. The `getItemsCache()` method serves as a fallback and reads the result of the last call from the instance variable `itemsCache` and returns it.

```
private Collection<Item> getItemsCache() {
    return itemsCache;
}
```

The reasoning behind this is that it is better that the service continues to work with outdated data than that the service does not work at all. This can lead to orders being charged at an outdated price. However, this is probably the better option compared to accepting no orders at all.

In general, if a service fails, a default value can also be used, or an error can be reported. Reporting an error is the right solution if incorrect data cannot be accepted under any circumstances. Which approach is correct inevitably depends on the domain logic. It should only be avoided that, in case of an error, the REST call burdens the server or blocks the client for too long.

## Monitoring

The state of the circuit breakers provides a good overview of the state of the system. An open circuit breaker is an indication of a problem. So Hystrix is a good source of metrics because it provides information about the circuit breaker state via HTTP as a stream of JSON data.

## Hystrix Dashboard

The Hystrix dashboard can display this data on a web page and thereby show what is happening in the system at the moment (see [figure 14-7](#)).

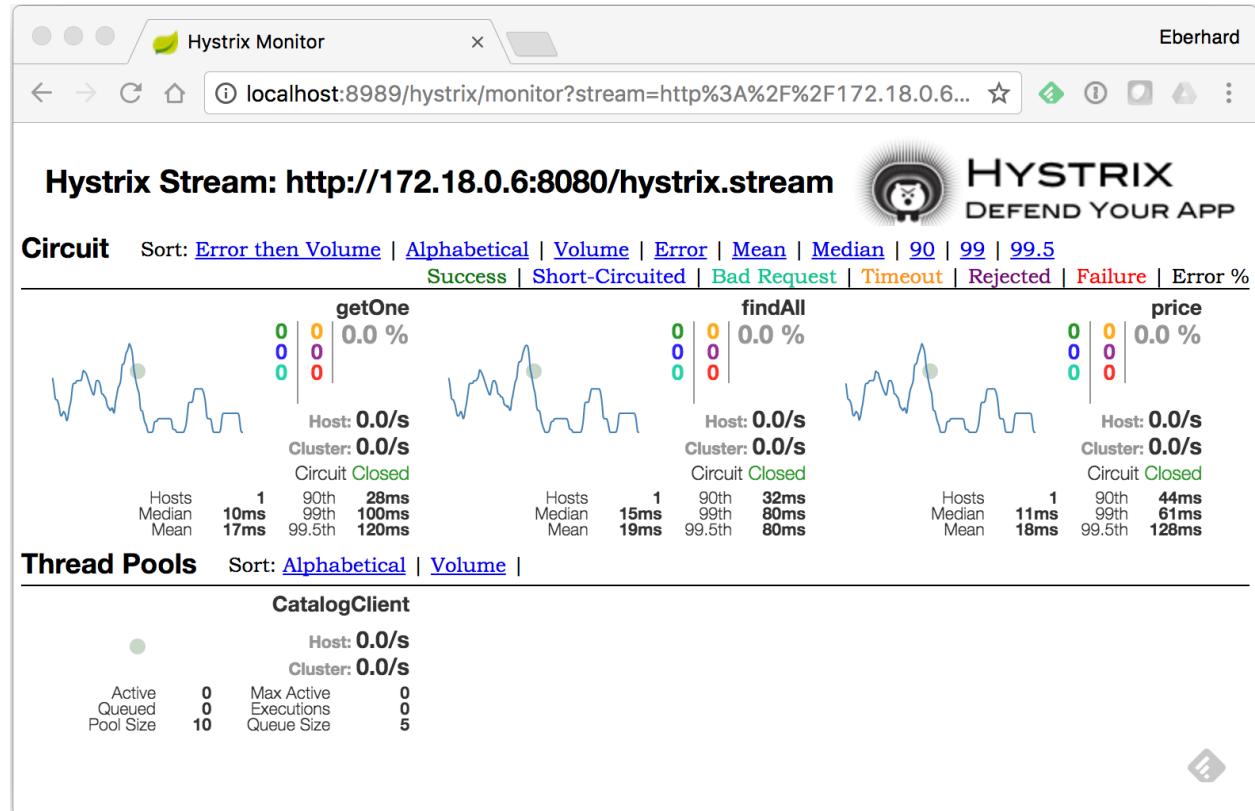


Fig. 14-7: Hystrix Dashboard

The upper area shows the state of the circuit breaker for the functions `getOne()`, `findAll()`, and `price()`. The circuit breakers of all three functions are closed, so no errors occurred at the moment. The dashboard also shows information about the average latency of the requests and current throughput.

Hystrix executes the calls in a separate thread pool. The state of this thread pool is also shown on the dashboard. It contains ten threads and is currently processing no requests.

## Other Monitoring Options

The Hystrix metrics are also available via the Spring Boot mechanisms (see [section 5.3](#)) and can be exported to other monitoring systems. Therefore, Hystrix metrics can be seamlessly integrated into an existing monitoring infrastructure.

## Turbine

The metrics of a single microservice instance are not particularly meaningful. Microservices can be scaled independently. This means that many instances can exist for each microservice and that the Hystrix metrics of all instances must be displayed together.

You can do this with [Turbine<sup>148</sup>](#), which queries the HTTP data streams of the Hystrix servers and consolidates them into a single stream of data displayed by the dashboard. Spring Cloud offers a simple way to implement a Turbine server – see for example, <https://github.com/ewolff/microservice/tree/master/microservice-demo/microservice-demo-turbine-server>.

## 14.6 Recipe Variations

Netflix is only one technological option for implementing synchronous microservices. Various alternatives to the technologies of the Netflix stack are available.

- The Zuul project is not maintained that much any more. An alternative might be [Zuul2<sup>149</sup>](#). It is based on [asynchronous I/O<sup>150</sup>](#), so it consumes fewer resources and is more stable. However, Spring Cloud won't support Zuul2<sup>151</sup>, and so another alternative might be [Spring Cloud Gateway<sup>152</sup>](#). But the approach with Apache for routing shown in [chapter 15](#) and Kubernetes in [chapter 17](#) is probably even better.
- Netflix lately does not invest in Hystrix a great deal. They suggest using [resilience4j<sup>153</sup>](#) instead, which is a very similar Java library that also supports typical resilience patterns.
- The *Consul example* ([chapter 15](#)) uses Consul rather than Eureka for service discovery and Apache httpd rather than Zuul for routing. However, this project also uses Hystrix for resilience and Ribbon for load balancing. Consul also supports DNS and can thus handle any programming language as implemented in the Consul DNS example. Consul Template offers the possibility to configure services with Consul by filling a configuration file template with the data from Consul. In the example, Apache httpd is configured this way. Eureka has quite a few advantages over Consul. Apache httpd as a web server is familiar to many developers and might therefore be the less risky compared to Zuul. On the other hand, Zuul provides dynamic filters that Apache httpd does not support.
- *Kubernetes* (see [chapter 17](#)) and a *PaaS* such as *Cloud Foundry* (see [chapter 18](#)) offer service discovery, routing, and load balancing. At the same time, the code remains independent of the infrastructure. Nevertheless, the examples in those chapters uses Hystrix for resilience, too. These solutions require the use of a Kubernetes or PaaS environment. Thus, just deploying some Docker containers on a Linux server is no longer possible.

<sup>148</sup><https://github.com/Netflix/Turbine/wiki>

<sup>149</sup><https://github.com/Netflix/zuul>

<sup>150</sup><https://medium.com/netflix-techblog/zuul-2-the-netflix-journey-to-asynchronous-non-blocking-systems-45947377fb5c>

<sup>151</sup><https://github.com/spring-cloud/spring-cloud-netflix/issues/1498>

<sup>152</sup><https://spring.io/projects/spring-cloud-gateway>

<sup>153</sup><https://github.com/resilience4j/resilience4j>

- Functionalities such as load balancing and resilience can be implemented with an HTTP proxy rather than Ribbon. This is a further development of the sidecar concept. An example is [Envoy<sup>154</sup>](#). This proxy implements some resilience patterns. Envoy is also part of Istio (see [section 23.3](#)) and is used as a sidecar in Istio application. Istio is a service mesh that supports many technologies for the operation of microservices systems. With a proxy, the application itself can be kept free of these aspects. Apache httpd or nginx also can at least implement load balancing, so they could provide basic features of a sidecar.
- Asynchronous communication (see [chapter 10](#)) seems at first sight to be a contradiction to communication via a synchronous protocol like REST. But *Atom* (see [chapter 12](#)) can be combined with concepts from this chapter. Atom uses REST, so the microservices only need to implement other types of REST resources. A combination with messaging systems like *Kafka* (see [chapter 11](#)) is also feasible. However, in this case, the system not only has the complexity of the messaging system, but must also offer a REST environment.
- *Frontend integration* ([chapter 7](#)) works on a different level than REST and can be combined with the Netflix stack. In particular, integration with *links and JavaScript* ([Chapter 8](#)) is possible without any problems. With *ESI* (see [chapter 9](#)), Varnish rather than Zuul implements routing. So Varnish would have to extract the IP addresses of the microservices from Eureka. However, this is not possible without further effort.

## 14.7 Experiments

- Supplement the system with an additional microservice.
  - A microservice that a call center agent uses to create notes for a call can be used as an example. The call center agent should be able to select the customer.
  - Of course, you can copy and modify one of the existing microservices.
  - Register the microservice in Eureka.
  - The customer microservice must be called via Ribbon which finds the microservice automatically via Eureka. Otherwise, the microservice must be looked up explicitly in Eureka.
  - Package the microservice in a Docker image and add the image to `docker-compose.yml`. There you can also determine the name of the Docker container.
  - Create a link in `docker-compose.yml` from the container with the new service to the container `eureka`. That way the microservice can register at the Eureka server.
  - The microservice must be accessible from the home page. To do this, you have to create a link similar to the other links in the file `index.html` in the Zuul project. Zuul automatically sets up the routing for the microservice as soon as the microservice is registered in Eureka.
- Try scaling and load balancing.
  - Increase the number of instances of a service – for example, with `docker-compose up --scale customer=2`.

---

<sup>154</sup><https://github.com/lyft/envoy>

- Use the Eureka dashboard to determine whether two customer microservices are running. It is available at port 8761 – for example, at <http://localhost:8761/> when Docker is running on the local computer.
- Observe the logs of the order microservice with `docker logs -f ms_order_1` and have a look whether different instances of the customer microservice are called. This should be the case because Ribbon is used for load balancing. For this, you have to trigger requests to the order application – for example, a simple reload of the starting page.
- Simulate the failure of a microservice.
  - Watch the logs of the order microservice with `docker logs -f ms_order_1` and have a look at how the catalog microservice is called. For this, you have to trigger requests to the order application. For example you can reload the starting page.
  - Find the IP address of the order microservice with the help of the Eureka dashboard at port 8761 – for example, <http://localhost:8761/> when Docker is running locally.
  - Open the Hystrix dashboard at port 8989 on the Docker host – for example, at <http://localhost:8989/> when Docker is running on the local computer.
  - Using this IP address, enter the URL of the Hystrix JSON data stream in the Hystrix dashboard. This can for instance be <http://172.18.0.6:8080/actuator/hystrix.stream>. The Hystrix dashboard should show closed circuit breakers like in [figure 14-7](#).
  - Shut down all catalog instances with `docker-compose up --scale catalog=0`.
  - Watch the log of the order microservices during the next calls.
  - Also observe the Hystrix dashboard. The circuit breaker will open only when multiple calls have failed.
  - When the circuit breaker is open, the order microservice should work again because now the fallback is activated. Then a cached value is used.
- Only the access to the catalog microservice is safeguarded with Hystrix. In the order microservice the class `CustomerClient` in package `com.ewolff.microservice.order.clients` implements the access to the customer microservice. Extend the access to the customer microservice using Hystrix. For this, use the class `CatalogClient` from the same package as an example.
- Extend the Zuul setup by a fixed route. In `application.yml` in directory `src/main/resource` in project `microservice-demo-zuul-server` add for example the following to make the INNOQ home page appear at <http://localhost:8080/innoq>:

```

zuul:
  routes:
    innoq:
      path: /innoq/**
      url: http://innoq.com/

```

- Add a filter to the Zuul configuration. You can find a tutorial dealing with Zuul filters at <https://spring.io/guides/gs/routing-and-filtering/>.

- Create your own microservice that, for example, returns only simple HTML. Integrate it into Eureka and deploy it as part of the Docker compose environment. If it is registered in Eureka, it can be addressed immediately from the Zuul proxy at a URL like <http://localhost:8080/mymicroservice>.

## 14.8 Conclusion

The Netflix stack provides a variety of projects to build microservice architectures. The stack solves the typical challenges of synchronous microservices as follows:

- *Service discovery* is offered by Eureka. Eureka focuses on Java with the Java client, but also offers a REST API and libraries for other languages. Eureka can therefore also be used with other languages.
- For *resilience*, Hystrix is the de facto standard for Java and covers this area very well. Non-Java applications can use Hystrix via a sidecar at most. There are ports of Hystrix for other languages like such as, for example, Go (see [section 5.4](#)). Hystrix is independent from the other technologies and can therefore be used on its own.
- Ribbon implements client-side *load balancing*, which has many advantages. Because Ribbon is a Java library, other technologies are difficult to use with Ribbon. Especially in the area of load balancing, there are numerous classic load balancers that provide alternatives. Ribbon relies on Eureka for service discovery, but can also use Consul.
- *Routing* is solved by Zuul. Zuul's dynamic filters are very flexible, but many developers are rather familiar with reverse proxies based on web servers like Apache httpd or nginx. In this case, a reverse proxy might be the safer option. Additional features such as SSL termination, request throttling, or similar functionality may also be required, which Zuul does not offer. [Section 15.3](#) shows how Apache httpd can be used with Consul to provide routing. Zuul requires Eureka to find the microservices.

The servers in the Netflix stack are written in Java, enabling the servers from the Netflix stack and the microservices to be packed into JAR files. Thanks to Spring Cloud, they are also uniformly configurable. Also the handling of metrics and logs is identical. This uniformity can be an advantage for operation.

## Advantages

- Eureka together with client-side caching is very fast and resilient.
- Zuul is very flexible because of its filters.
- Client-side load balancing avoids single points of failure or bottlenecks.
- Hystrix is very mature and the de facto standard for Java.

## Challenges

- The Netflix stack implements many solved problems (for example, reverse proxy and service discovery) anew.
- The technologies focus on Java and therefore limits technology freedom.
- The code depends on the Netflix stack (Ribbon, Hystrix, but also Eureka due to `@EnableDiscoveryClient` and the client API).

# 15 Recipe: REST with Consul and Apache httpd

This chapter shows an implementation for a synchronous microservices system with Consul and the Apache httpd server.

Essential concepts of the chapter are:

- Consul is a very powerful service discovery technology.
- Apache httpd can be used as a load balancer and router for HTTP requests in a microservices system.
- Consul Template can create a configuration file for the Apache httpd server that includes information about all registered microservices. Consul Template configures and restarts Apache httpd when new microservice instances are started.

## Where Does Consul Come From?

Consul is a product of the company Hashicorp<sup>155</sup>, which offers various products in the field of microservices and infrastructure. Of course, Hashicorp also offers commercial support for Consul.

## License and Technology

Consul<sup>156</sup> is an open source product. It is written in Go and is licensed under the Mozilla Public License 2.0<sup>157</sup>. The code is available at GitHub<sup>158</sup>.

## 15.1 Example

The domain structure is identical to the example in the Netflix chapter (chapter 14) and consists of three microservices (see figure 15-1).

---

<sup>155</sup><https://www.hashicorp.com/>

<sup>156</sup><https://www.consul.io/>

<sup>157</sup><https://github.com/hashicorp/consul/blob/master/LICENSE>

<sup>158</sup><https://github.com/hashicorp/consul>

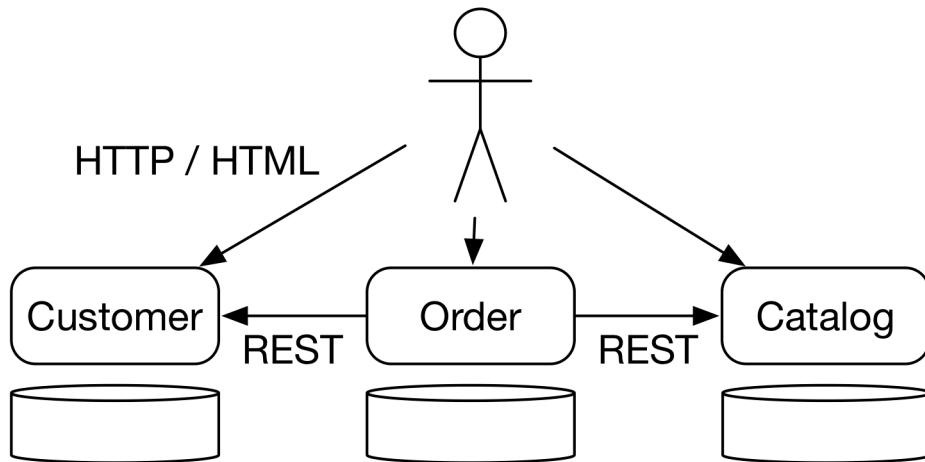


Fig. 15-1: Architecture of the Consul Example

- The *catalog* microservice manages information such as price or name for the items that customers can order.
- The *customer* microservice stores customer data.
- The *order* microservice can accept new orders. It uses the catalog and customer microservice.

## Architecture of the Example

The example in this chapter uses [Consul<sup>159</sup>](#) for service discovery and [Apache httpd server<sup>160</sup>](#) for routing the HTTP requests.

An overview of the Docker containers is shown in [figure 15-2](#). The three microservices provide their UI and REST interfaces at the port 8080. They are accessible only within the network between the Docker containers. Consul offers port 8500 for the REST interface and the HTML UI, as well as UDP port 8600 for DNS requests. These two ports are also bound to the Docker host, making these ports accessible at the Docker host and thus from other computers. The Docker host also provides the Apache httpd at port 8080. Apache httpd forwards calls to the microservices in the Docker network so that you can also access the microservices from outside.

---

<sup>159</sup><https://www.consul.io/>

<sup>160</sup><https://httpd.apache.org/>

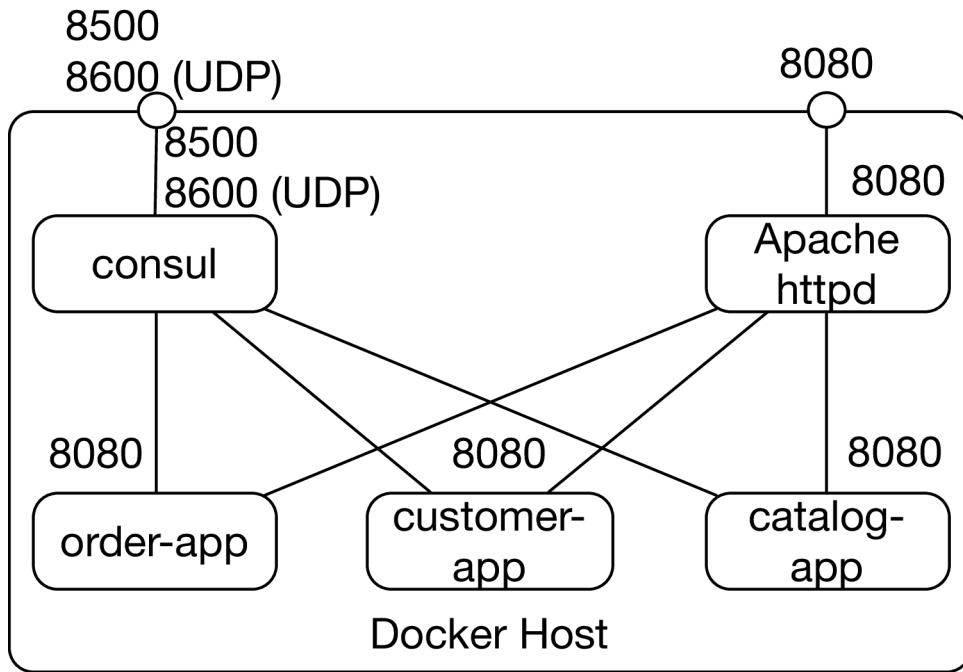


Fig. 15-2: Overview of the Consul Example

## Building the Example

Section 0.4 describes what software has to be installed for starting the example.

First download the code with `git clone https://github.com/ewolff/microservice-consul.git`. The code has then to be translated with `./mvnw clean package` (macOS, Linux) or `mvnw.cmd clean package` (Windows) in directory `microservice-consul-demo`. See [appendix B](#) for more details on Maven and how to troubleshoot the build. Afterwards the Docker containers can be built in the directory `docker` with `docker-compose build` and started with `docker-compose up -d`. See [appendix C](#) for more details on Docker, Docker Compose, and how to troubleshoot them. Subsequently, the Docker containers are available on the Docker host.

When the Docker containers are running on the local computer, the following URLs are available:

- <http://localhost:8500> is the link to the Consul dashboard.
- <http://localhost:8080> is the URL of the Apache httpd server. It can display the web UI of all microservices.

<https://github.com/ewolff/microservice-consul/blob/master/HOW-TO-RUN.md> describes the necessary steps for building and running the example in detail.

## 15.2 Service Discovery: Consul

Consul<sup>161</sup> is a service discovery technology. It ensures that microservices can communicate with each other. Consul has some features that set it apart from other service discovery solutions.

- Consul has an *HTTP REST API* and in addition supports *DNS*. DNS<sup>162</sup> (Domain Name System) is the system that maps host names such as www.innoq.com to IP addresses on the Internet. In addition to returning IP addresses, it can also return ports at which a service is available. This is a feature of the SRV DNS records.
- With *Consul Template*<sup>163</sup>, Consul can generate configuration files. The files may contain IP addresses and ports of services registered in Consul. *Consul Template* also provides Consul's service discovery feature to systems which cannot access Consul via the API. The systems only have to use some kind of configuration file, which they often already do anyway.
- Consul can perform *health checks* and exclude services from service discovery when the health check fails. For example, a health check can be a request to a specific HTTP resource to determine whether the service can still actually process requests. A service may still be able to accept HTTP requests, but it may not be able to process them properly due to a database failure. The service can signal this through the health check.
- Consul supports *replication* and can thus ensure high availability. If a Consul server fails, other servers with replicated data take over and compensate for the failed server.
- Consul also supports *multiple data centers*. Data can be replicated between data centers to further increase availability and protect Consul against the failure of a data center. The search for services can be limited to the same data center. Services in the same data center usually deliver higher performance.
- Finally, Consul can be used not only for service discovery, but also for the *configuration* of services. Configuration has different requirements. Availability is important for service discovery. Faulty information can be tolerated. If the service is accessed and is not available, it does not matter much; you can simply use another instance of the service. However, if services are configured incorrectly, this can lead to errors. So correct information is much more important for configuration than for service discovery.

### Consul Dashboard

Consul provides access to the information about registered microservices in its dashboard (see [figure 15-3](#)). It shows all services that have registered at the Consul server and also the result of their health checks. In the example all microservices are healthy and can accept requests. In addition to the registered services, it displays the servers on which Consul is running and the contents of the configuration database. You can access the dashboard at port 8500 on the Docker host.

<sup>161</sup><http://www.consul.io>

<sup>162</sup><http://www.zytrax.com/books/dns/>

<sup>163</sup><https://github.com/hashicorp/consul-template>

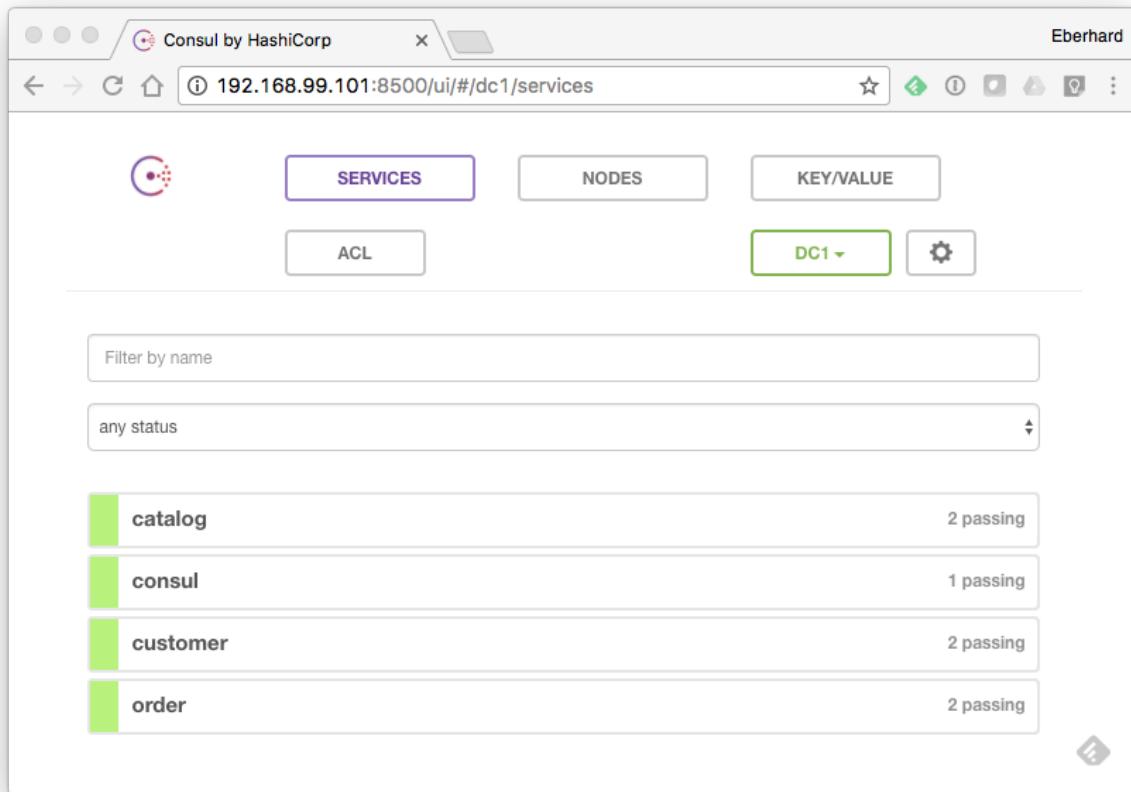


Fig. 15-3: Consul Dashboard

## Reading Data with DNS

You can also access the data from the Consul server via DNS. This can be done, for example, with the tool `dig`. `dig @localhost -p 8600 order.service.consul` returns the IP address of the order microservice if the Docker containers run on the local computer `localhost`. Communication to the Consul DNS server takes place via the UDP port 8600. `dig @localhost -p 8600 order.service.consul` SRV returns, in addition to the IP address, the port at which the service is available. DNS SRV records are used for this purpose. They are part of the DNS standard and allow you to specify a port for services in addition to the IP address.

## Consul Docker Image

The example uses a Consul Docker image directly from the manufacturer Hashicorp. It is configured so that Consul stores only the data in the main memory and runs on a single node. This simplifies the setup of the system and reduces the resource consumption. This configuration is of course unsuitable for a production environment because data can be lost, and a failure of the Consul Docker container would bring the entire system to a standstill. In production, a cluster of Consul servers should be used and the data should be stored persistently.

## 15.3 Routing: Apache httpd

The [Apache httpd server<sup>164</sup>](#) is one of the most widely used web servers. There are modules that adapt the server to different usage scenarios. In the example, modules are configured that turn Apache httpd into a reverse proxy.

### Reverse Proxy

Although a conventional proxy can be used to process traffic from a network to the outside, a reverse proxy is a solution for inbound network connections. It can forward external requests to specific services. This means that the entire microservices system can be accessible under one URL, but can use different microservices internally.

The concept of a reverse proxy has already been explained in [section 14.3](#).

### Load Balancer

In addition, Apache httpd serves as a load balancer. httpd distributes network traffic to multiple instances to make the application scalable. In the example, there is only one Apache httpd, which functions simultaneously as reverse proxy and load balancer for requests from the outside. The requests that the microservices send to each other are not handled by this load balancer. For the communication between the microservices, the library Ribbon is used, as already in the Netflix example (see [section 14.4](#)).

One of the strengths of this solution is that it uses a well-proven software, which teams often have existing experience with. Because microservices place high demands on the operation and infrastructure, such a conservative choice is advantageous to avoid the learning curve and effort of using another technology. Rather than Apache httpd, you can, of course, also use [nginx<sup>165</sup>](#), for example.

There are also approaches like [Fabio<sup>166</sup>](#) which are written specifically for the load balancing of microservices and are easier to use and configure.

## 15.4 Consul Template

For each microservice, Apache httpd must have an entry in its configuration file. [Consul Template<sup>167</sup>](#) can be used for this. In the example, the `00-default.ctmpl` file is used as a template to create the Apache httpd configuration. It is written in the [Consul Templating Language<sup>168</sup>](#). For each microservice, it creates an entry that distributes the load between the instances and redirects external requests to these instances.

<sup>164</sup><https://httpd.apache.org/>

<sup>165</sup><https://nginx.org/>

<sup>166</sup><https://github.com/eBay/fabio>

<sup>167</sup><https://github.com/hashicorp/consul-template>

<sup>168</sup><https://github.com/hashicorp/consul-template#templating-language>

## The Template

The essential part is:

```

{{range services}}
<Proxy balancer://{{.Name}}>
{{range service .Name}}  BalancerMember http://{{.Address}}:{{.Port}}
{{end}}
</Proxy>
ProxyPass      /{{.Name}} balancer://{{.Name}}
ProxyPassReverse /{{.Name}} balancer://{{.Name}}

{{end}}

```

The Consul Template API functions are enclosed in {{ and }}. For each service, a configuration is generated with {{range service}}. It contains a reverse proxy configured with the <Proxy> element and the Name of the microservice. This element contains the microservice instances as BalancerMember with the Address and Port of each instance for distributing the load between the microservice instances. The end is formed by ProxyPass and ProxyPassReverse, which likewise belong to the reverse proxy.

## Starting the Consul Template

The Consul Template is started with the following section from the Dockerfile in the directory docker/apache:

```

CMD /usr/bin/consul-template -log-level info -consul consul:8500 \\
    -template "/etc/apache2/sites-enabled/000-default.ctmpl:/etc/apache2/sites-enabled\\
    /000-default.conf:apache2ctl -k graceful"

```

Consul Template executes the command apache2ctl -k graceful if there are new services or services have been removed and the configuration has therefore been changed. This causes Apache httpd to read the updated configuration and restart. However, open connections are not closed, but remain open, until communication is terminated. If no Apache httpd is running yet, one will be started. Thus, Consul Template takes control of Apache httpd and ensures that one instance of Apache httpd is always running in the Docker container.

To do this, Consul Template must run in the same Docker container as Apache httpd. This contradicts the Docker philosophy that only one process should run in one container. However, this cannot be avoided in the concrete example because these two processes are so closely related.

## Conclusion

Consul Template can ensure that a microservice can be reached from outside as soon as it has registered with Consul. To do this, the Apache httpd server does not need to know anything about the service discovery or Consul. It receives the information in its configuration and restarts.

## 15.5 Consul and Spring Boot

The Consul integration in Spring Boot is comparable to the integration of Eureka (see [section 14.2](#)). There relevant section of the configuration file application.properties is:

```
spring.application.name=catalog
spring.cloud.consul.host=consul
spring.cloud.consul.port=8500
spring.cloud.consul.discovery.preferIpAddress=true
spring.cloud.consul.discovery.instanceId=\${spring.application.name}:
\${spring.application.instance_id:\${random.value}}
```

The section configures the following values:

- spring.application.name defines the name under which the application is registered in Consul.
- spring.cloud.consul.host and spring.cloud.consul.port determine at which port and on which host Consul can be accessed.
- Via spring.cloud.consul.discovery.preferIpAddress, the services register with their IP address and not with the host name. This circumvents problems that arise because host names cannot be resolved in the Docker environment.
- spring.cloud.consul.discovery.instanceId assigns an unambiguous ID to each microservice instance, for example, for discriminating between instances for load balancing.

## Code Dependencies

In addition, a dependency to `spring-cloud-starter-consul-discovery` has to be inserted in `pom.xml` for the build. Moreover, the main class of the application that has the annotation `@SpringBootApplication` must also to be annotated with `@EnableDiscoveryClient`.

## Health Check with Spring Boot Actuator

Finally, the microservice must provide a health check. Spring Boot contains the Actuator module for this, which offers a health check as well as metrics. The health check is available at the URL `/health`. This is exactly the URL Consul requests, so it is enough to add a dependency to `spring-boot-starter-actuator` in the `pom.xml`. Specific health checks may need to be developed if the application depends on additional resources.

## Consul and Ribbon

Of course, a microservice must also use Consul to communicate with other microservices. For this, the Consul example uses the Ribbon library analogous to the Netflix example (see [section 14.4](#)). Ribbon has been modified in the Spring Cloud project so that it can also deal with Consul. Because the microservices use Ribbon, the rest of the code is unchanged compared to the Netflix example.

## 15.6 DNS and Registrar

The microservices have code dependencies on the Consul API for registering. This is not necessary. [Registrar<sup>169</sup>](#) can register Docker containers with Consul without the need for code. When the Docker containers are configured in such a way that they use Consul as a DNS server, the lookup of other microservices can also occur without code dependencies. This eliminates any dependencies on Consul in the project.

### Structure of the Example

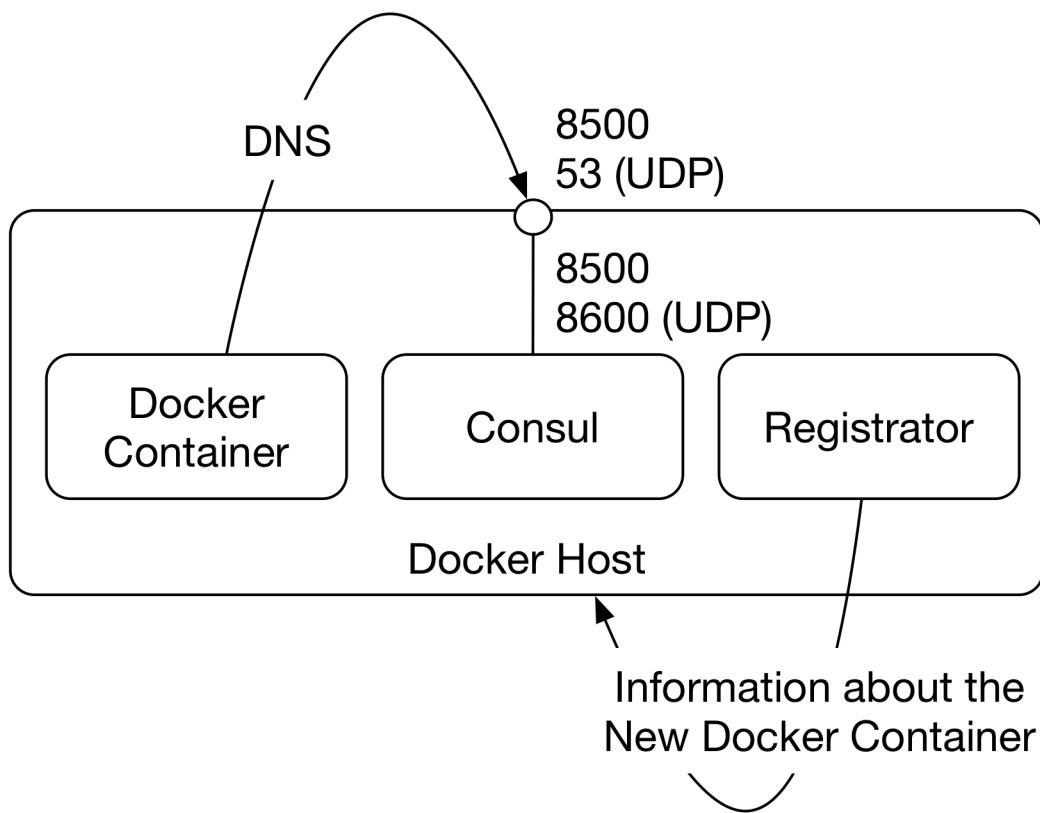


Fig. 15-4: Consul with DNS

[Figure 15-4](#) shows an overview of the approach.

<sup>169</sup><https://github.com/gliderlabs/registrator>

- *Registrar* runs in a Docker container. Via a socket, Registrar collects information from the Docker daemon about all newly launched Docker containers. The Docker daemon runs on the Docker host and manages all Docker containers.
- The *DNS interface of Consul* is bound to the UDP port 53 of the Docker host. This is the default port for DNS.
- The *Docker containers* use the Docker host as the DNS server.

The `dns` setting in the `docker-compose.yml` is configured to use the IP address in the environment variable `CONSUL_HOST` as the IP address of the DNS server. Therefore, the IP address of the Docker host has to be assigned to `CONSUL_HOST` before starting `docker-compose`. Unfortunately, it is not possible to configure DNS access from the Docker containers in a way that does not use this environment variable.

Registrar registers every Docker container started with Consul. Thus, not just the microservices but also the Apache httpd server or Consul itself can be found among the services in Consul.

Consul registers the Docker containers with `.service.consul` added to the name. In `docker-compose.yml`, `dns_search` is set to `.service.consul` so that this domain is always searched. In the end, the order microservice uses the URLs `http://msconsuldns_customer:8080/` and `http://msconsuldns_catalog:8080/` to access the customer and catalog microservices. Docker Compose creates the prefix `msconsuldns` for the name of the Docker containers to separate the project from other projects. Consul Template also uses the name to configure the Apache httpd for routing.

In this setup, Consul is responsible for load balancing. If there are several instances of a microservice, Registrar registers them all under the same name. Consul then returns one of the instances for each DNS request.

The executable example can be found at <https://github.com/ewolff/microservice-consul-dns> and the instructions for starting it at <https://github.com/ewolff/microservice-consul-dns/blob/master/HOW-TO-RUN.md>.

As a result of this approach, the microservices no longer contain any code dependencies on Consul. Therefore, with these technologies, implementing microservices with a programming language other than Java presents no problems.

## Configuration Is Also Possible in a Transparent Manner

`Envconsul`<sup>170</sup> also enables configuration data to be read from Consul and made available to the applications as environment variables. In this way, Consul can also configure the microservices without them having to include Consul-specific code.

## 15.7 Recipe Variations

Consul is very flexible and can be used in many different ways.

---

<sup>170</sup><https://github.com/hashicorp/envconsul>

## Combination with Frontend Integration

Like other approaches Consul can be combined with frontend integration (see [chapter 7](#)). SSI (server-side includes) with Apache httpd is especially simple to combine because an Apache httpd is already present in the system.

## Combination with Asynchronous Communication

Synchronous communication can be combined with asynchronous communication (see [chapter 10](#)). However, one type of communication should suffice normally. Atom or other asynchronous approaches via HTTP (see [chapter 12](#)) are easy to integrate into an HTTP-based system like Consul.

## Other Load Balancers

Rather than Apache httpd, a server like nginx or a load balancer like HAProxy can, of course, also be used for routing of requests from the outside. Such a load balancer can also replace Ribbon so that the internal load balancing is then configured with Consul Template. In this case only one type of load balancer is used for load balancing and routing. Unlike the Java library Ribbon, Apache httpd or nginx can be used with any programming language. Each microservice then has its own httpd or nginx instance so that no bottleneck or single point of failure is created.

## Service Meshes

[Chapter 23](#) discusses service meshes. They provide a lot of useful features – for example, for resilience, monitoring, tracing and logging. Istio is an example of a service mesh. A service mesh injects proxies into the communication between the microservices. Istio supports Consul to achieve that. So with Istio, Consul can be extended to become a complete platform for the operation of microservices.

## 15.8 Experiments

- Supplement the Consul system without DNS with an additional microservice.
  - A microservice used by a call center agent to create notes for a call can be a example. The call center agent should be able to select the customer.
  - Of course, you can copy and modify one of the existing microservices.
  - Register the microservice in Consul.
  - Ribbon has to be used to call the customer microservice. Ribbon does the lookup of the microservice in Consul. Otherwise, the microservice must be searched explicitly in Consul.
  - Package the microservice in a Docker image and reference the image in `docker-compose.yml`. There you can also specify the name of the Docker container.
  - Create in `docker-compose.yml` a link from the container with the new service to the container `consul`.

- The microservice must be accessible from the home page. To do this, you have to create a link in the file `index.html` in the Docker container `apache`. Consul Template automatically sets up the routing for the microservice in Apache as soon as the microservice is registered in Consul.
- Supplement the DNS Consul system (see [section 15.6](#)) with an additional microservice. This is similar to the previous experiment. There are just a few differences:
  - The call to the customer microservice has to use the host name `msconsuldns_customer`.
  - A registration in Consul is not necessary since Registrar automatically registers each Docker container.
- Currently, the Consul installation is not a cluster and is therefore unsuitable for a production environment. Change the Consul installation so that Consul runs in the cluster and the data from the service registry is saved to a hard disk. To do this, the configuration has to be changed in the Dockerfile and several instances of Consul have to be started. For more information, see <https://www.consul.io/docs/guides/bootstrapping.html>.
- Consul can also be used for saving the configuration of a Spring Boot application; see <https://cloud.spring.io/spring-cloud-consul/#spring-cloud-consul-config>. Use Consul to configure the example application with Consul.
- Replace Apache httpd with nginx, another web server, or HAProxy, for example. To do this, you need to create an appropriate Docker image or search for a matching Docker image in the [Docker hub<sup>171</sup>](#). In addition, the web server must be provided with reverse proxy extensions and configured with Consul Template. Additional documentation about Consul Template can be found on the [Github web page<sup>172</sup>](#). [Consul Template examples<sup>173</sup>](#) are also available for many systems.
- Try scaling and load balancing.
  - Increase the number of instances of a service – for example, with `docker-compose up --scale customer=2`.
  - Use the Consul dashboard to check if two customer microservices are running. It is available under port 8500 – for example, at <http://localhost:8500/> when Docker is running on the local computer.
  - Observe the logs of the order microservice with `docker logs -f msconsul_order_1` and see if different instances of the customer microservice are called. This should be the case because Ribbon is used for load balancing. To do this, you must trigger requests to the order application – for example, by reloading the home page.
- Add your own health check for one of the microservices. Check whether load balancing actually excludes a service when the health check is no longer successful.

---

<sup>171</sup><https://hub.docker.com>

<sup>172</sup><https://github.com/hashicorp/consul-template>

<sup>173</sup><https://github.com/hashicorp/consul-template/tree/master/examples>

## 15.9 Conclusion

Setting up a microservices system with Consul is another option for a synchronous system. This infrastructure meets the typical challenges of synchronous microservices as follows:

- *Service discovery* is covered by Consul. Consul is very flexible. Due to the DNS interface and Consul Template, it can be used with many technologies. This is particularly important in the context of microservices. Although a system might not need to use a variety of technologies from the start, being able to integrate new technologies in the long term is helpful.
- Consul is *more transparent* to use than Eureka. The Spring Cloud applications still need special Consul configurations. But Consul offers a configuration in Apache format for Apache httpd, so at least in this case Consul is transparent.
- If Registrator is used to register the microservices and Consul is used as the DNS server, Consul is *fully transparent* and can be used without any code dependencies. With Envconsul, Consul can even configure the microservices without code dependencies.
- Consul can be used to *configure* the microservices. In this way, with only one technological approach, both service discovery and configuration can be implemented.
- *Resilience* is not implemented in this example.
- *Routing* with Apache httpd is a relatively common approach. This reduces the technological complexity, which is quite high in a microservices system by default. With the large number of new technologies and a new architectural approach, it is certainly helpful to cover at least some areas with established approaches.
- *Load balancing* is implemented with Ribbon like in the Netflix example (see [section 14.4](#)). However, it is no problem to provide each microservice instance with an Apache httpd, for example, which is configured by Consul Template in such a manner that to provide load balancing for outbound calls. For Consul, DNS Consul even implements load balancing transparently with the DNS server. So, in that case, no additional technology for load balancing is needed.

### Comparison to Netflix

The technology stack from this example has the advantage that it also supports heterogeneous microservices systems because there are no more code dependencies on Consul if using DNS. Consul as a service discovery technology is much more powerful than Eureka with the DNS interface Consul provides and Consul Template. Apache is a standard, widely used reverse proxy. It is therefore probably more mature than Zuul. Also, Zuul is not really supported any more, whereas Apache is still one of the most broadly used web servers. For resilience, the stack does not really offer a good solution. However, it can still be combined with libraries such as Hystrix.

The main benefit of the Consul technology stack is its independence from a concrete language and environment. The Netflix stack is based on Java and it is hard to integrate other languages. Also, Netflix has discontinued several of the projects – that is, Hystrix and Zuul. So there are quite a few disadvantages of the Netflix stack, but not really a lot of advantages. The Consul stack is therefore usually preferable over the Netflix stack.

## Advantages

- Consul does not have a Java focus but supports many different technologies.
- Consul also supports DNS.
- Consul Template can even configure many services (Apache httpd) transparently via configuration files.
- Entirely transparent registration and service discovery with Registrar and DNS are possible.
- The use of well-established technologies such as Apache httpd reduces the risk.

## Challenges

- Consul is written in Go. Therefore, monitoring and deployment differ from Java microservices.

# 16 Concept: Microservices Platforms

The following chapters describe microservices platforms. Readers learn that:

- Microservices platforms provide support for the operation and also for the communication of microservices.
- PaaS (Platform as a Service) cloud offerings and Docker scheduler are examples for microservices platforms.
- Microservices platforms have their own advantages and disadvantages, which make them superior to other approaches in some scenarios.

## 16.1 Definition

The platforms in the next chapters differ from all other technologies presented so far in that they not only enable the communication of microservices, but also support aspects of operation such as deployment, monitoring, or log analysis.

### Support for HTTP and REST

The platforms support HTTP and REST with load balancing, routing, and service discovery. But they can also be supplemented with other communication mechanisms. This makes it possible to use the platforms for setting up asynchronous systems. However, this chapter mainly considers synchronous communication. Microservices on platforms that use this type of communication differ most from microservices on different infrastructures concerning their mode of operation.

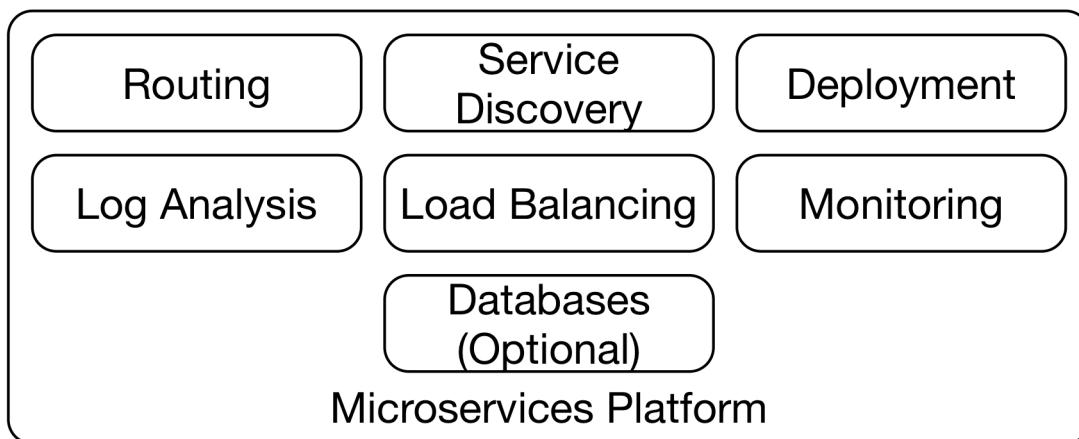


Fig. 16-1: Features of Microservices Platforms

## Expenditure for Installation and Operation

Microservices platforms are very powerful, but consequently also very complex. They make it much easier to work with microservices, but installing and operating the platform itself can really be a challenge. If the platform is running in the public cloud, the complex installation and operation does not play a role, because the responsibility lies with the operator of the public cloud. However, if an installation is carried out in the company's own data center, the operations team has to make the effort.

That being said, the effort for installing the platform has to be made only once. After that, deploying the microservices is much easier. This means that the cost of installing the platform will be amortized quickly.

Only limited operations support is then necessary when installing the microservices. In this way, microservices can be deployed quickly and easily even if operations cannot support each deployment of each microservice.

## Migration to a Microservices Platform

In contrast to the solutions shown so far, microservices platforms require a fundamental change in the operation and installation of the applications. The other examples can be run on virtual machines or even physical servers and can be combined with existing deployment tools. Because the microservices platforms also cover the deployment and operation of the microservices, they include features for which operations teams already have established tools in most cases. Therefore, the use of a microservices platform is a bigger step than the use of other technologies.

Such a move can be deterrent for conservative operations teams. In addition, the introduction of microservices often leads to organizational changes and many new technologies in addition to a new architecture. It can be helpful if a complex microservices platform does not also have to be introduced in this situation.

## Influence on the Macro Architecture

Of course, a platform supports only certain technologies. In addition to the programming language, these are above all the technologies for deployment, monitoring, logging, and so on. The platform, therefore, defines large parts of the technical infrastructure.

For this reason, microservices platforms have a relationship to the macro architecture (see [chapter 2](#)).

- The macro architecture should state the *requirements of the platform*. This ensures that the platform can actually run the microservices. So the platform specifies, for example, deployment and logging in the macro architecture.
- The microservices platform *restricts the options* the team can choose in the macro architecture. After all, it is impossible to operate microservices that cannot be run on the platform. For example, the platform can define restrictions concerning the programming language.

- The microservices platform can *enforce* compliance with macro architecture rules. If developers disregard the rules, the microservices simply cannot run on the platform. This ensures that the rules are actually observed.

## Specific Platforms

The two following chapters show two approaches for microservices platforms.

- *Kubernetes* (see chapter 17) can run Docker containers and solves challenges such as load balancing, routing, and service discovery at the network level. It is quite flexible because it is able to run arbitrary Docker containers. With Operators<sup>174</sup> or Helm<sup>175</sup>, other services can be integrated into Kubernetes, for example, for monitoring.
- Cloud Foundry (see chapter 18) supports applications. All you have to do is provide Cloud Foundry with, as an example, a Java application. Cloud Foundry creates a Docker container from it, which can then be executed. Cloud Foundry also solves load balancing, routing, and service discovery. In addition, Cloud Foundry includes additional infrastructure such as databases.

## 16.2 Variations

Microservices platforms appear to be particularly suitable for synchronous microservices and REST communication, because they offer particularly good support for this. However, the platforms can be extended to allow other communication mechanisms. Frontend integration can be implemented with these platforms. Client frontend integration is completely independent of the platform used. Only with server-side frontend integration would the server have to be installed and operated on the platform.

The platform can also cover the operational aspect for asynchronous microservices. Better support for the operation of the microservices alone is a good reason to use the platforms. Operation is one of the most important challenges with microservices. This aspect is independent of the communication mechanism used.

## Physical Hardware

So the question arises whether other environments for the operation of microservices exist. A theoretical alternative to the platforms is physical hardware. However, physical hardware is hardly used any more for cost reasons.

---

<sup>174</sup><http://coreos.com/operators>

<sup>175</sup><https://helm.sh/>

## Virtual Hardware

Also, virtual hardware is inflexible and heavyweight, as already discussed in [section 4.7](#). So the only alternative to a platform is “Docker without scheduler,” as discussed in [section 4.7](#). In this scenario, Docker containers are installed on classic servers.

In this case, the macro architecture standardizes the operation (see [section 2.2](#)) to use only one technology for log analysis or monitoring and thus to work efficiently. Microservices platforms already have such features. So you do not have to build support for logging or monitoring yourself, but you can use these parts of the platform. This can be the simpler solution because implementing log analysis for many microservices can be very costly.

Resilience and other features such as load balancing have to be implemented at the virtual machine level because the Docker infrastructure does not offer these. In the end, it can happen that the features of a microservices platform such as log analysis, monitoring, reliability, and load balancing are recreated step by step instead of just installing and using a prepackaged microservices platform.

## 16.3 Conclusion

Microservices platforms appear to be very helpful due to the large number of features, even though the operation of these platforms can be complex. In practice, Kubernetes is a very important platform for the operation of microservices, whereas PaaS such as Cloud Foundry are not in focus despite their convincing features.

Microservices platforms should definitely be considered for microservices because they represent a significant simplification and complete solution to typical microservices challenges. The only reason against a microservices platform is the high cost of installation. In case of a small number of microservices or at the beginning of a project, the installation of a microservices platform can be bypassed. In addition, this effort is eliminated if using a public cloud offering.

# 17 Recipe: Docker Containers with Kubernetes

This chapter describes Kubernetes, a runtime environment for Docker containers. The reader discovers the following points in this chapter:

- Kubernetes can run Docker containers in a cluster and comprises a complete infrastructure for microservices.
- Kubernetes does not introduce code dependencies into the example.
- MOMs or other tools can also run on Kubernetes.

## 17.1 Kubernetes

Kubernetes<sup>176</sup> is increasingly gaining importance as runtime environment for the development and the operation of microservices.

### Licence and Community

Kubernetes is an open source project and under the Apache licence. It is managed by the Linux foundation and was originally created at Google. An extensive ecosystem has emerged around Kubernetes, offering various extensions.

### Kubernetes Versions

Different Kubernetes variants are available:

Minikube<sup>177</sup> is a version of Kubernetes for installing a test and development system on a laptop. But there are many more versions<sup>178</sup>, which can either be installed on servers or directly used as cloud offerings. kops<sup>179</sup> is a tool which enables the installation of a Kubernetes cluster in many different types of environments like AWS (Amazon Web Services). Amazon Elastic Container Service for Kubernetes (Amazon EKS)<sup>180</sup> provides Kubernetes clusters on AWS. Google Cloud also supports Kubernetes with the Google Container Engine<sup>181</sup>. Microsoft Azure provides the Azure Container Service<sup>182</sup> and IBM Bluemix the IBM Bluemix Container Service<sup>183</sup>.

---

<sup>176</sup><https://kubernetes.io/>

<sup>177</sup><https://github.com/kubernetes/minikube>

<sup>178</sup><https://kubernetes.io/docs/getting-started-guides/>

<sup>179</sup><https://github.com/kubernetes/kops>

<sup>180</sup><https://aws.amazon.com/eks/>

<sup>181</sup><https://cloud.google.com/container-engine>

<sup>182</sup><https://azure.microsoft.com/en-us/services/container-service/>

<sup>183</sup>[https://console.ng.bluemix.net/docs/containers/container\\_index.html](https://console.ng.bluemix.net/docs/containers/container_index.html)

## Features

As already mentioned, Kubernetes offers a platform based on Docker that supports important features such as:

- Kubernetes runs Docker containers in a *cluster* of nodes. Thus, Docker containers can use all resources in the cluster.
- In the event of a failure, Docker containers can be restarted. This is even possible if the original node on which the container was running is no longer available. In this way, the system achieves *fail-safety*.
- Kubernetes also supports *load balancing* and can distribute the load between multiple nodes.
- Kubernetes supports *service discovery*. Microservices that run in Docker containers can easily find and communicate with each other via Kubernetes.
- Finally, because Kubernetes works on the level of Docker containers, the microservices have *no code dependencies* to Kubernetes. This is not only elegant, but also means that a Kubernetes system supports virtually all programming languages and frameworks for the implementation of microservices.

## Kubernetes Concepts

In addition to the already discussed Docker concepts (see [chapter 5](#)), Kubernetes introduces several additional concepts.

- *Nodes* are the servers on which Kubernetes runs. They are organized in a cluster.
- A *pod* consists of multiple Docker containers that together provide a service. For example, this can be a container with a microservice together with a container for log processing. The Docker containers in a pod can share Docker volumes and thus efficiently exchange data. All containers belonging to one pod always run on one node. So to scale the system to more nodes, more instances of the pod need to be started and distributed on the nodes.
- A *replica set* ensures that always a certain number of instances of a pod runs. This allows the load to be distributed to the pods. In addition, the system is fail-safe. If a pod fails, a new pod is automatically started.
- A *deployment* generates a replica set and provides the required Docker images.
- *Services* make pods accessible. The services are registered under one name in the DNS and have a fixed IP address under which they can be contacted throughout the cluster. In addition, services enable routing for requests from the outside.
- Kubernetes is *declarative* – that is, the configuration defines a desired state. Kubernetes makes sure that the system fits the desired state. So a replica set actually defines the number of pods that should be running, and it is up to Kubernetes to make the number of actually running pods match that number. If the configuration of the replica set changes, the number of running pods is adjusted accordingly. And if some of the pods crash, enough pods are started to match the declaration in the replica set.

## 17.2 The Example with Kubernetes

The services in this example are identical with the examples presented in the two preceding chapters (see [section 14.1](#)).

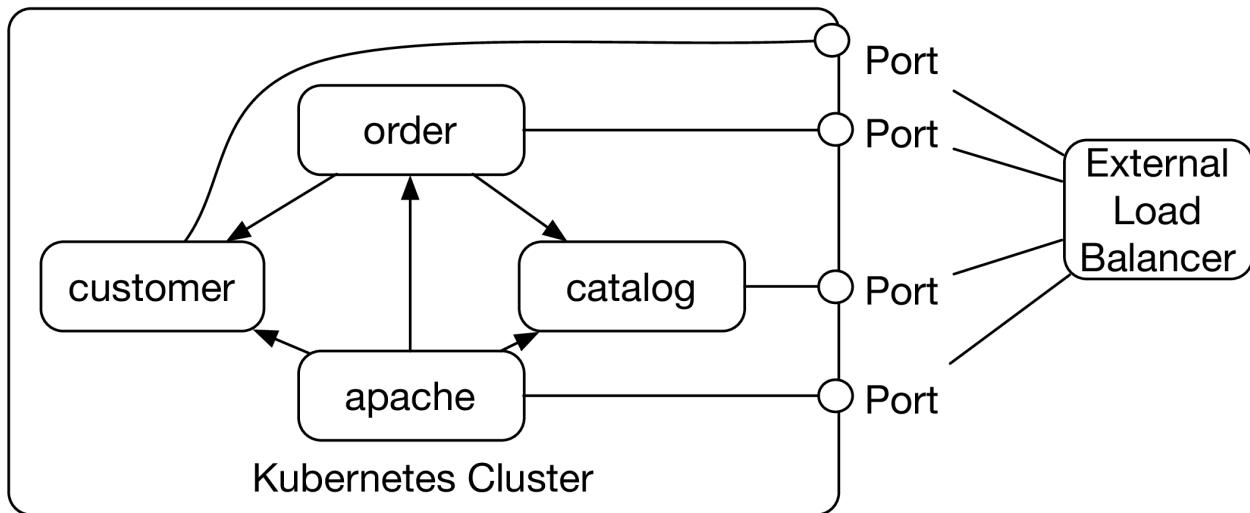


Fig. 17-1: The Microservices System in Kubernetes

- The *catalog* microservice manages the information about the items. It provides an HTML UI and a REST interface.
- The *customer* microservice stores the customer data and also provides an HTML UI and a REST interface.
- The *order* microservice can receive new orders. It provides an HTML UI and uses the REST interfaces of the catalog and customer microservice.
- There is also an *Apache web server* for facilitating access to the individual microservices. It forwards the calls to the respective services.

[Figure 17-1](#) shows how the microservices interact. The *order* microservice communicates with the *catalog* and *customer* microservice. The *Apache httpd* server communicates with all other microservices to display the HTML UIs.

In addition, the microservices are accessible from the outside via node ports. On each node in the cluster, request to a specific port is forwarded to the service. However, the port numbers are assigned by Kubernetes, so no port number appears in the figure. Also, a load balancer is set up by the Kubernetes service to distribute the load across Kubernetes nodes.

## Implementing Microservices with Kubernetes

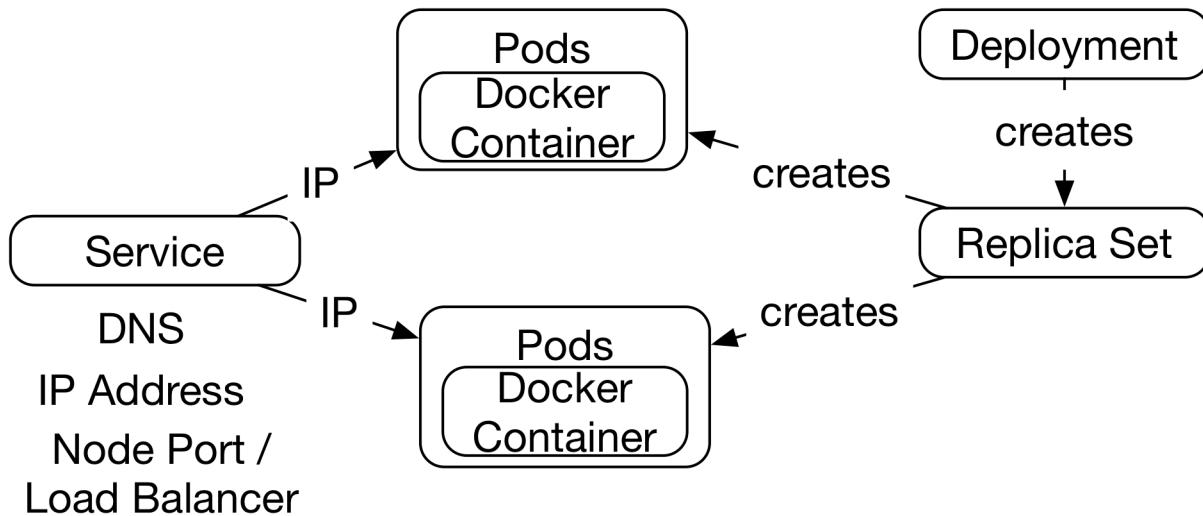


Fig. 17-2: A Microservice in Kubernetes

Figure 17-2 shows the interaction of the Kubernetes components for a microservice. A *deployment* creates a *replica set* with the help of Docker images. The *replica set* starts one or multiple pods. The *pods* in the example only comprise a single Docker container in which the microservice is running.

## Service Discovery

A *service* makes the replica set accessible. The service provides the pods with an IP address and a DNS record. Other pods communicate with the service by reading the IP address from the DNS. Thereby, Kubernetes implements service discovery with DNS. In addition, microservices receive the IP addresses of other microservices via environment variables. Thus, they could also use this information to access the service.

## Fail-Safety

The microservices are so fail-safe because the replica set ensures that a certain number of pods is always running. So if a pod fails, a new one is started.

## Load Balancing

Load balancing is also covered. The number of pods is determined by the replica set. The service implements load balancing. All pods can be accessed with the same IP address which the service defines. Requests go to this IP address, but are distributed to all instances. The service implements this functionality by interfering with the IP network between the Docker containers. Because the IP address is cluster-wide unique, this mechanism works even if the pod is moved from one node to another.

Kubernetes does not implement load balancing at the DNS level. If it did, a different IP addresses would be returned for the same service name for each DNS lookup, so that the load would be distributed during DNS access. However, such an approach presents a number of challenges. DNS supports caching. If a different IP address is to be returned each time a DNS access occurs, the caching must be configured accordingly. However, problems still occur frequently because caches are not invalidated in time.

## **Service Discovery, Fail-Safety, and Load Balancing without Code Dependencies**

For load balancing and service discovery, no special code is necessary. A URL like `http://order:8080/` suffices. Accordingly, the microservices do not use any special Kubernetes APIs. There are no code dependencies to Kubernetes, and no specific Kubernetes libraries are used.

### **Routing with Apache httpd**

In the example, Apache httpd is configured as a reverse proxy. It therefore routes access from the outside to the correct microservice. Apache httpd leaves load balancing, service discovery, and fail-safety to the Kubernetes infrastructure.

### **Routing with Node Ports**

Services also offer a solution for routing – that is, external access to microservices. The service generates a node port. Under this port the services are available on every Kubernetes node. If the pod that implements the service is not available on the called Kubernetes node, Kubernetes forwards requests to another Kubernetes node where such a pod is running. In this way, an external load balancer can distribute the load to the nodes in the Kubernetes cluster. The requests are simply distributed to the node port of the service on all nodes in the cluster. The service type `NodePort` is used for this purpose, which must be specified when creating the service.

### **Routing with Load Balancers**

Kubernetes can create load balancers in a Kubernetes production environment. In an Amazon environment, for example, Kubernetes configures an ELB (Elastic Load Balancer) to access the node ports in the cluster. The service type `LoadBalancer` is used for this purpose.

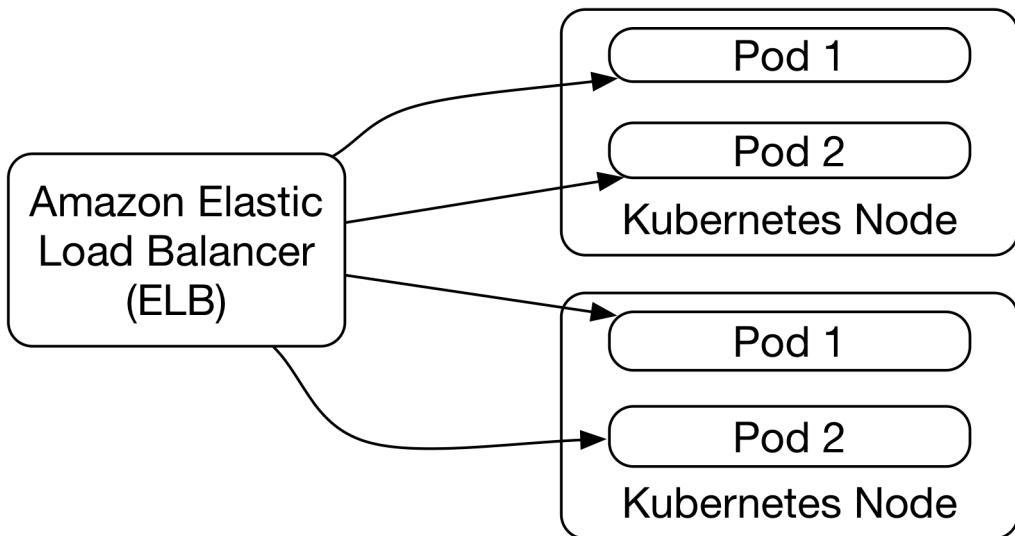


Fig. 17-3: Kubernetes Service Type LoadBalancer in der Amazon Cloud

The services in the example are of the type `LoadBalancer`. However, if they run on Minikube, they are treated as services of type `NodePort` because Minikube cannot configure and provide load balancers.

### Routing with Ingress

Kubernetes offers an extension called [Ingress<sup>184</sup>](#), which can configure and alter the access of services from the Internet. Ingress can also implement load balancing, terminate SSL or implement virtual hosts. This behavior is implemented by an Ingress controller.

In the example, the Apache web server forwards requests to the individual microservices. This could also be done with a Kubernetes Ingress. Then the routing would be done by a part of the Kubernetes infrastructure, which might work better – in the cluster, for example – and is easier to configure. The configuration is done in Kubernetes YAML files and supports other Kubernetes parts like, such as services. However, the example contains a few static web pages that the Apache web server provides. So the Apache web server has to be configured in any event. Using an Ingress is therefore not a huge advantage.

## 17.3 The Example in Detail

[Section 0.4](#) describes what software has to be installed for starting the example.

You can access the example at <https://github.com/ewolff/microservice-kubernetes>. <https://github.com/ewolff/microservice-kubernetes/blob/master/HOW-TO-RUN.md> explains in detail how to install the required software for running the example.

The following steps are necessary for running the example:

<sup>184</sup><https://kubernetes.io/docs/concepts/services-networking/ingress/>

- [Minikube<sup>185</sup>](https://github.com/kubernetes/minikube#installation) as minimal Kubernetes installation has to be installed. Instructions for this can be found at <https://github.com/kubernetes/minikube#installation>.
- [kubectl<sup>186</sup>](https://kubernetes.io/docs/tasks/tools/install-kubectl/), a command line tool for handling Kubernetes, also has to be installed. <https://kubernetes.io/docs/tasks/tools/install-kubectl/>. describes its installation.
- The script `docker-build.sh` generates the Docker images for the microservices and uploads them into the public Docker hub. This step is optional because the images are already available on the Docker hub. It has to be performed only when changes are introduced to the code or to the configuration of the microservices. Before starting the script, the Java code has to be compiled with `./mvnw clean package` (macOS, Linux) or `mvnw.cmd clean package` (Windows) in directory `microservice-kubernetes-demo`. See [appendix B](#) for more details on Maven and how to troubleshoot the build. Then the script `docker-build.sh` creates the images with `docker build`, with `docker` tag they receive a globally unique name and `docker push` uploads them into the Docker hub. Using the public Docker hubs spares the installation of a Docker repository and thereby facilitates the handling of the example.
- The script `kubernetes-deploy.sh` deploys the images from the public Docker hub in the Kubernetes cluster and thereby generates the pods, the deployments, the replica sets, and the services. For this, the script uses the tool `kubectl`. `kubectl run` serves for starting the image. The image is downloaded at the indicated URL in the Docker hub. In addition, it defines which port the Docker container should provide. So `kubectl run` generates the deployment, which creates the replica set and thereby the pods. `kubectl expose` generates the service that accesses the replica set and thus creates the IP address, DNS entry, node port, or load balancer.

This excerpt from `kubernetes-deploy.sh` shows the use of the tools using the catalog microservice as example.

```
#!/bin/sh
if [ -z "$DOCKER_ACCOUNT" ]; then
  DOCKER_ACCOUNT=ewolff
fi;
...
kubectl run catalog \\
--image=docker.io/$DOCKER_ACCOUNT/microservice-kubernetes-demo-catalog:latest
\\
--port=80
kubectl expose deployment/catalog --type="LoadBalancer" --port 80
...
```

An alternative is to use Kubernetes YAML files. They describe the desired state of deployments and services. For example, here is the part of `microservices.yaml` for the catalog microservices.

---

<sup>185</sup><https://github.com/kubernetes/minikube>

<sup>186</sup><https://kubernetes.io/docs/user-guide/kubectl-overview/>

...

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    run: catalog
  name: catalog
spec:
  replicas: 1
  selector:
    matchLabels:
      run: catalog
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        run: catalog
    spec:
      containers:
        - image: docker.io/ewolff/microservice-kubernetes-demo-catalog:latest
          name: catalog
          ports:
            - containerPort: 8080
          resources: {}
status: {}
```

...

...

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    run: catalog
  name: catalog
spec:
  ports:
    - port: 8080
```

```
protocol: TCP
targetPort: 8080
selector:
  run: catalog
type: LoadBalancer
status:
  loadBalancer: {}
```

...

The information in the YAML file is very similar to the parameters of the previous commands. Using `kubectl apply -f microservices.yaml`, all the services and deployment would be created in the Kubernetes cluster. The same command would be used to update the services and deployments after any changes.

## Some Minikube Commands

`minikube dashboard` displays the dashboard in the web browser, which displays the deployments and additional elements of Kubernetes. This makes it easy to understand the state of the services and deployments ([Figure 17-4](#)).

Name	Labels	Pods	Age	Images
hystrix-das...	run: hyst...	1 / 1	11 days	docker.io/...
customer	run: cus...	1 / 1	11 days	docker.io/...
order	run: order	1 / 1	11 days	docker.io/...
apache	run: apa...	1 / 1	11 days	docker.io/...
catalog	run: cat...	1 / 1	11 days	docker.io/...

Name	Status	Restarts	Age

Fig. 17-4: Kubernetes Dashboard

`minikube service apache` opens the Apache service in the web browser and thereby offers access to the microservices in the Kubernetes environment.

The script `kubernetes-remove.sh` can be used to delete the example. It uses `kubectl delete service` for deleting the services, and `kubectl delete deployments` for deleting the deployments.

## 17.4 Additional Kubernetes Features

Kubernetes is a powerful technology with many features. Here are some examples of additional Kubernetes features:

### Monitoring with Liveness and Readiness Probes

- Kubernetes recognizes the failure of a pod via [Liveness Probes<sup>187</sup>](#). A custom Liveness Probe can be used to determine when a container is started anew depending on the needs of the application.

<sup>187</sup><https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/>

- A [Readiness Probe](#)<sup>188</sup> indicates whether a container can process requests or not. For example, if the application is blocked by processing a large amount of data or has not yet started completely, the Readiness Probe can report this state to Kubernetes. In contrast to a Liveness Probe, the container is not restarted as a result of a failed Readiness Probe. Kubernetes assumes that after some time the pod will signal via the Readiness Probe that it can handle requests.

## Configuration

- The configuration of applications is possible with [ConfigMaps](#)<sup>189</sup>. The configuration data is provided to the applications, for example, as values in environment variables.

## Separating Kubernetes Environments with Namespaces

- Kubernetes environments can be separated with [Namespaces](#)<sup>190</sup>. Namespaces are virtual clusters so that, for example, services and deployments are completely separated. That allows different environments to coexist – that is, multiple teams can share a cluster and use namespaces to separate their environments. Also, this makes it possible to separate the microservices from infrastructure such as databases or monitoring infrastructure. That way, users only see the services and deployments in which they are interested.

## Applications with State

- Kubernetes can also handle applications that have state. Applications without state can be simply restarted on another node in a cluster. This facilitates fail-safety and load balancing. If the application has state and therefore requires certain data in a Docker volume, the required Docker volumes must be available on each node the application runs on. This makes the handling of such applications more complex. Kubernetes offers [persistent volumes](#)<sup>191</sup> and [stateful sets](#)<sup>192</sup> for dealing with this challenge.
- Another option for dealing with applications with state are [operators](#)<sup>193</sup>. They allow the automated installation of applications with state. For example, the [Prometheus operator](#)<sup>194</sup>, can install the monitoring system Prometheus ([chapter 20](#)) in a Kubernetes cluster. It introduces Kubernetes resources for Prometheus components such as Prometheus, ServiceMonitor, or Altermanger. With the Prometheus operator, these key words are used by the Kubernetes configuration rather than pods, services, or deployments. The operator also determines how Prometheus saves the monitoring data, thus solving a key challenge.

<sup>188</sup><https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/#define-readiness-probes>

<sup>189</sup><https://kubernetes.io/docs/tasks/configure-pod-container/configmap/>

<sup>190</sup><https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/>

<sup>191</sup><https://kubernetes.io/docs/concepts/storage/persistent-volumes/>

<sup>192</sup><https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#stable-storage>

<sup>193</sup><https://coreos.com/operators>

<sup>194</sup><https://github.com/coreos/prometheus-operator>

## Extensions with Helm

- Kubernetes offers a complex ecosystem with numerous extensions. [Helm<sup>195</sup>](#) can install extensions as [Charts<sup>196</sup>](#) and thereby assumes the functionality of a package manager for Kubernetes. This extensibility is an important advantage of Kubernetes.

[Section 23.2](#) shows how Helm can be used to simplify the deployment of a microservice. Using a Helm chart requires only the name of the microservice to be defined. The Kubernetes configuration is generated with a template and the provided name. This makes the installation of the microservices much easier and more uniform.

## 17.5 Recipe Variations

Kubernetes offers above all a runtime environment for Docker containers and is therefore very flexible.

### MOMs in Kubernetes

The example in this chapter uses communication with REST. Of course, it is also possible to operate a MOM such as Kafka ([section 11](#)) in Kubernetes. However, MOMs store the transmitted messages to guarantee delivery. Kafka even saves the complete history. Reliable storage of data in a Kubernetes cluster is feasible, but not easy. Using a MOM other than Kafka does not solve the problem. All MOMs store messages permanently to guarantee delivery. Thus, for reliable communication with a MOM, Kubernetes has to store the data reliably and scalable.

### Frontend Integration with Kubernetes

Kubernetes can be quite easily combined with frontend integration ([chapter 7](#)), because Kubernetes does not make any assumptions about the UI of the applications. Client-side frontend integration does not place any demands on the backend. For server-side integration, a cache or web server must be hosted in a Docker container. However, these servers do not store any data permanently, so they can easily be operated in Kubernetes.

### Docker Swarm and Docker Compose

[Section 4.7](#) already introduces some alternatives to Kubernetes for operating Docker containers in clusters. Kubernetes offers a very powerful solution and is further developed by many companies in the container area. However, Kubernetes is also very complex due to its many features. A cluster with Docker Compose and Docker Swarm can be a simpler but also less powerful alternative. However, Docker Swarm and Compose at least also offer basic features such as service discovery and load balancing.

---

<sup>195</sup><https://helm.sh/>

<sup>196</sup><https://github.com/kubernetes/charts/>

## Docker versus Virtualization

As Kubernetes takes over cluster management, it includes features that virtualization solutions also offer. This can also lead to operational concerns, because reliable cluster operation is a challenge. Adding another technology in this area is often viewed critically. When deciding against Kubernetes, Docker can still be used without a scheduler (see [section 4.7](#)). But then the Kubernetes features for service discovery, load balancing, and routing are missing, requiring implementing them by different means.

## 17.6 Experiments

- Supplement the Kubernetes system with an additional microservice.
  - A microservice used by a call center agent to create notes for a call can be used as an example. The call center agent should be able to select the customer.
  - For calling the customer microservice, the host name `customer` has to be used.
  - Of course, you can copy and modify one of the existing microservices.
  - Package the microservice in a Docker image and upload it into the Docker repository. This can be done by adapting the script `docker-build.sh`.
  - Adapt `kubernetes-deploy.sh` in such a manner that the microservice is deployed.
  - Adapt `kubernetes-remove.sh` in such a manner that the microservice is deleted.
- <https://kubernetes.io/docs/getting-started-guides/> is an interactive tutorial that shows how to use Kubernetes. It complements this chapter well. Work through the tutorial to get an impression of the Kubernetes features.
- Kubernetes supports rolling updates. A new version of a pod is rolled out in such a way that no interruptions to the service occur. See <https://kubernetes.io/docs/tasks/run-application/rolling-update-replication-controller/>. Run a rolling update! To do this you need to create a new Docker image. The scripts for compiling and delivering to the Docker hub are included in the example.
- Cloud providers such as Google or Microsoft offer Kubernetes infrastructures, see <https://kubernetes.io/docs/getting-started-guides/#hosted-solutions>. Make the example work in such an environment! You can use the scripts without changes because `kubectl` also supports these technologies.
- Test the load balancing in the example.
  - `kubectl scale` alters the number of pods in a replica set. `kubectl scale -h` indicates what options are available. For example, scale the replica set `catalog`.
  - `kubectl get deployments` shows how many pods are running in the respective deployment.
  - Use the service. For example `minikube service apache` opens the web page with links to all microservices. Select the order microservice and display the orders.
  - `kubectl describe pods -l run=catalog` displays the running pods. There you can also find the IP address of the pods in a line that starts with `IP`.

- Log into the Kubernetes node with `minikube ssh`. To read out the metrics, you can then use a command such as `curl 172.17.0.8:8080/metrics`. You have to adapt the IP address. This way, you can display the metrics of the catalog pods that Spring Boot creates. For example, the metrics contain the number of requests that have been responded to with an HTTP 200 status code (OK). If you now use the catalog microservice via the web page, each pod should process some of the requests and therefore the metrics of all pods should go up.
  - Also use `minikube dashboard` for observing the information in the dashboard.
- The example currently uses the public Docker hub. Install your own Docker registry<sup>197</sup>. Save the Docker images of the example in the registry and deploy the example from this registry.
  - At <https://github.com/GoogleCloudPlatform/kubernetes-workshops>, you can find material for a Kubernetes workshop to further familiarize yourself with this system.
  - Port the Kafka example (see chapter 11) or the Atom example (see chapter 12) to Kubernetes. This shows how asynchronous microservices can also run in a Kubernetes cluster. Kafka stores data, which can be difficult in a Kubernetes system. Explore how to run a Kafka cluster in a Kubernetes system in production.
  - Use `kubectl logs -help` for familiarizing yourself with the log administration in Kubernetes. Take a look at the logs of at least two microservices.
  - Use `kail`<sup>198</sup> for displaying the logs of some pods.

## 17.7 Conclusion

Kubernetes solves the challenges of synchronous microservices as follows:

- DNS offers *service discovery*. Thanks to DNS, microservices can be used from any programming language and even transparently. However, DNS only provides the IP address, so the port must be known. No code is required to register the services. When you start the service, a DNS record is created automatically.
- Kubernetes ensures *load balancing* by distributing the traffic for the IP address of the Kubernetes service to the individual pods on the IP level. This is transparent for callers and for the called microservice.
- Kubernetes covers *routing* via the load balancer or node ports of the services. This is also transparent for the microservices.
- Kubernetes offers *Resilience* via the restarting of containers and load balancing. In addition, a library such as Hystrix can be useful, for example, for implementing timeouts or circuit breakers. A proxy like `Envoy`<sup>199</sup> can be an alternative to Hystrix. Envoy is also part of Istio (see section 23.3) and implements resilience for Istio.

<sup>197</sup><https://docs.docker.com/registry/>

<sup>198</sup><https://github.com/boz/kail>

<sup>199</sup><https://github.com/lyft/envoy>

Within one package, Kubernetes offers complete support for microservices including service discovery, load balancing, resilience, and scalability in the cluster. In this way, Kubernetes solves many challenges arising during the operation of a microservices environment. The code of the microservices remains free of these concerns. No dependencies on Kubernetes are introduced into the code.

This is attractive, but it also represents a fundamental change. Wheres Consul or the Netflix stack also run on virtual machines or even bare metal, Kubernetes requires everything to be packed in Docker containers. This can be a fundamental change compared to an existing mode of operation and can make the migration to this environment harder.

## Advantages

- Kubernetes solves most typical challenges of microservices (load balancing, routing, service discovery).
- The code has no dependencies on Kubernetes.
- Kubernetes also covers operation and deployment.
- The Kubernetes platform enforces standards and thereby the definition of a macro architecture.

## Challenges

- A complete change of operation is required to use Kubernetes rather than other log or deployment technologies.
- Kubernetes is very powerful, but thus also very complex.

# 18 Recipe: PaaS with Cloud Foundry

This chapter introduces PaaS (Platform as a Service) as a runtime environment for microservices.

The text answers the following questions:

- What is a PaaS and how does it differ from other runtime environments?
- Why is a PaaS suited for microservices?

The chapter introduces Cloud Foundry as a concrete example for a PaaS .

## 18.1 PaaS: Definition

Fundamentally different services are available in the cloud, such as:

### IaaS

An IaaS (Infrastructure as a Service) offers virtual computers on which software has to be installed. Thus, IaaS is a simple solution which essentially corresponds to classical virtualization. The decisive difference is the billing model, which for IaaS bills only the actually used resource per hour or per minute.

### SaaS

SaaS (Software as a Service) is a cloud offer from which software can be rented, for example, for word processing or financial accounting. For software development version controls or continuous integration servers can be purchased as SaaS.

### PaaS

PaaS stands for Platform as a Service. PaaS offers a platform on which custom software can be installed and run. The developer provides only the application to the PaaS. The PaaS makes the application executable. Unlike Docker (see [chapter 5](#)) and Kubernetes (see [chapter 17](#)), the operating system and the software installed on it is not under the developer's control. For the microservices examples, the `Dockerfile` specifies that an Alpine Linux distribution and a specific version of the Java Virtual Machine (JVM) should be used. This is no longer necessary with a PaaS. The JAR file contains the executable Java application and thus everything that the PaaS needs.

To start the application in the runtime environment, the PaaS can create a Docker container. But the decision about which JVM and Linux distribution to use is up to the PaaS. The PaaS must be prepared to run different types of applications. .NET applications and Java applications each require their own virtual machine, whereas Go applications do not. The PaaS must create the appropriate environment. Nowadays, PaaS is flexible enough to support different environments and even roll your own environment. Therefore, you can usually define which JDK should be used.

## PaaS Restricts Flexibility and Control

Developers have less control over the Docker images. However, the question is whether a developer should spend time with the selection of the JVM and the Linux distribution in the first place. Often, these issues lie with operations anyway. Some PaaS offer the possibility to configure the Linux distribution or JVM. Often, the user can even define complete runtime environments. Nevertheless, flexibility is limited.

To run existing applications, the PaaS might not be flexible enough. For example, an application might need a specific JVM version that the PaaS does not support. Microservices, however, are usually newly developed, so this disadvantage does not play a great role.

## Routing and Scaling

The PaaS must forward requests from the user to the application, so routing is a feature of a PaaS. Similarly, PaaS can usually scale applications individually, ensuring scalability.

## Additional Services

Many PaaS can provide the application with additional services such as databases. Also, typical features for operation such as support for analyzing log data or monitoring are often part of a PaaS.

## Public Cloud

PaaS are offered in the public cloud. The developers have only to deploy their application into the PaaS and then the application runs on the Internet. This is a very simple way to provide Internet applications. The public cloud offers further advantages: If the application is under high load, it can scale automatically. Scalability is virtually unlimited because the application's public cloud environment can provide a great number of resources.

## PaaS in Your Own Data Center

The situation is somewhat different when the applications run in your own data center. The use of PaaS is still very easy for the developers, but the PaaS must be installed. This can be a very complicated process, which outweighs the advantages somewhat. However, the PaaS needs to be installed only once. After that, developers can use the PaaS to install a variety of applications and bring them into production. Operations only needs to ensure that the PaaS functions reliably.

Particularly in the case of operations departments that still have many manual processes and where the provision of resources takes a long time, PaaS can considerably accelerate the rollout of applications without the need for major changes to the organization or processes.

## Macro Architecture

As already shown in [chapter 16](#), microservices platforms have an impact on the macro architecture. While a system such as Kubernetes (see [chapter 17](#)) can run any kind of Docker container, a PaaS works at the level of applications. A PaaS is therefore more restrictive. For example, all Java applications will probably be standardized to one or a few Java versions and Linux distributions. Programming languages not supported by the PaaS simply cannot be used to implement microservices. Monitoring and deployment are determined by the selection of the PaaS. Therefore, a PaaS creates an even higher standardization of the macro architecture than is the case in a Kubernetes environment.

Modern PaaS can be customized and are quite flexible. In that case, the way the PaaS is customized and which technology options are supported can serve as a way to define the macro architecture. As a result, the macro architecture is not defined by the PaaS supplier but by whoever customizes the PaaS.

## 18.2 Cloud Foundry

[Cloud Foundry](#)<sup>200</sup> serves as PaaS technology for the example in this book, for the following reasons:

- Cloud Foundry is an *open source project* involving a number of companies. Cloud Foundry is managed by a *foundation*, in which Cloud Foundry providers such as Pivotal, SAP, IBM, and Swisscom are organized. This ensures broad support, and in addition, many PaaS are already based on Cloud Foundry.
- Cloud Foundry can be easily installed as a Pivotal Cloud Foundry for Local Development on a *laptop* to set up a local PaaS for developers to test microservices systems.
- Many public cloud providers have an offering based on Cloud Foundry. An overview can be found at <https://www.cloudfoundry.org/how-can-i-try-out-cloud-foundry-2016/>.
- Finally, Cloud Foundry can be installed in your own data center. [Pivotal Cloud Foundry](#)<sup>201</sup> is an option for this.

## Flexibility

Cloud Foundry is a very flexible PaaS.

---

<sup>200</sup><https://www.cloudfoundry.org/>

<sup>201</sup><https://pivotal.io/platform>

- Cloud Foundry supports applications in *different programming languages*. A buildpack must be available for the chosen programming language. The buildpack creates the Docker image from the application, which is then executed by Cloud Foundry. The [list of buildpacks<sup>202</sup>](#) shows which buildpacks can be downloaded from the Internet.
- In a Cloud Foundry system, *modified or self-written buildpacks* can be installed. This enables support for additional programming languages or the adaptation of existing support to your own needs.
- The *configuration of the buildpacks* can, for example, change memory settings or make other adjustments. Thus, an existing or self-written buildpack can be adapted to the needs of the microservice.
- Deploying [Docker containers<sup>203</sup>](#) in a Cloud Foundry environment is also possible. However, in this case it is important to pay attention to the special features of Docker under Cloud Foundry. Ultimately, this makes it possible to run virtually any software with Cloud Foundry.

## 18.3 The Example with Cloud Foundry

The microservices in this example are identical to the examples from the previous chapters (see [section 14.1](#)).

- The *catalog* microservice administrates the informationen concerning the goods.
- The *customer* microservice stores the customer data.
- The *order* microservice can receive new orders. It uses the catalog and customer microservice via REST.
- In addition, there is the *Hystrix dashboard*, a Java application for visualizing the monitoring of the Hystrix circuit breaker.
- Finally, a web page *microservices* contains links to the microservices and thus facilitates the entry into the system.

[Figure 18-1](#) shows an overview of the microservices.

---

<sup>202</sup><https://docs.cloudfoundry.org/buildpacks/>

<sup>203</sup><https://docs.cloudfoundry.org/adminguide/docker.html>

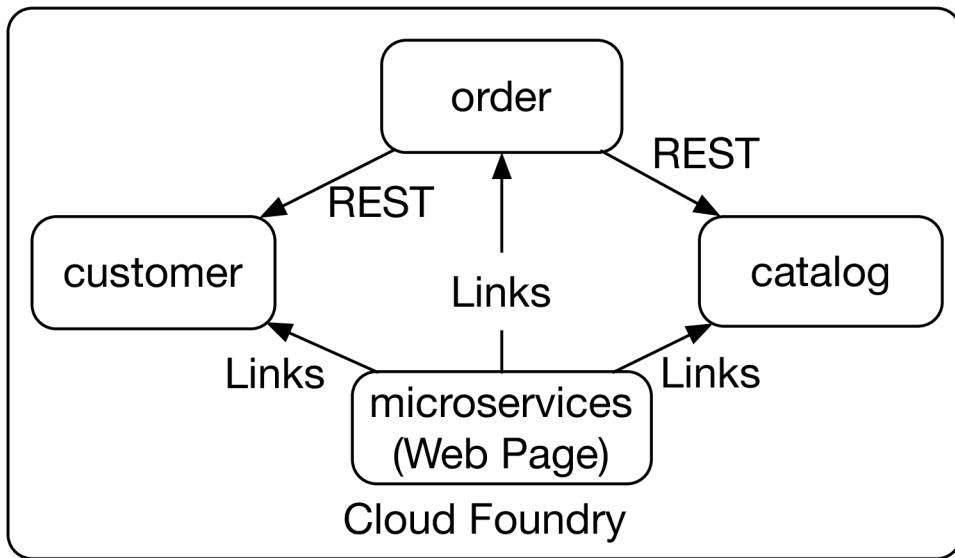


Fig. 18-1: The Microservices System in Cloud Foundry

## Starting Cloud Foundry

Section 0.4 describes what software has to be installed for starting the example.

A detailed description how the example can be built and started is provided at <https://github.com/ewolff/microservice-cloudfoundry/blob/master/HOW-TO-RUN.md>.

The Cloud Foundry example is available at <https://github.com/ewolff/microservice-cloudfoundry>. Download the code with `git clone https://github.com/ewolff/microservice-cloudfoundry.git`.

To start the system, the application must first be compiled with Maven. To do so, you have to execute `./mvnw clean package` (macOS, Linux) or `mvnw.cmd clean package` (Windows) in the sub directory `microservice-cloudfoundry-demo`. See [appendix B](#) for more details on Maven and how to troubleshoot the build.

The example should be started on a local Cloud Foundry installation. The required installation is described at <https://pivotal.io/pcf-dev>. Upon the start of the Cloud Foundry environment, the Paas should be assigned enough memory, for example, with `cf dev start -m 8086`. After the login with `cf login -a api.local.pcfdev.io --skip-ssl-validation`, the environment should be usable.

## Deploying the Microservices

Now deploy the microservices. Execute `cf push` in the sub directory `microservice-cloudfoundry-demo`. This command evaluates the file `manifest.yml`, with which the microservices for Cloud Foundry are configured. `cf push catalog` deploys a single application, such as `catalog`.

```

1  -----
2  memory: 750M
3  env:
4    JBP_CONFIG_OPEN_JDK_JRE: >
5      [memory_calculator:
6        {memory_heuristics:
7          {metaspace: 128}}]
8  applications:
9    - name: catalog
10   path: .../microservice-cloudfoundry-demo-catalog-0.0.1-SNAPSHOT.jar
11    - name: customer
12      path: .../microservice-cloudfoundry-demo-customer-0.0.1-SNAPSHOT.jar
13    - name: hystrix-dashboard
14      path: .../microservice-cloudfoundry-demo-hystrix-dashboard-0.0.1-SNAPSHOT.jar
15    - name: order
16      path: .../microservice-cloudfoundry-demo-order-0.0.1-SNAPSHOT.jar
17    - name: microservices
18      memory: 128M
19      path: microservices

```

The following breaks down part of the configuration can be distinguished.

- Line 2 ensures that each application is provided with 750 MB RAM.
- Line 3-7 changes the memory distribution so that enough memory for the Java bytecode is available in the meta space of the JVM.
- Lines 8-16 configure individual microservices and specify the JAR files to be deployed. For each of the microservices, the common settings in lines 2-7 apply. The paths to the JARs are abbreviated to increase the clarity of the listing.
- Finally, lines 17-19 deploy the application `microservices`, which displays a static HTML page with links to the microservices. The directory `microservices` stores an HTML file, `index.html`, as well as an empty file, `Staticfile`, which marks the content of the directory as static web application.

Cloud Foundry uses the Java buildpack to create Docker containers, which are then started. At <http://microservices.local.pcfdev.io/> the static web page is provided that allows the user to use the individual microservices.

As you can see, the configuration to run the microservices on Cloud Foundry is very simple.

## DNS for Routing or Service Discovery

The microservices themselves have no code dependencies to Cloud Foundry. DNS is used for service discovery. The order microservice calls the catalog and customer microservices. To do

this, it uses the host names `catalog.local.pcfdev.io` and `customer.local.pcfdev.io`, which are derived from the names of the services. The `local.pcfdev.io` domain is the default, but can be customized in the Cloud Foundry configuration. `catalog.local.pcfdev.io`, `order.local.pcfdev.io`, and `customer.local.pcfdev.io` are also the hostnames used in the web browser for the links in the HTML UI of the microservices. Behind this is a routing concept in Cloud Foundry, which makes the microservices accessible from outside and implements load balancing.

With `cf logs`, it is possible to have a look at the microservices logs. `cf events` returns the last entries. At <https://local.pcfdev.io/>, you can find a dashboard showing basic information about the microservices and an overview of the logs (see figure 18-2).

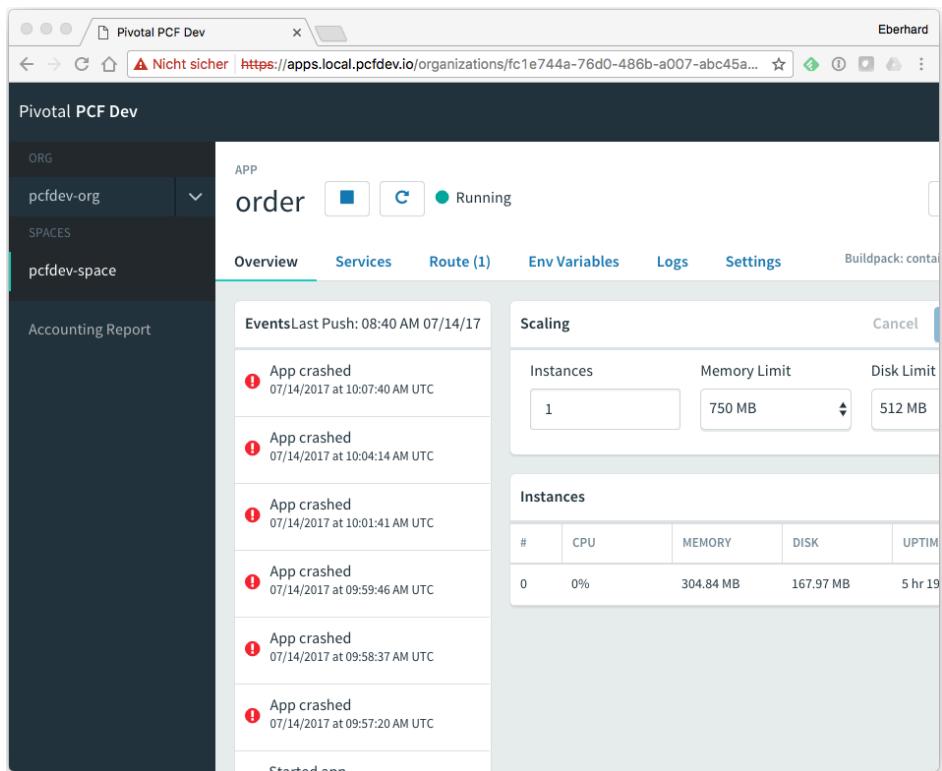


Fig. 18-2: Cloud Foundry Dashboard

With `cf ssh catalog`, the user can log into the Docker container in which the microservice `catalog` runs. This allows the user to examine the environment more closely.

## Using Databases and Other Services

It is possible to provide the microservices with additional *services* such as databases. `cf marketplace` shows the services in the marketplace. These are all services available in the Cloud Foundry installation. From these services, instances can then be created and made available to the applications.

## Example for a Service from the Marketplace

p-mysql is the MySQL service that the local Cloud Foundry installation provides and can be used to run the examples. With `cf marketplace -s p-mysql` you can obtain an overview of the different offerings for the service p-mysql. `cf cs p-mysql 512mb my-mysql` generates a service instance named my-mysql with the configuration 512mb. The command `cf bind-service` can make the service available to an application.

With `cf ds my-mysql`, the service can be deleted again.

## Using Services in Applications

The application must be configured with the information for accessing the service. For this, Cloud Foundry uses environment variables that contain server address, user account, and password. The application must read this information. The respective programming languages provide different possibilities for this. The [buildpack documentation<sup>204</sup>](#) contains more information.

A configuration using environment variables can also be used for settings provided during the installation of the microservice. In the `manifest.yml`, you can set variables that deployed applications can read.

## Services for Asynchronous Communication

Some services add asynchronous communication to Cloud Foundry. For example, the local Cloud Foundry installation offers RabbitMQ and Redis as services. These are both technologies that can send messages asynchronously between microservices. Other Cloud Foundry offerings can provide additional MOMs as services.

## 18.4 Recipe Variations

This chapter refers to the PaaS concept. Cloud Foundry is not the only available PaaS.

- [OpenShift<sup>205</sup>](#) supplements Kubernetes with support for different programming languages to thereby automate the generation of the Docker containers.
- [Amazon Elastic Beanstalk<sup>206</sup>](#) is only available in the Amazon Cloud. It can install applications in virtual machines and scale these virtual machines. Thus, Elastic Beanstalk represents a simplification compared to the IaaS approach. However, because Beanstalk is based on IaaS and some additional features, it rests on a very stable foundation. In Beanstalk applications, additional services from the Amazon offer can be used. These include, for example, databases, but also MOMs. Thus, Elastic Beanstalk benefits from the numerous components available in the Amazon Cloud.

<sup>204</sup><https://docs.run.pivotal.io/buildpacks/>

<sup>205</sup><https://www.openshift.com/>

<sup>206</sup><https://aws.amazon.com/elasticbeanstalk/>

- Heroku<sup>207</sup> is available only in the public cloud. Similar to Cloud Foundry, it has buildpacks for supporting different programming languages and a marketplace for additional services.

## 18.5 Experiments

- Supplement the Cloud Foundry system with an additional microservice.
  - As an example, you can take a microservice that a call center agent can use for making notes about a conversation. The call center agent should be able to select the customer.
  - Calling the customer microservice must use the host name `customer.local.pcfdev.io`.
  - Of course, you can copy and modify one of the existing microservices.
  - Enter the microservice in the file `manifest.yml`.
  - The deployment of a Java application can be derived quite easily from the existing `manifest.mf`. For other languages, the documentation of the buildpacks<sup>208</sup> might be helpful.
- Get acquainted with the possibilities of running Cloud Foundry. Start the example and have a look at the logs with `cf logs`. Log into the Docker container of an application with `cf ssh` and see the latest events of a microservice with `cf events`.

It is also possible to use the Cloud Foundry infrastructure in other areas of the application. However, these experiments are harder to do.

- Replace the integrated database in the microservices with MySQL. For information on how to start MySQL with Cloud Foundry, see section 18.3. However, the code and configuration of the microservices must be changed in order for the applications to use MySQL. There is a Guide<sup>209</sup> for that.
- Cloud Foundry also provides RabbitMQ<sup>210</sup> in the marketplace. This MOM can be used for asynchronous communication between microservices. A guide<sup>211</sup> shows how to use RabbitMQ with Spring. So you can port the example from chapter 11 to RabbitMQ and then run it with a RabbitMQ instance created by Cloud Foundry.
- User-provided service instances<sup>212</sup> are an alternative. With this approach, a microservice can obtain information about the Kafka instance with Cloud Foundry mechanisms. The Kafka instance is not running under the control of Cloud Foundry. Change the Kafka example from chapter 11 so that it uses a user-provided service Kafka instance. This involves the configuration of the port and host for the Kafka server.
- Change one of the microservices and use Blue/Green Deployment<sup>213</sup> to deploy the change so that the microservice does not fail during the deployment. The Blue/Green Deployment first creates a new environment and then switches to the new version so that no downtime occurs.

<sup>207</sup><https://www.heroku.com/>

<sup>208</sup><https://docs.cloudfoundry.org/buildpacks/>

<sup>209</sup><https://spring.io/guides/gs/accessing-data-mysql/>

<sup>210</sup><https://www.rabbitmq.com/>

<sup>211</sup><https://spring.io/guides/gs/messaging-rabbitmq/>

<sup>212</sup><https://docs.cloudfoundry.org/devguide/services/user-provided.html>

<sup>213</sup><https://docs.cloudfoundry.org/devguide/deploy-apps/blue-green.html>

## 18.6 Serverless

PaaS deploy applications. Serverless goes even further and enables the deployment of individual functions. Thereby, Serverless allows even smaller deployments than with PaaS. A REST service can be divided into a variety of functions: one per HTTP method and resource.

The advantages of Serverless are similar to those of PaaS; a high degree of abstraction and thus relatively simple deployment. In addition, Serverless functions are activated only when a request is made so that no costs are incurred if no requests are processed. They can also scale very flexibly.

Serverless technologies include [AWS Lambda<sup>214</sup>](#), [Google Cloud Functions<sup>215</sup>](#), [Azure Functions<sup>216</sup>](#) and [Apache OpenWhisk<sup>217</sup>](#). OpenWhisk allows you to install a Serverless environment in your own data center.

As with a PaaS, Serverless also includes support for operation. Metrics and log management are supplied by the cloud provider.

### REST with AWS Lambda and the API Gateway

With AWS Lambda, you can implement REST services. The API gateway in the AWS cloud can call Lambda functions. A separate function can be implemented for each HTTP operation. Instead of a single PaaS application, there are many Lambda functions.

A technology such as [Amazon SAM<sup>218</sup>](#) can make the large number of Lambda functions quite easy to use. So even deploying a lot of functions takes hardly more effort than implementing the REST methods in a class. Often, Lambda functions can also significantly reduce the cost of running a solution.

### Glue Code

In another area, Lambda functions are also very helpful. A Lambda function can be called in response to an event in the Amazon Cloud. [S3<sup>219</sup>](#) (Simple Storage Service) offers storage for large files in the Amazon Cloud. When a new file is uploaded, a Lambda function can convert it to another format. However, these are not real microservices, but rather glue code to complement functionalities in Amazon services.

## 18.7 Conclusion

For synchronous microservices, Cloud Foundry's solutions are very similar to those of Kubernetes.

<sup>214</sup><https://aws.amazon.com/lambda/>

<sup>215</sup><https://cloud.google.com/functions/>

<sup>216</sup><https://azure.microsoft.com/services/functions/>

<sup>217</sup><https://developer.ibm.com/code/open/apache-openwhisk/>

<sup>218</sup><http://docs.aws.amazon.com/lambda/latest/dg/deploying-lambda-apps.html>

<sup>219</sup><https://aws.amazon.com/s3/>

- *Service discovery* also works via DNS. It is therefore transparent for client and server. In addition, no code needs to be written for the registration.
- *Load balancing* is also transparently implemented by Cloud Foundry. If several instances of a microservice are deployed, the requests are automatically distributed to these instances.
- For the *routing* of external requests, Cloud Foundry relies on DNS and a distribution of the requests to the various microservice instances.
- For *resilience*, the Cloud Foundry example uses the Hystrix library. Cloud Foundry itself does not offer a solution in this area.

In addition, a PaaS provides a standardized runtime environment for microservices and can therefore be an important antidote to the high level of operational complexity that microservices bring. Thus, a PaaS enforces a standardization often desirable in the context of macro architecture (see [chapter 2](#)).

Compared to Docker ([chapter 5](#)) or Kubernetes ([chapter 17](#)), a PaaS provides less flexibility. This can also be a strength due to the standardization that goes hand in hand with reduced flexibility. Modern PaaS also offer the possibility of adapting the environment with concepts such as buildpacks or even of running Docker containers with arbitrary applications.

## Benefits

- PaaS solve typical problems of microservices (load balancing, routing, service discovery).
- Cloud Foundry introduces no code dependencies.
- PaaS also cover operation and deployment.
- PaaS enforce standardization and thereby the definition of a macro architecture.
- Developers only have to deliver applications. Docker is hidden.

## Challenges

- Cloud Foundry requires a complete switch of the operation approach.
- Cloud Foundry is very powerful, but thus also very complex.
- Cloud Foundry provides a high degree of flexibility; however, compared to Docker containers this flexibility still has its limits.

# 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 system 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](#).

# 19 Concept: Operation

This chapter deals with the operation of microservices and covers the following points:

- Operations can help to evaluate the business results of changes to the software.
- Better operation can accelerate response times to problems, thereby improving application availability and quality.
- Operation influences micro and macro architecture.

## 19.1 Why Operation Is Important

Microservices change the importance of operation for various reasons (see figure 19-1).

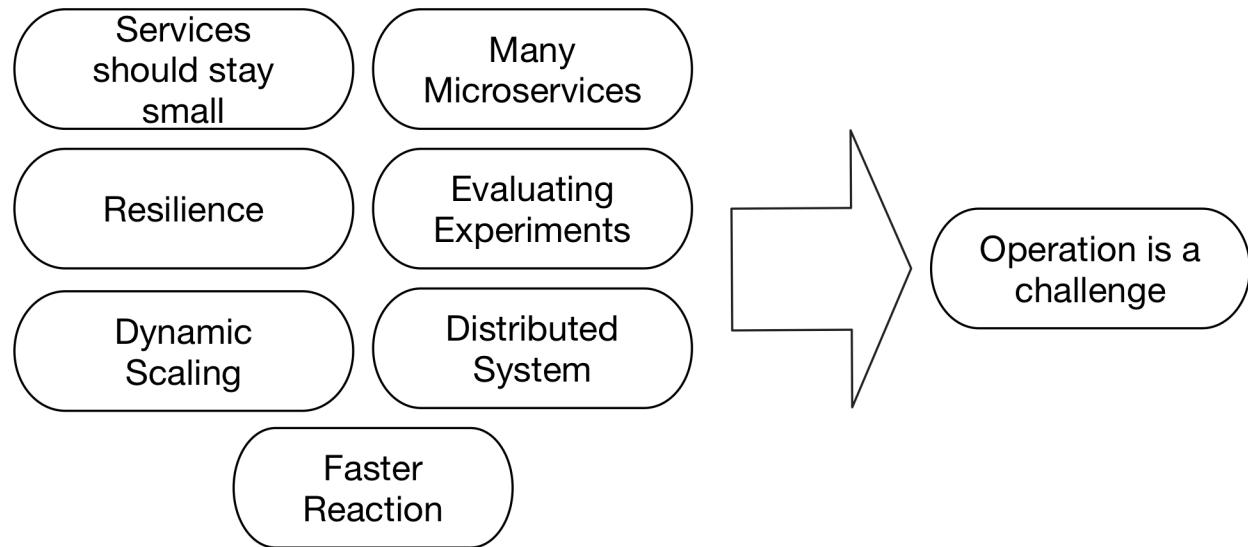


Fig. 19-1: Factors Influencing the Operation of Microservices

### Many Microservices

In a microservices architecture, a system is not a single deployment monolith. Instead, each module of the system is a separate microservice that must be individually deployed, operated, and monitored. So, many more applications need to be deployed and monitored.

If a project runs for a longer period of time, more and more code is written. This can lead to the microservices growing in size. This is problematic because the advantages of microservices can be lost. Thus, new microservices should be built over time so that the size of the microservices

remains constant and only the number of microservices grows. But with the number of microservices increasing, the challenges of operation become greater, too.

Of course, it is not acceptable for the costs of operation to increase by an order of magnitude. Measures must therefore be taken to keep expenditure within reasonable bounds. Standardization and automation are suitable methods for this purpose.

Even if new microservices are created in the system, no manual effort should be required to integrate the microservices into the environment. This not only simplifies operation, but also facilitates the creation of new microservices in order to keep the size of the microservices constant. For this purpose, a template for a new microservice that already contains all necessary technologies for efficient operation can be helpful.

## Evaluating Experiments

Every change to a microservices system should aim to achieve a business objective. For example, the optimization of the user registration may be aimed at increasing the number of customers who register.

The change is comparable to a scientific experiment. In a scientific experiment, a hypothesis is made, then an experiment is carried out, and the result is measured to check the hypothesis. The same procedure is used here. The hypothesis is: "The new registration will increase the number of active users." Then the change must be made and the results measured.

Collecting and measuring data from applications is a classic task of operations. Usually, operations confines itself to monitoring system metrics. In principle, however, the results of the experiments could also be evaluated with the approaches operations uses.

## Distributed System

Microservices turn the system into a distributed system. Instead of local method calls, microservices communicate via the network. This increases the number of possible error sources. The network and server hardware may fail. This increases the demands on operations, which is responsible for the reliability of all components.

Troubleshooting is also a challenge in a distributed system. An error in a microservice can be caused by one of the microservices it calls. Therefore, there a way must exist to track calls between microservices in order to identify the source of the error. This tracing is only required in a microservices environment and poses an additional challenge during operation. [Chapter 22](#) shows Zipkin as a technological solution for this problem.

In order to analyse the microservices, further measures are necessary. For a deployment monolith, it is enough to examine the processes and resource consumption on the server using operating system tools. Log files can also be a good source of information. In a distributed system, the number of servers is too large to be successful with this approach. You cannot log on to every server and look for the cause of the error. So there must be a centralized infrastructure that collects information from all microservices in one central location. [Chapter 21](#) describes the Elastic Stack for managing log files, and [chapter 22](#) deals with Prometheus for collecting metrics about all microservices.

## Ensuring Performance

Each system must offer a certain performance. Capacity tests are used to avoid performance problems. To do this, the tests have to use a realistic amount of data, simulate user behavior, and run on production-like hardware, which is hardly feasible:

- The *data volumes* from production often are too large to be processed in a test environment.
- It can also be difficult to simulate *user behavior*. In the end, it is unclear how long users wait and think until they perform the next action or how often they choose a certain course of a process.
- Setting up the production environment is often already difficult. This applies to sizing as well as to the integration of third-party systems. Building additional *environments for tests* that are similarly powerful as the production environment is often impossible.
- For *new features*, it is impossible to predict the behavior of the users and the success of the feature, and thus the load. Capacity tests alone are not an adequate measure to secure the performance of the application in production, because the test scenarios are based on pure assumptions and therefore cannot produce realistic results.

## Supplements to Tests

In addition to capacity tests, other measures may be useful.

- Effective *monitoring* can detect a problem with performance in production at an early stage. This reduces the time it takes to respond and resolve the problem. Because microservices are individually scalable, more instances of a microservice can be started to cope with the load. Ideally, the problem is then solved without a user noticing anything about it.
- If the problem is more complex, a *fix must be delivered*. For this, a fast and highly automated deployment is useful, which microservices typically bring with them.

Similar considerations apply not only to performance problems, but also to problems with the implemented logic, for example. Monitoring and fast deployment can also speed up the response to such problems. For example, if registration numbers or sales suddenly drop, this can be an indication of a problem with the implemented logic. Maybe nobody is actually able to register any more because of an error. Monitoring is therefore also useful as a supplement to tests of the logic.

## Dynamic Scaling

One advantage of microservices is that they can be scaled independently. Additional or fewer instances of each microservice can run to handle the current load. This requires the use of technologies that allow new instances to be started. Kubernetes (see [chapter 17](#)) can do this, for example, and uses resources from an entire cluster for the operation of the microservices.

But even if such a technology is used, only a limited number of servers are available in the cluster. Although the infrastructure becomes more flexible, capacity planning is still necessary to ensure scalability.

Therefore, the necessary prerequisites must be created to enable dynamic scaling of the microservices system.

## 19.2 Approaches for the Operation of Microservices

The operation of microservices consists of (see figure 19-2):

- The *deployment*: This aspect is solved by technologies such as Docker (see [chapter 5](#)), microservices platforms such as Kubernetes (see [chapter 17](#)), or a PaaS such as Cloud Foundry (see [chapter 18](#)).
- *Monitoring* is a focus of [chapter 20](#).
- The *analysis of log files* is described in [chapter 21](#).
- Finally, [chapter 22](#) shows how *tracing* can trace the calls between microservices.

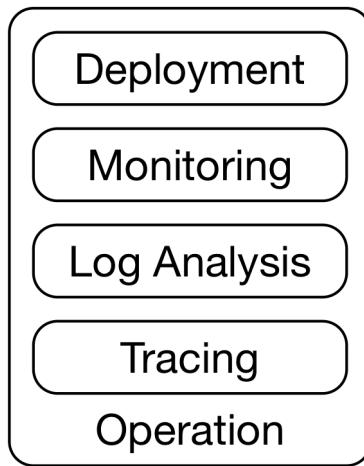


Fig. 19-2: Challenges in the Operation of Microservices

### Independent Deployment

Microservices also bring advantages for the operation.

A microservice is much smaller than a deployment monolith and therefore easier to deploy. Even if the microservice fails for some time during deployment, this should not have any dramatic consequences thanks of resilience, because the other microservices continue to run.

This requires that the microservices be deployed independently. If a feature requires changes to multiple microservices, independent deployment must still be ensured. If the client and server of

an interface have to be deployed simultaneously, deployment is no longer independent. Instead, the microservices can provide the old and the new interface in parallel to decouple the deployment of the interface's client and server. This way, a new server can be deployed. The client can still use the old interface, so the client can be deployed at a later point in time. Eventually, the old interface can be removed, which requires yet another deployment. The deployment of the individual microservice is easier, but more deployments are needed to update both client and server.

If the deployments are not decoupled, several microservices or even all microservices must be deployed in a coordinated manner. This is difficult because all deployments have to run smoothly, and in case of a failure, all deployments have to be rolled back.

Independent deployment is one of the most important advantages of microservices because it leads to a high degree of independence. Therefore, you should always strive for independent deployment.

### Step-by-step Introduction of Operation for Microservices

As discussed, the operation of microservices is of great importance. In addition to operation, many other fundamental changes are necessary for microservices. New technologies, frameworks, and architectures must be implemented. In addition, organizational changes may be necessary. Independent and self-organized teams should develop independent microservices to derive maximum benefit from the microservices architecture. With so many changes, the risk of problems increases.

When the first microservice is supposed to go into production, it is not yet necessary to have the level of automation and sophistication that would be required for running a great number of microservices. It is therefore conceivable to gradually build up the necessary operation technologies. However, this implies that if the setup of the environments cannot keep pace with the number of microservices, the microservices system becomes unreliable and costly to operate. At some point, manual processes are by far too expensive to support the huge number of microservices.

## 19.3 Effects of the Discussed Technologies

The technologies discussed so far have an impact on operation.

- *Docker* (see [chapter 5](#)) allows a very easy installation of software. The complete environment, including the Linux distribution, is included in the Docker image. The image just needs to be pulled from a repository to install a microservice. At the same time, Docker is very efficient, so not too much additional hardware is needed.
- *Links and client-side integration* ([chapter 8](#)) means that only several web applications need to be run. Most companies already run web applications, so that only more environments of the same type need to be run.
- With *UI integration on the server* – for example, with ESI (Edge Side Includes; see [chapter 9](#)) – an additional server must be operated for the integration. For ESI, this might be a Varnish cache. With Server Side Includes, a web server is enough, which may already be used in the system and only needs to be configured appropriately.

- *Kafka* ([chapter 11](#)) or other Message-oriented Middlewares (MOMs) introduce powerful, but also complex software for asynchronous communication. Accordingly, operation is also complex. A failure or problem of the MOM affects the entire microservices system. Therefore, this alternative is a challenge for operations.
- In contrast, asynchronous communication with *Atom* ([chapter 12](#)) uses HTTP and REST. This means that no other infrastructure is required compared to a synchronous REST system. So operation is identical to that of a REST or web application.
- The *Netflix stack* ([chapter 14](#)) implements all necessary infrastructure with Java. Spring Cloud enables uniform configuration of microservices and infrastructure services from the Netflix stack. Especially with a Java microservices system, the operation of the entire system can be simplified in this way. On the other hand, the Netflix stack uses its own custom solution for routing rather than a web server, with which operations has already gained experience in most cases.
- The *Consul stack* (see [chapter 15](#)) introduces on the one hand with Consul, a Go application, which might cause more effort for operation compared to a system that just uses Java. On the other hand, Consul is so flexible that a configuration of web servers like Apache httpd is possible without any problems. The example uses the Apache httpd for routing, but Apache httpd could also be used for load balancing between microservices. This makes it possible to use technologies that companies usually already know, which minimizes the risk.
- *Kubernetes* or a *PaaS* such as *Cloud Foundry* offer a complete solution for the operation of microservices. However, they are also complex and take over functions from existing systems, such as virtualization software. It is therefore a major step to put them into production; but then they offer many advantages.

## 19.4 Conclusion

Although the subject of “operation” is located at the end of the book, it is still an important subject. Without a good operation, the many microservices cannot be brought into production. The reliability and performance of a microservices solution also depends heavily on the operation, which is why it is so important.

# 20 Recipe: Monitoring with Prometheus

This chapter focuses on the monitoring of microservices. Essential topics are:

- This chapter describes the basics of monitoring: Which metrics need to be monitored and what they are useful for.
- Microservices monitoring places new requirements on monitoring tools.
- Prometheus has advantageous features for monitoring microservices environments.
- Other tools are also suitable for monitoring microservices.

## 20.1 Basics

Monitoring visualizes and analyses metrics that describe the state of the system (see [figure 20-1](#)).

There are different kinds of metrics.

- *Counters* can be increased and thereby can count events. For example, the number of registered users or the number of processed tasks can be counted.
- *Gauges* display a value which changes over time. For example, a gauge can measure the memory usage or the network traffic.
- A *histogram* measures the number of certain events. For example, a histogram can count the number of processed requests, and additionally distinguish the number of successfully processed requests from the requests that failed due to different kinds of errors.

## Processing Metrics

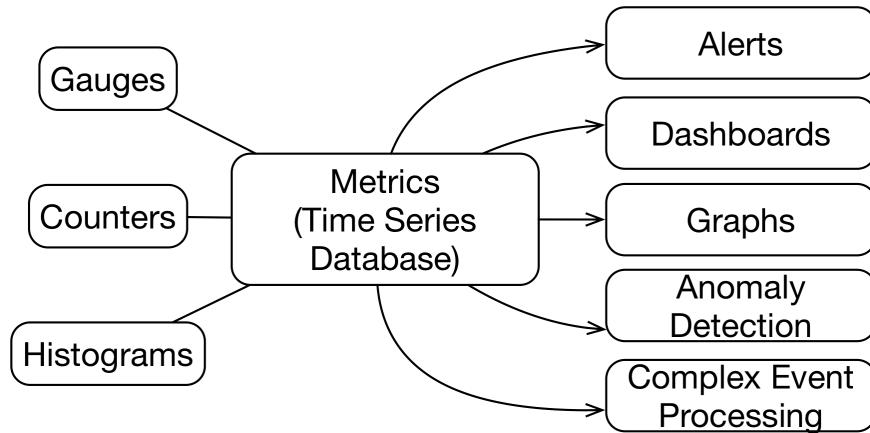


Fig. 20-1: Metrics: Kinds and Processing

Often, metrics are saved in time series databases. Time series databases are specialized in saving data indexed based on time.

The metrics can be processed in different ways.

- A metric that attains a certain value triggers an *alert* to inform an operations team member who then analyzes and solves the problem.
- With the help of a *graph*, the metrics can be visualized, making it easier to quickly recognize changes. This allows a human to analyze the system.
- A *dashboard* displays multiple metrics in an overview. This very quickly conveys an impression of the state of the system. For example, a screen close to the coffee machine can display the dashboard so that the employees are informed about the current state of the system while getting a coffee.
- *Complex event processing (CEP)* processes metrics and implements analyses. For example, the metrics can be condensed to average values over a certain interval.
- *Anomaly detection* automatically recognizes whether metrics lie outside of the expected range, thereby indicating problems early on.

## Different Metrics for Different Stakeholders

Different stakeholders can be interested in different metrics. For example, system metrics focus on the level of operating system or hardware. Network throughput or hard disk throughput fall into this category. Operations experts are usually interested in such metrics.

Application metrics describe, for example, the number of different kinds of requests or of method calls to a certain method. These metrics help developers to do their work.

Finally, business metrics such as sales or the number of registered users are often interesting for the product owners or users.

Technically, a single system can save all these metrics. However, the stakeholders often want to use different tools. Specialized tools generally offer better analysis options or are already familiar to the stakeholders. For example, for web analytics tools such as [Google Analytics<sup>220</sup>](#) can be used. So when the stakeholders insist on using different tools, it is conceivable to store the different metrics in different systems.

The metrics can be used to draw different conclusions. Product owners can evaluate the business success, operations can plan capacities, and developers can analyze errors.

## 20.2 Metrics for Microservices

Regarding the processing of metrics, microservices differ from classical systems in some important points.

### More Services, More Metrics

There are substantially more instances of microservices than would be the case with a deployment monolith. Therefore, there are also a lot more metrics. Consequently, the system for processing metrics has to scale appropriately. In addition, all metrics of all microservices should be stored in one system to keep the overview.

### Services Instead of Instances

For most microservices, several instances are running. This is the only way to ensure fail-safety and to handle the load. For metrics, the individual instances are not interesting, but rather the results of all instances of a microservice and therefore the quality of service it provides to the user. Thus, average metrics of all instances of a specific microservice are important, but not the metrics for each specific instance.

### Away from System Metrics

In addition, the processing of metrics has to change. In a classical system, the failure of a server normally causes an alert or even leads to the operations expert being called up from sleep. In a microservices system, this might not be necessary. Each microservice can be scaled independently so that usually multiple instances of each microservice are running. When a server fails, new instances can be automatically started on other servers.

Due to their robustness and resilience microservices can even compensate for the failure of other microservices. Therefore, the failure of a server or of an instance of a microservice is less problematic than in a traditional system.

The system can even compensate an overload by starting new instances on new servers. This allows it to keep pace with the higher load without manual intervention.

---

<sup>220</sup><https://analytics.google.com/>

## Towards Application Metrics

Overall, system metrics are not as important for microservices as for traditional systems. However, it is important to analyze the system from a user's point of view. For example, requests may take too long, which might affect user behavior. And this might happen despite the fact that the system metrics are all normal. For example, there may be an error in the software or data may lead to the problem. If the additional waiting times have a negative effect on revenue, this can be a reason for immediately triggering an alert when waiting times are increased, even though the system metrics are normal. Thus, the priority shifts from system metrics to application and business metrics. Service-level agreements (SLAs) can be a good source of metrics. SLAs define what the customer expects from the system in terms of response times, for example.

## Alerts Based on Domain Logic

Whether application metrics show unusual values and whether it is necessary to react to them is a decision that must be made by domain experts. Typically, developers have a good insight into the implemented domain functionality so that alerts can be defined by operations in close collaboration with the developers. However, operations alone cannot often do this.

## 20.3 Metrics with Prometheus

Prometheus<sup>221</sup> is a relatively new tool for monitoring. It has a number of interesting features.

- Prometheus supports a *multidimensional data model*. For example, the duration of HTTP requests per URL and per HTTP methods (for example, GET, POST, DELETE) can be stored. For each URL, there is a value for each of the four HTTP methods. Prometheus can sum up the duration of all HTTP methods for a URL, or the duration of an HTTP method for all URLs. This allows more flexibility in the analysis. The evaluation is carried out using a query language. The results not only are calculated, but also displayed graphically.
- Many monitoring solutions require the monitored systems to write the metrics to the monitoring system (push model). Prometheus has a *pull model* and collects data from an HTTP endpoint in the monitored systems at specified intervals. This corresponds to the procedure for asynchronous communication via HTTP depicted in chapter 12. For communicating with applications that use a push model, teams can use a tool such as [Push Gateway](#)<sup>222</sup>.
- An [alertmanager](#)<sup>223</sup> can trigger alerts based on queries and can send the alert messages as e-mail or per [PagerDuty](#)<sup>224</sup>.
- With *dashboard*, the user can execute queries and can have the results displayed graphically. Prometheus can also integrate into other visualization tools – for example, into [Grafana](#)<sup>225</sup>.

<sup>221</sup><http://prometheus.io/>

<sup>222</sup><https://github.com/prometheus/pushgateway>

<sup>223</sup><https://prometheus.io/docs/alerting/alertmanager/>

<sup>224</sup><https://www.pagerduty.com/>

<sup>225</sup><https://prometheus.io/docs/visualization/grafana/>

At its core, Prometheus is a multidimensional time series database and also offers support for alerts and graphical evaluations. With the multidimensional database, it is possible to capture the metrics of all instances of all microservices and then sum them up for a certain type of microservice.

## Examples for Multidimensional Metrics

Prometheus can count the number of HTTP requests. Based on this, Prometheus can use the built-in [functions<sup>226</sup>](#) to calculate and process the number of HTTP requests per second. This allows Prometheus to calculate, for example, the average for each handler ([figure 20-2](#)). The formula `avg(rate(http_requests_total[5m])) by (handler)` is used for this purpose.

---

<sup>226</sup><https://prometheus.io/docs/querying/functions/>

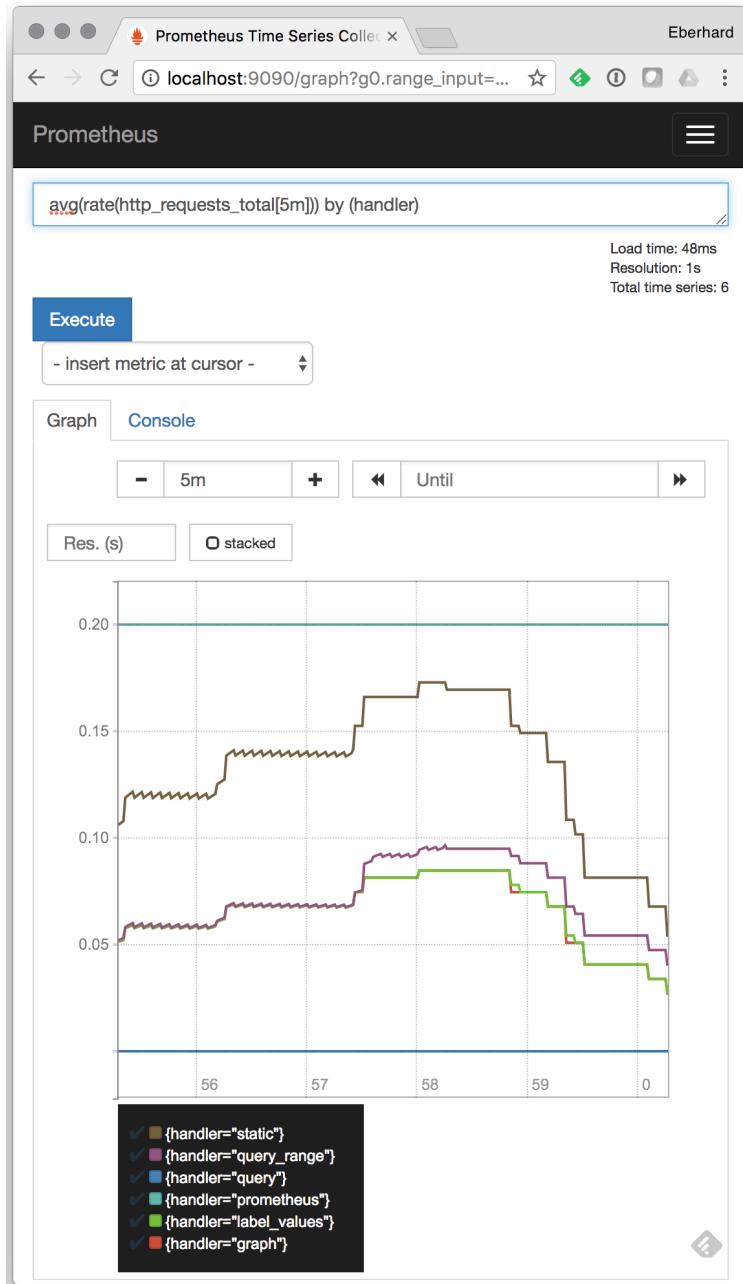


Fig. 20-2: Prometheus Dashboard with Calculated Metrics

Prometheus can also calculate the average based on HTTP codes. All that needs to be done is to change the formula to `avg(rate(http_requests_total[5m])) by (code)`. Figure 20-3 shows the result.

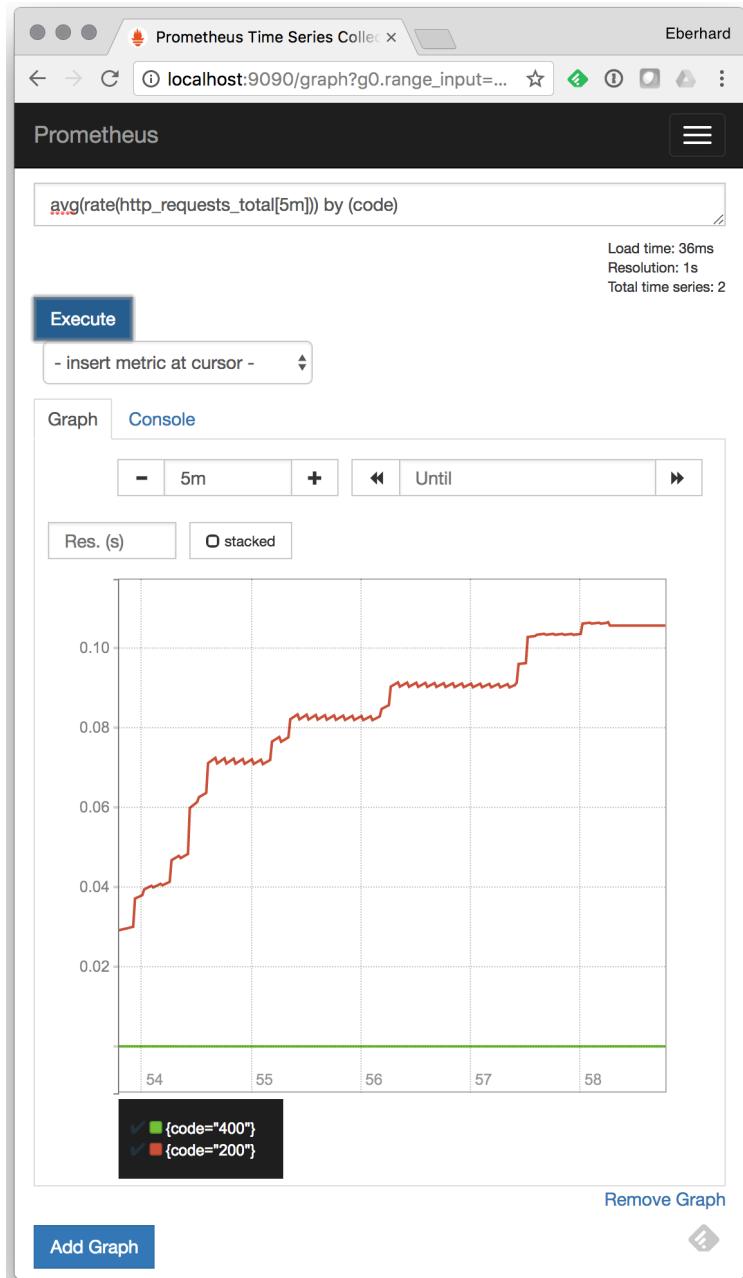


Fig. 20-3: Prometheus Dashboard with Other Calculated Metrics

This enables Prometheus to extract information from the metrics of all microservices that are of interest to users. A single very slow instance of a microservice is less problematic than a general problem affecting all instances of a microservice. Prometheus' evaluations make it possible to draw exactly such conclusions.

## 20.4 Example with Prometheus

The example for microservices with Consul (see [chapter 15](#)) can be monitored with Prometheus. This only requires a Docker Compose configuration with some additional containers.

### Starting the Environment

[Section 0.4](#) describes which software has to be installed for starting the example.

The example project can be found at <https://github.com/ewolff/microservice-consul>. First, the code has to be downloaded with `git clone https://github.com/ewolff/microservice-consul.git`.

<https://github.com/ewolff/microservice-consul/blob/master/HOW-TO-RUN.md> provides instructions that describe in detail how to start the example and how to install the necessary software. The approach for monitoring with Prometheus is explained in more detail at <https://github.com/ewolff/microservice-consul/blob/master/HOW-TO-RUN.md#run-the-prometheus-example>.

For starting the system, the application first has to be compiled with `./mvnw clean package` (macOS, Linux) or `mvnw.cmd clean package` (Windows) in the sub directory `microservice-consul-demo`. See [appendix B](#) for more details on Maven and how to troubleshoot the build.

Afterwards create the Docker containers with `docker-compose -f docker-compose-prometheus.yml build` in the sub directory `docker`. See [appendix C](#) for more details on Docker, Docker Compose, and how to troubleshoot them. The normal system configuration of the Consul example has no Prometheus Docker container. So the configuration in the file `docker-compose-prometheus.yml` has to be used. The application can be started with `docker-compose -f docker-compose-prometheus.yml up -d`.

As with the Consul demo, the application is available at port 8080 – for example, at the URL <http://localhost:8080/>. On the Docker host Prometheus can be reached at port 9090. So if the Docker containers are running locally, the Prometheus UI is available at <http://localhost:9090/>. For debugging, <http://localhost:9090/targets> is useful. The state of all jobs Prometheus uses to fetch metrics from the various microservices is displayed at this URL.

### Code in Spring Boot

In the example project, the order microservice has an interface to Prometheus for monitoring data. The other microservices do not offer such an interface.

As [section 5.3](#) explains, Spring Boot offers Sprint Boot Actuator as a way to collect metrics about the microservices. These metrics include, for example, the number and duration of HTTP requests. To support Prometheus, these metrics only need to be provided in such a way that Prometheus can pull them.

This requires the following changes to the project:

- An additional dependency to the Prometheus client library has to be inserted in `pom.xml`.

- Unfortunately, Prometheus cannot use the usual Spring Boot Actuator REST endpoints for metrics. Therefore, the Prometheus servlet has to be integrated. Also, the Spring Boot Actuator metrics have to be provided to Prometheus with a `SpringBootMetricsCollector`. This class belongs to the Prometheus client library. In the class `PrometheusConfiguration`, an instance of this class is created as a Spring Bean to make the metrics available to Prometheus.

## Prometheus Configuration

Offering metrics alone is not enough. Prometheus also has to pick up the metrics from the order microservice. The sub directory `docker/prometheus` in the example project contains a Dockerfile to create a Prometheus installation in a Docker image. It is based on the Prometheus Docker image of the Prometheus development team and only adds a custom configuration. The configuration format is described in more detail in the [Prometheus configuration documentation<sup>227</sup>](#).

## Configuration in the Example

The configuration in the file `prometheus.yaml` looks like this:

```

1 global:
2   scrape_interval: 15s
3
4 scrape_configs:
5   - job_name: 'prometheus'
6     scrape_interval: 5s
7     static_configs:
8       - targets: ['localhost:9090']
9
10  - job_name: 'order'
11    scrape_interval: 10s
12    metrics_path: '/prometheus'
13    static_configs:
14      - targets: ['order:8380']

```

The section `global` (line 1) contains settings which influence the entire Prometheus system. In the example, this is only the default setting `scrape_interval` (line 2) which determines how often the metrics are fetched from the monitored microservices (“scraping”).

The section `scrape_configs` (line 4) defines the jobs. A job is a configuration with which Prometheus fetches metrics from a server for storage and processing.

The job `prometheus` (lines 5-8) serves to monitor Prometheus itself. Prometheus fetches metrics from the server `localhost` at port 9090. That is the port on which Prometheus listens and also provides

---

<sup>227</sup><https://prometheus.io/docs/operating/configuration/>

its metrics. Prometheus scrapes the metrics every 5 seconds and therefore more frequently than the default value of 15 seconds.

The other job is called `order` (lines 10-14) and serves for monitoring the `order` microservice. Prometheus fetches the metrics every 10 seconds from the server `order` at port 8380. In the Docker compose configuration, a Docker Compose link is generated so that the host name `order` is resolved to the `order` microservice. `metrics_path` is also defined for the job. The default is `/metrics`. However, for a Spring Boot application the JSON-formatted Spring Boot Actuator metrics are available at this path. Prometheus requires a custom text-based format made available in a Spring Boot application under the path `/prometheus`.

## 20.5 Recipe Variations

The service mesh Istio uses Prometheus as a monitoring tool. Istio uses proxies to measure the traffic between the microservices. So it can provide information about the communication such as the number of requests, the amount of transferred data and the number of failed requests without any modification to the microservice (see [section 23.4](#)).

### Other Tools

A large number of tools are available in the area of monitoring.

- StatsD<sup>228</sup> can consolidate the collected metrics in order to send less data over the network. The StatsD exporter<sup>229</sup> can export the metrics to Prometheus.
- collectd<sup>230</sup> measures system metrics. With the collectd exporter<sup>231</sup>, Prometheus can use the data.
- There are also alternatives which replace Prometheus – for example, the TICK stack<sup>232</sup>.
  - Telegraf collects data and passes it on.
  - InfluxDB is a time series database.
  - Chronograf offers visualization and analysis.
  - Kapacitor takes care of alerts and anomaly detection.
- Grafana<sup>233</sup> provides graphical analysis for metrics. It can display data stored in Prometheus. Istio uses this approach to provide dashboards for microservices (see [section 23.4](#)).

<sup>228</sup><https://github.com/etsy/statsd>

<sup>229</sup>[https://github.com/prometheus/statsd\\_exporter](https://github.com/prometheus/statsd_exporter)

<sup>230</sup><https://collectd.org/>

<sup>231</sup>[https://github.com/prometheus/collectd\\_exporter](https://github.com/prometheus/collectd_exporter)

<sup>232</sup><https://www.influxdata.com/time-series-platform/>

<sup>233</sup><https://grafana.com/>

## 20.6 Experiments

### Experiments: Choosing Metrics

Which metrics are relevant for a microservice is different for each microservice and also differs from project to project. The monitoring can be adapted to your own requirements as follows.

- Consider a project you know. Which *metrics* are currently being measured? Note: The technical metrics from operations are often easy to find out about because they are processed in the familiar monitoring applications. From a business perspective, tools for web analysis or reports can be used. Ultimately, these are also metrics, even if they are processed in other tools.
- Which *stakeholders* are there? Obvious stakeholders are development and operations. However, the business side is also relevant. On the business side, there can even be many different stakeholders if, for example, several business divisions are interested in the application. Perhaps other groups such as QA are also interested.
- Together with stakeholders, you can find out which *further metrics* are relevant. Microservices allow fast release cycles. In order to better understand what should be changed and brought into production, there must be more and better metrics, particularly business metrics.
- What changes with *microservices*? There will then be more servers, more communication between the microservices over the network, and a clearer division based on domains. What impact does this have on the metrics?
- Do the additional metrics and the larger number of deployable artifacts result in a *data volume* that cannot be processed with the current monitoring technologies?

### Experiments: Extending Prometheus

The Prometheus installation can be extended in different ways.

- Instead of the static configuration, where each microservice must be listed in the Prometheus configuration, service discovery with Consul can be used. New services are then automatically included in monitoring. For this, the following steps are necessary:
  - Delete the link from Docker container `prometheus` to Docker container `order` in the file `docker-compose-prometheus.yml`. Now access will take place via Consul so that a Docker Compose link to the microservice is not necessary anymore.
  - Insert a link from Docker container `prometheus` to Docker container `consul` into the file `docker-compose-prometheus.yml` so that Prometheus can read the information about the services from Consul.
  - Delete the job `order` in the Prometheus configuration `prometheus.yml`.
  - Create a new job `consul` in the Prometheus configuration `prometheus.yml`. The documentation<sup>234</sup> states that instead of `scrape_configs`, an element named `consul_sd_configs` has

<sup>234</sup>[https://github.com/prometheus/docs/blob/master/content/docs/operating/configuration.md#consul\\_sd\\_config](https://github.com/prometheus/docs/blob/master/content/docs/operating/configuration.md#consul_sd_config)

to be created in which the server is then defined. Because of the Docker Compose link, the hostname of the Consul server is `consul`. The port also has to be indicated. Therefore, '`consul:8500`' is correct.

- Build the Docker images with `docker-compose -f docker-compose-prometheus.yml build` and start them anew with `docker-compose -f docker-compose-prometheus.yml up -d`.
- Only the *order* microservice provides metrics for Prometheus at the moment. Thus, as an exercise, the microservices *catalog* and *customer* can also be provided with Prometheus support. To do this, you must:
  - Insert the dependencies with groupId `io.prometheus` in `pom.xml` so that the Prometheus client code is available. `pom.xml` from the *order* microservice should serve as template.
  - Copy the package `com.ewolff.microservice.order.prometheus` into the other project and rename it there – for example, to `com.ewolff.microservice.catalog.prometheus`.
  - If the Consul service discovery from the previous experiment is used, Prometheus will automatically find the new service. Without Consul, a new job must be created in the `scrape_configs` section of `prometheus.yml`. The *order* job can be a good template for this.
  - As previously described, recompile the applications with `./mvnw clean package` (macOS, Linux) or `mvnw.cmd clean package` (Windows) in the directory `microservice-consul-demo`. Then build the Docker containers with `docker-compose -f docker-compose-prometheus.yml build` again, and restart them with `docker-compose -f docker-compose-prometheus.yml up -d`. Now the metrics of another microservice should be shown in the dashboard. You can check it at <http://localhost:9090/targets>.
- Use the [node exporter<sup>235</sup>](#) for displaying the metrics of the host in Prometheus. These are metrics of the system like disk I/O.
- [cAdvisor<sup>236</sup>](#) can be used for reading the metrics of a Docker container. cAdvisor also can provide the metrics to [Prometheus<sup>237</sup>](#).
- Monitor the Docker runtime environment with Prometheus. The [Docker documentation<sup>238</sup>](#) shows how the monitoring can be implemented.
- Install [Grafana<sup>239</sup>](#) as an alternative graphical frontend for Prometheus.
- Install the alertmanager. To do so, analogous to the Prometheus image from directory `microservice-consul/docker/prometheus`, you can create your own alertmanager Docker image which is based on the official [alertmanager Docker image<sup>240</sup>](#) and just adds a configuration. The [documentation of the configuration<sup>241</sup>](#) and [examples for alerts<sup>242</sup>](#) can help with creating your own configuration.

<sup>235</sup>[https://github.com/prometheus/node\\_exporter](https://github.com/prometheus/node_exporter)

<sup>236</sup><https://github.com/google/cadvisor>

<sup>237</sup><https://github.com/google/cadvisor/blob/master/docs/storage/prometheus.md>

<sup>238</sup><https://docs.docker.com/engine/admin/prometheus/>

<sup>239</sup><https://prometheus.io/docs/visualization/grafana/>

<sup>240</sup><https://hub.docker.com/r/prom/alertmanager/tags/>

<sup>241</sup><https://prometheus.io/docs/alerting/configuration/>

<sup>242</sup>[https://prometheus.io/docs/alerting/notification\\_examples/](https://prometheus.io/docs/alerting/notification_examples/)

## 20.7 Conclusion

Monitoring is essential for every type of microservice and is demanding due to the high number of services. Central monitoring for all services often has considerable advantages. Central monitoring should therefore be considered for each microservices system.

Prometheus as a monitoring tool in microservices environments has the advantage that the multidimensional model allows raw metrics to be combined into metrics that show the behavior of all instances of a specific microservice later on, thus enabling metrics from the user's point of view. The user uses not only one instance of a microservice, but the entire system, so such metrics provide a better impression about what is happening in the system and how it matters for users. Other technologies can further support the monitoring of a microservices system or replace Prometheus.

### Advantages

- Prometheus polls data and protects itself against overload.
- Prometheus' multidimensional data can be evaluated in different ways. For example, all instances of a microservice can be summed up.
- Prometheus can be integrated with other solutions.

### Challenges

- Prometheus covers an area for which most organizations already have solutions, making retraining necessary.

# 21 Recipe: Log Analysis with the Elastic Stack

This chapter describes the analysis of log data.

- First, this chapter shows why logs are so widely used and how they facilitate system operation.
- Logs in microservices systems have to respond to other demands than logs in traditional systems. This chapter explains how even large microservices systems can be provided with a suitable system for log analysis.
- As a concrete solution for the analysis of log data, this chapter introduces the Elastic Stack.

Thereby, the reader gets to know how logs can be efficiently and effectively processed in a microservices system.

## 21.1 Basics

Logs differ from metrics. Metrics represent the current state of the system; logs record events. These can comprise errors or business events such as user registrations, whereas metrics measure, for example, throughput. Experience shows that both data types require specialized tools. Although it is technically possible to write metrics into logs, it has proven to be a bad approach. It adds even more data to the logs, which usually are already quite huge. Also, calculating metrics from the logs takes additional time and processing power.

### Why Logs?

Logs are simple text files containing information about events that happened. This has several advantages.

- *Every programming language and infrastructure* supports log files. Thus, there is no lock-in and no restriction regarding the technologies used.
- The data are *permanently stored*, enabling tracing and analyzing of events long after they happened.
- Linear writing into a file is *very fast*. Therefore, logging hardly affects system performance.
- Log files are *easy to analyze*. The data is human-readable. Even simple tools like grep or tail make it easy to quickly get an overview and to analyze the data.

This simplicity is the strength of logs.

## Logs with Microservices

For microservices, the situation is dramatically different compared to traditional systems:

- Microservice instances *come and go*. It is not enough to store the log data in the instance. The instance and the stored log data can be lost.
- With microservices-based architectures, *a large number of systems* exist. Nobody can log into all these systems and analyze their log files. This requires much too much effort. So it is not enough to use simple tools like grep or tail.
- *New microservices* can be introduced whose logs also have to be analyzed.
- This requires a *centralized storage* of the logs for analysis. It has to collect the logs from all systems.
- Because much more log data has to be analyzed in a microservices system, the usual command line tools are not powerful enough for analysis. *More efficient tools* for the analysis of larger data volumes are necessary.
- The logs should be optimized for centralized storage and analysis via a tool – that is, be *machine-readable*. It is not necessary that they are human-readable.

## Log Information

To simplify log analysis, defining a uniform format for the logs is required.

An important part of the format is the log level. *Error* can stand for events which have a negative impact on users. Such a log entry can result from a complete failure of an operation. *Warning* can stand for an event where negative effects on the user were prevented. An example of such a scenario is when a system was not available, but the failure could be compensated by a default value. *Info* can refer to information with a business meaning, like a new registration. *Debug* contains details relevant only for developers.

Log messages can also store additional standardized information. For example, each incoming request can contain an ID that log messages then include. This allows the user to better understand the relationships between the log entries. When another microservice is called, the ID can also be transferred so that log messages caused by the request in other microservices can be traced in the logs. [Chapter 22](#) focuses on tracing, which is about the traceability of calls between microservices.

Also, additional context information can be output. This can include the current user or the web browser used.

## Sending Logs Instead of Storing Them

Logs are usually written to a file. In a microservices system, an overview of all microservices is necessary. All log information must be collected on a central server, not just written to a local file. So a process can read the local files and send them to the central server.

An alternative can be asynchronous communication. The log data is no longer stored in a file, but sent directly to the central server. The transmission of log data can be separated from the transmission of business data if the requirements in terms of reliability, speed, and so on are different.

## Tool: Map/Reduce

Companies like Google originally used *map/reduce* to analyze log data. With this approach, a function is applied to each log line. This is the map phase. In functional programming *map* means that a function is applied to a list of items. The map function can filter relevant data from the log line. This allows, for example, one to separate logs of user registrations from other events. In the *reduce* step, the individual data records are combined, for example, by counting. The result might be a statistic of user registrations.

Map/reduce can process large amounts of data and distribute the work well over many servers. But the results are only available after a long processing time, which is often unacceptable.

## Tool: Search Engines

In the meantime, other tools have been established to handle log data. Search engines are very fast and can process text data efficiently. They are also well suited for the analysis of large amounts of data.

Search engines are optimized to handle the addition of data particularly well. They are less able to deal with updates. Because log data is only added but never changed, search engines also fit well with log data from this perspective.

## Elasticsearch

Nowadays, search engines such as [Elasticsearch<sup>243</sup>](#) can do much more than just search text. They can also handle other data such as numbers or geodata. For this purpose, the search engines process structured data such as JSON documents.

Log entries also have a structure. They consist of a timestamp, a log level, information about the process or thread, and the actual log message. This structure should be available to the search engine to do more efficient searches. A tool such as [Logstash<sup>244</sup>](#) can parse the log data and pass it on to the search server as JSON.

However, to parse the log files and forward them as JSON to the search engine is actually nonsense. The microservice puts the data into a log entry, which is then parsed again. It would be much easier if the microservices would directly log JSON data. The [GELF format<sup>245</sup>](#) can be used to transmit JSON logs directly over the network. GELF plug-ins are even available for some logging frameworks, so that the use of Logstash or Filebeat can then be avoided.

If the data is sent to a server and not stored locally, log messages are actually nothing more than events. The logs are concerned with only technical events; see [chapter 10](#) for dealing with domain events. The boundaries are fluid. A customer's registration can be processed with a log system as well as a domain event, for example, to create a special report.

<sup>243</sup><https://www.elastic.co/products/elasticsearch>

<sup>244</sup><https://www.elastic.co/products/logstash>

<sup>245</sup><http://docs.graylog.org/en/2.3/pages/gelf.html>

## 21.2 Logging with the Elastic Stack

As an example of log processing in a microservices system, an installation of log tools is available in the Consul project (see chapter 15). Figure 21-1 shows the structure.

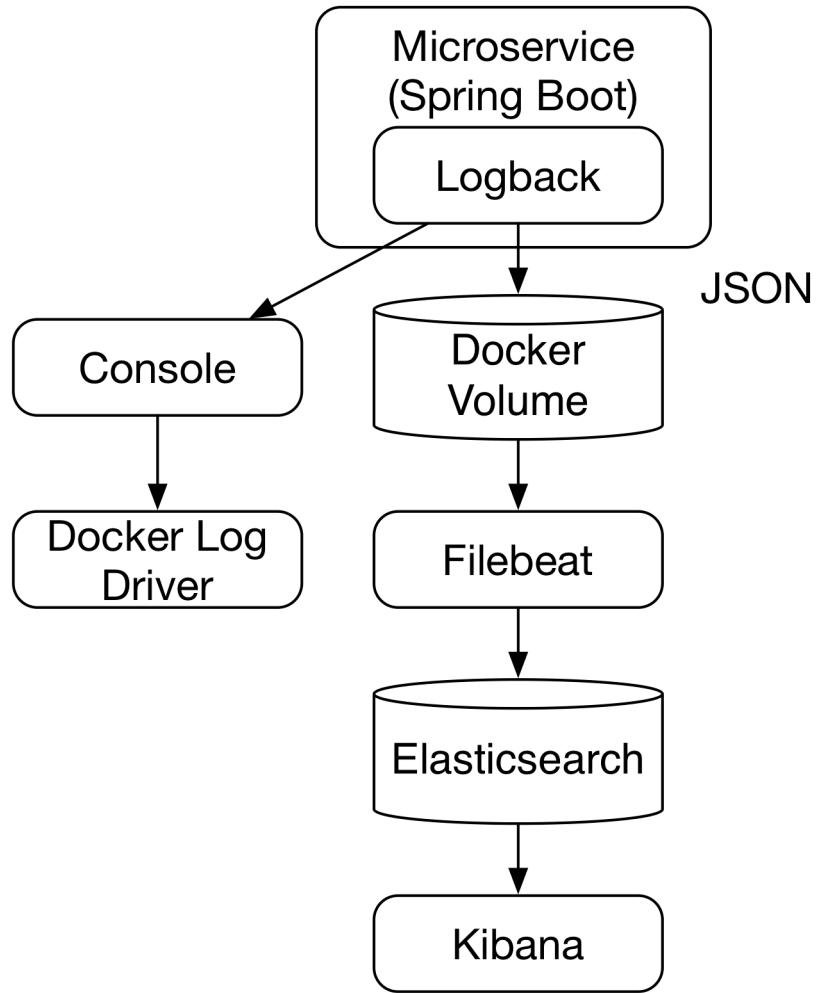


Fig. 21-1: Processing of Log Data

- The microservice is implemented with Spring Boot and uses **Logback**<sup>246</sup> as a library for logging. This library allows logging of JSON data. In a microservices architecture, it makes sense to standardize the format of the JSON data so that a uniform approach can be used for analysis. But standardizing the library is not necessary, because only the format of the data is relevant. A standardization of the library would unnecessarily restrict the freedom of technology.
- Logback outputs the log data on the *console*. This allows the developer to get an impression of what the application does when testing it locally. Docker also processes the output on the

<sup>246</sup><https://logback.qos.ch/>

console. Docker's log processing can also be the basis for common logging across different microservices.

- From the console, a *Docker Log Driver* processes the log information and provides it, for example, for the `docker log` command. The [Docker Log Driver](#)<sup>247</sup> can be configured in such a way that other types of processing are also possible.
- Logback also writes the log data in a JSON format to a *Docker volume*. This volume is shared by all microservices. From there, the data can be processed further. Actually, this volume is not necessary because the data should be stored primarily by the search engine on which the analyses are based. But in this way, a backup of the data also exists. An alternative would be to send the data directly over the network to Elasticsearch.
- [Filebeat](#)<sup>248</sup> reads the JSON data from the Docker volume and sends them to Elasticsearch. In the example, a single Filebeat instance exists for all logs. When a new microservice is added to the system, it only needs to log into a file with a different name. Filebeat reads this file automatically, so no additional configuration is necessary. In a production system, a separate instance of Filebeat for each microservice is the better approach to handle large amounts of data. Concepts like Kubernetes pods can be helpful for this because in a pod, the containers can easily share volumes. For example, a pod could host a microservice that writes logs to a volume. The Filebeat process in the same pod can then read data from this volume and send the data to Elasticsearch. Filebeat also adds some metadata to the log entries.
- Finally, [Elasticsearch](#)<sup>249</sup> stores the log data.
- [Kibana](#)<sup>250</sup> offers a user interface for visualizing, searching, and analyzing the data.

## 21.3 Example

Section 0.4 describes what software needs to be installed to start the example.

The example at <https://github.com/ewolff/microservice-consul> contains support for the Elastic Stack. You can download it with `git clone https://github.com/ewolff/microservice-consul.git`.

<https://github.com/ewolff/microservice-consul/blob/master/HOW-TO-RUN.md> discusses installation and use of the example in more detail. <https://github.com/ewolff/microservice-consul/blob/master/HOW-TO-RUN.md#run-the-elasticsearch-example> provides a detailed description of the structure of the example with the Elastic Stack.

To start the system with the log analysis, you first have to compile the application with Maven in the sub directory `microservice-consul-demo` using `./mvnw clean package` (macOS, Linux) or `mvnw.cmd clean package` (Windows). See [appendix B](#) for more details on Maven and how to troubleshoot the build. The configuration for Logback is contained in the file `logback-spring.xml` in each microservices project.

<sup>247</sup><https://docs.docker.com/engine/admin/logging/overview/#configure-the-default-logging-driver>

<sup>248</sup><https://www.elastic.co/products/beats/filebeat>

<sup>249</sup><https://www.elastic.co/products/elasticsearch>

<sup>250</sup><https://www.elastic.co/de/products/kibana>

Afterwards you can create the Docker container in sub directory docker with `docker-compose -f docker-compose-elastic.yml build`. The standard Docker configuration for the example does not have a Docker container for Kibana, Elasticsearch, or Filebeat. Changing this requires the configuration in file `docker-compose-elastic.yml`. With `docker-compose -f docker-compose-elastic.yml up -d`, you can start the application. See [appendix C](#) for more details on Docker, Docker Compose, and how to troubleshoot them.

The file `docker-compose-elastic.yml` contains, in addition to the Docker containers for microservices the Docker containers for Filebeat, Elasticsearch, and Kibana, which are depicted in [figure 21-1](#). It also contains the required Docker volume for storing the log files from which Filebeat imports the files into Elasticsearch.

The application is available at port 8080 on the Docker host. When the Docker containers are running locally, this is the URL <http://localhost:8080/>. Kibana is available at port 5601 (<http://localhost:5601>). When starting Kibana, the name of the relevant indices has to be given. The name is `filebeat-*`.

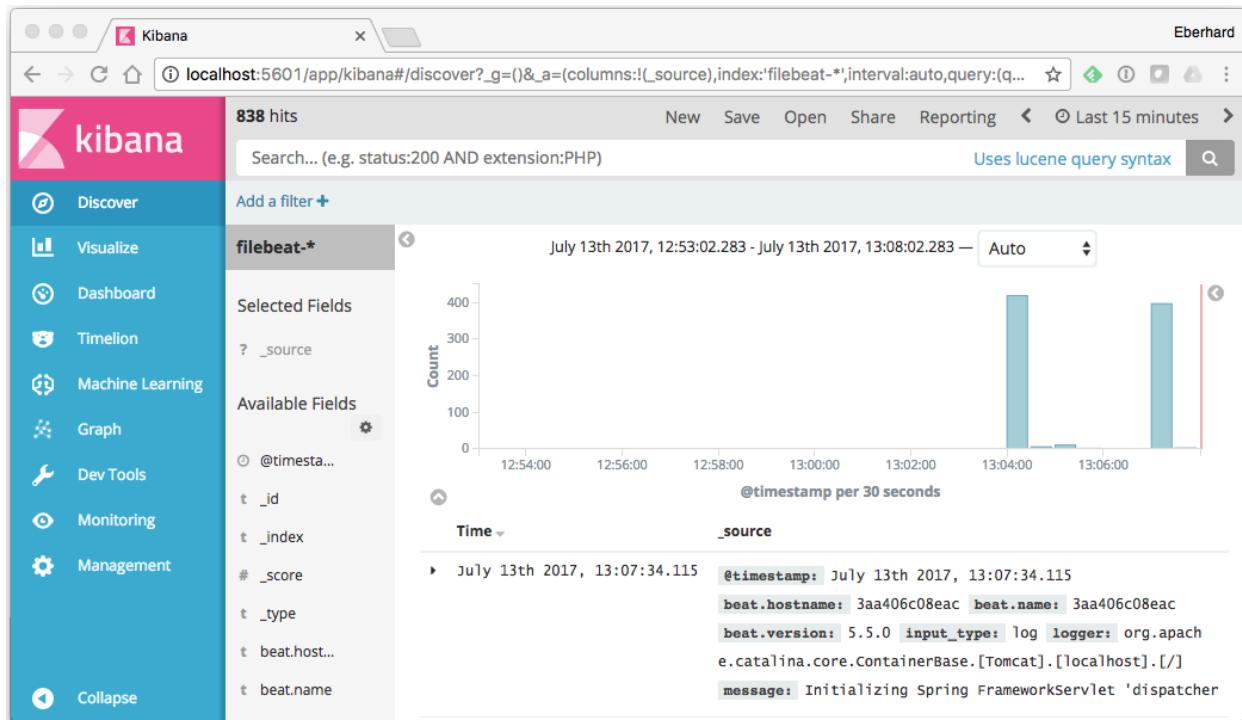


Fig. 21-2: Analysis with Kibana

Subsequently, the log data is available in Kibana ([figure 21-2](#)) and can be analyzed. The figure shows that the timestamp and other data, such as severity, are displayed as separate fields for which the user can search or filter.

## 21.4 Recipe Variations

It makes little sense to operate a microservices system without centralized analysis of the log data. There are simply too many microservices and microservices instances for manual analysis of the log data.

The Elastic Stack is very widely used; however, there are also alternatives.

- [Graylog](#)<sup>251</sup> also uses Elasticsearch for storing log data. In addition, it uses MongoDB for configuration and metadata. A web interface for analysis is integrated.
- [Apache Flume](#)<sup>252</sup> makes it possible to read data from one source, process it, and write it into a sink. It can read files and write in Elasticsearch, making it a possible alternative to Logstash. Compared with Flume or Logstash, Filebeat does not have so many options for parsing data.
- [Fluentd](#)<sup>253</sup> can also be an alternative for reading, processing, sending, and storing log data.
- Cloud services like [Loggly](#)<sup>254</sup> or [Papertrail](#)<sup>255</sup> spare the user the need to set up an infrastructure that can handle the large amounts of log data.
- Finally, [Splunk](#)<sup>256</sup> offers a commercial solution for the installation in your own data center as well as in the cloud.
- Microservices platforms can also offer log analysis ([chapter 16](#)). Cloud Foundry as PaaS ([chapter 18](#)) and Kubernetes ([chapter 17](#)) both offer support for log data.
- A service mesh like Istio (see [chapter 23](#)) adds proxies to the network traffic between the microservices. They can log information about the traffic.
- The example in [chapter 23](#) uses Elasticsearch and Kibana, too. But it sends JSON to Elasticsearch, eliminating the need to parse the log information.

## 21.5 Experiments

The Elastic Stack offers many interesting options.

- The Kibana reference documentation is a good basis to familiarize yourself with the analysis possibilities. It contains explanations for [discovery](#)<sup>257</sup> and [visualization](#)<sup>258</sup>.
- Integrate [Logstash](#)<sup>259</sup> into the setup. This requires a configuration to read and parse the files from Filebeat. The linked documentation can be a start for this.
- Replace the individual Elasticsearch instance with a cluster. The [documentation on starting with Elasticsearch](#)<sup>260</sup> can be helpful for this purpose.

<sup>251</sup><https://www.graylog.org/>

<sup>252</sup><https://flume.apache.org/>

<sup>253</sup><http://www.fluentd.org/>

<sup>254</sup><https://www.loggly.com/>

<sup>255</sup><https://papertrailapp.com/>

<sup>256</sup><https://www.splunk.com/>

<sup>257</sup><https://www.elastic.co/guide/en/kibana/current/discover.html>

<sup>258</sup><https://www.elastic.co/guide/en/kibana/current/visualize.html>

<sup>259</sup><https://www.elastic.co/guide/en/logstash/current/getting-started-with-logstash.html>

<sup>260</sup><https://www.elastic.co/guide/en/elasticsearch/reference/current/docker.html>

- Logging can be changed in such a way that the microservices directly send log data as JSON to Elasticsearch without saving the data beforehand. For this, the [Logback Elasticsearch Appender<sup>261</sup>](#) can be useful.
- Finally, the logs can be delivered directly by Docker using a [Docker Log Driver<sup>262</sup>](#). On the Internet, you will find instructions to also implement an integration into Elastic Stack using this method. Logstash can accept logs from different sources such as MOMs and send them to Elasticsearch.

## 21.6 Conclusion

Managing log data is very important in a microservices system because of the high number of microservices. The data must be collected and analyzed centrally. Instead of a simple solution with human-readable logs, a microservices system must have a solution that can handle large amounts of data and makes them available for machine analysis. The Elastic Stack can be used to create such an environment for log analysis.

### Advantages

- Elasticsearch can analyze large amounts of data quickly and scales well.
- The Elastic Stack is widely used, so a wealth of experience with this technology is already available.

### Challenges

- Log files can no longer be easily analyzed with simple command line tools, but only with the Elastic Stack.
- The setup of reliable and scalable log processing is complex due to the large amount of data.

---

<sup>261</sup><https://github.com/internetitem/logback-elasticsearch-appender>

<sup>262</sup><https://docs.docker.com/engine/admin/logging/overview/#configure-the-default-logging-driver>

# 22 Recipe: Tracing with Zipkin

In a distributed system, it may be necessary to track the path of a request across the microservices. This is done by tracing.

This chapter covers the following:

- The definition of tracing and in which situations tracing is useful.
- A concrete example of the tracing of a microservices system with Zipkin.
- Alternatives to Zipkin for tracking requests across the microservices.

## 22.1 Basics

Microservices are a distributed system. Microservices call each other. A problem with a microservice can be caused by the fact that one of the called microservices does not work or takes too long to respond to a request.

Tracing is particularly useful when microservices call each other. But also the calls of a database or an external system can add useful information to tracing.

### Is Tracing Necessary?

Calls between microservices should be the exception in a microservices system. Too much communication between the microservices over the network leads to an overhead and thus to poor performance. In addition, calls between microservices introduce potential failure due to a problem with the network or the called server. This makes such systems difficult to operate.

Such systems are also problematic in regards to maintenance because the calls represent dependencies between the microservices. Changing a microservice can be a challenge because of the dependencies; changes to other microservices may also have to be made. In this case, several microservices must be deployed, and coordinated deployment is required.

Ultimately, such a system violates the rule that modules should have high cohesion but loose coupling. A large amount of communication indicates a tight coupling.

Tracing is especially important for synchronous communication. The definition of synchronous communication allows a request to trigger another request, so true call trees exist. In asynchronous communication, a request can trigger another request only if it does not wait for a response. Because HTTP always sends a response, it is basically impossible for a request to cause another request, so cascades that require tracing do not really occur.

## 22.2 Tracing with Zipkin

[Zipkin<sup>263</sup>](#) is a tool for tracing. It has a server to which the tracing data can be sent. In addition, the data can be displayed with the UI.

### Zipkin: Structure

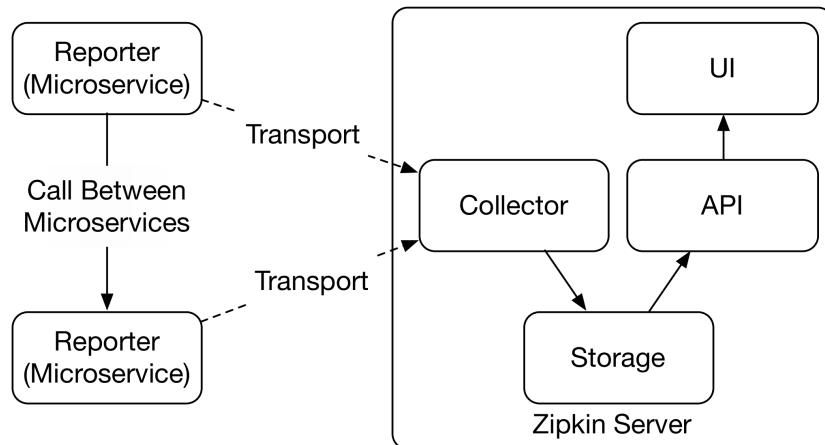


Fig. 22-1: Tracing Structure

Figure 22-1 shows the structure of Zipkin. As *reporters*, the microservices pass data to Zipkin with the help of a *transport*. Transport can be HTTP or Kafka. In order not to influence the performance of the system too much, the data is sent asynchronously by the reporters to the Zipkin server.

If a microservice calls another microservice, the called microservices also sends data through the transport to the server. That way, the Zipkin server can store data about the cascading requests.

In the *Zipkin server*, the *collector* receives the data and stores it in *storage*. This can be a system like Cassandra, Elasticsearch, or MySQL. The *API* provides access to the data and can run queries on the data. The user can analyze the data with the *UI*.

### Trace and Span ID

To trace the path of a call across the microservices, each call must be assigned a unique *trace ID*. This trace ID must be transferred with the subsequent calls. Another unique identifier exists for each call and each other measured time span: a *span ID*.

These concepts are not limited to REST, but can also be used with other communication protocols.

---

<sup>263</sup><http://zipkin.io/>

## Tracing in the Example

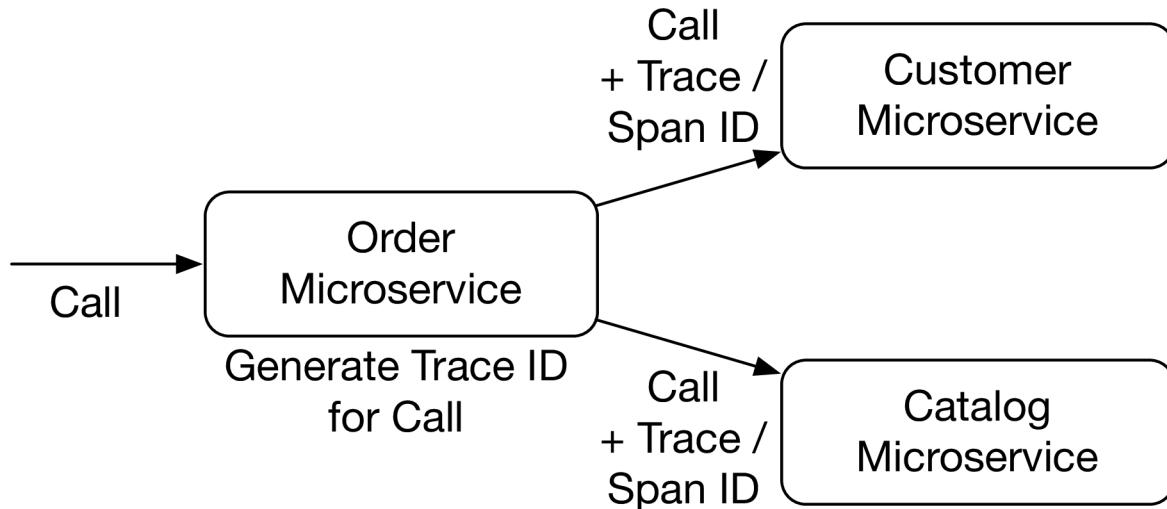


Fig. 22-2: Tracing in the Example

Figure 22-2 shows how tracing works in the example.

When the call arrives at the *order microservice*, the trace ID is generated. All calls to *customer* and *catalog* generated by the same call to *order* have the same trace ID. Each call to *customer* or *catalog* then has its own span ID.

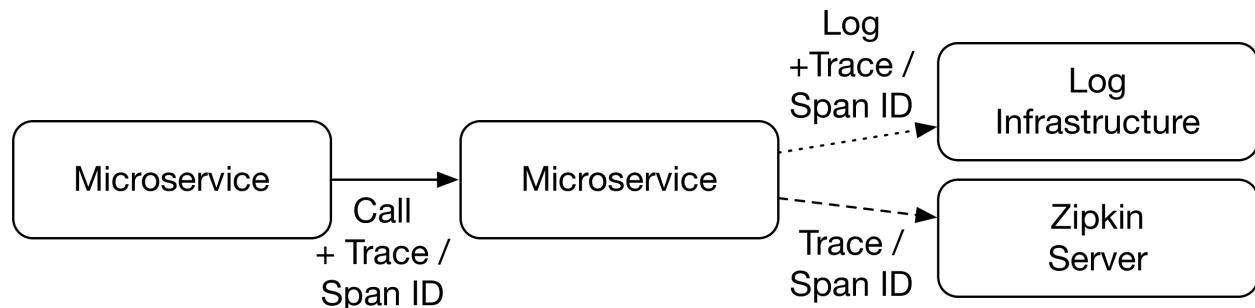


Fig. 22-3: Handling Tracing Data

Figure 22-3 shows what happens with the data: Trace ID and span ID are sent to the Zipkin server. The trace and span ID are also stored in the logs so that log entries can also be searched for trace IDs to find all logs for a specific call to the system.

[Spring Cloud Sleuth<sup>264</sup>](#) offers a very easy way to integrate Zipkin into a Spring Boot application. It ensures that trace IDs and span IDs are generated and forwarded during communication. It also transfers the trace data to the Zipkin server.

<sup>264</sup><https://cloud.spring.io/spring-cloud-sleuth/>

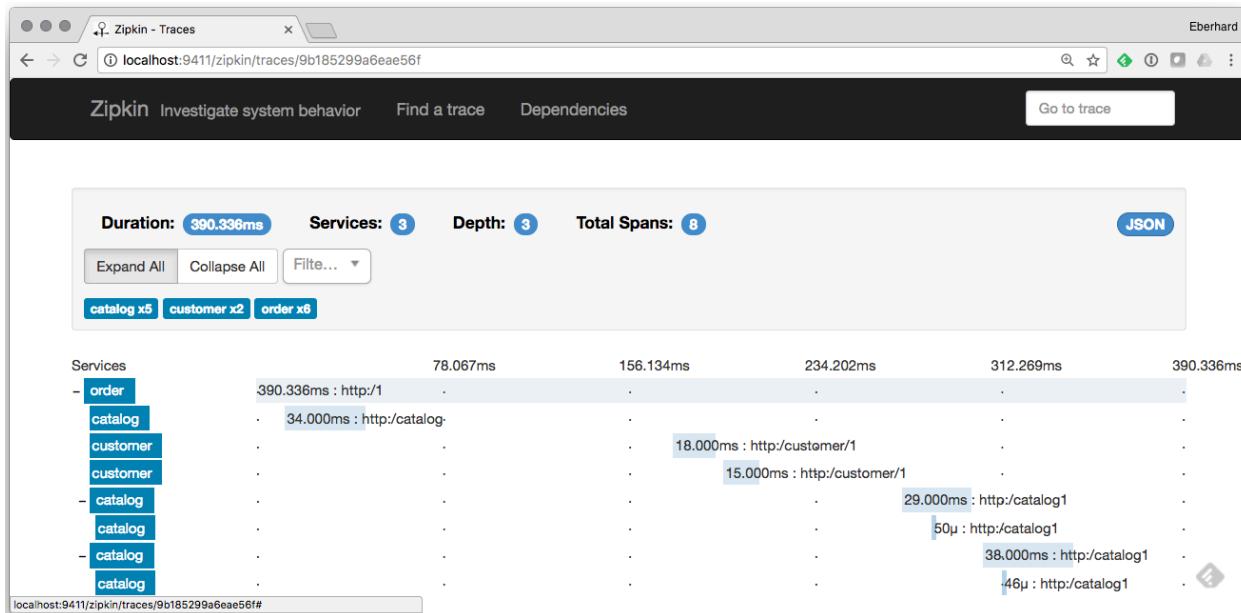


Fig. 22-4: Example with Zipkin

Figure 22-4 shows a trace where the order microservice calls the catalog and the customer microservice. The trace shows how much time is spent in the individual services.

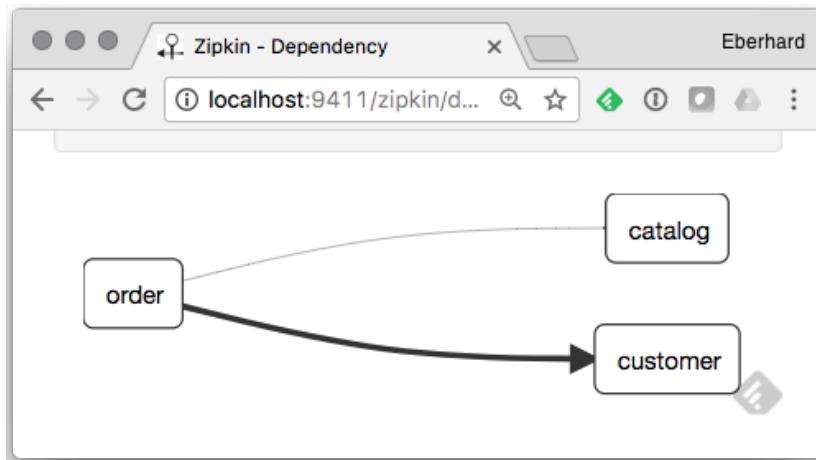


Fig. 22-5: Dependencies with Zipkin

Zipkin can also visualize the dependencies in the system (Figure 22-5). For a complex system, this can be helpful to get an overview of the interactions of the microservices.

## 22.3 Example

The Consul example project has an extension (see also chapter 15) that integrates Zipkin into the example project with Spring Cloud Sleuth. In Java projects, a dependency to `spring-cloud-starter-zipkin`

has to be added. Afterwards, the environment variable `SPRING_ZIPKIN_ENABLED` can be used to enable support for Zipkin, and `SPRING_ZIPKIN_BASE_URL` can be used to configure the URL of the Zipkin server. These settings are made in the file `docker-compose-zipkin.yml` so that Docker Compose passes them on to the microservices. See [appendix C](#) for more details on Docker, Docker Compose, and how to troubleshoot them.

The property `spring.sleuth.sleuth.sampler.percentage` in the file `application.properties` of the microservices project is set to the value 1.0 so that each span is forwarded to the Zipkin server. In a production system, this can be too much data. Because many calls are identical, a lower value may be enough. The default is 0.1 so that 10 percent of all spans are stored on the server. The IDs appear in the logs regardless of the set value. So it is still possible to find all log data from all microservices that belong to a specific request.

## How to Run the Example

[Section 0.4](#) describes what software has to be installed for starting the example.

First download the project with `git clone https://github.com/ewolff/microservice-consul.git`. Afterwards, you have to compile the application with Maven. To do so, you have to compile the code with `./mvnw clean package` (macOS, Linux) or `mvnw.cmd clean package` (Windows) in the sub directory `microservice-consul-demo`. See [appendix B](#) for more details on Maven and how to troubleshoot the build.

Subsequently, you can generate the Docker containers in sub directory `docker` with `docker-compose -f docker-compose-zipkin.yml build`. The configuration in file `docker-compose-zipkin.yml` contains a Docker container for the Zipkin server in addition to the normal system configuration. The configuration also enables support for Zipkin. With `docker-compose -f docker-compose-zipkin.yml up -d` you can start the application with Zipkin.

Afterwards, you can use the order microservice, and with Zipkin you can trace a call across several systems.

For this purpose, a Zipkin Docker container is configured in the Docker Compose configuration. It is available at port 9411 of the Docker host – that is, at <http://localhost:9411> if the Docker containers are running on the local computer.

At <https://github.com/ewolff/microservice-consul/blob/master/HOW-TO-RUN.md>, you can find detailed instructions for building and starting the example. <https://github.com/ewolff/microservice-consul/blob/master/HOW-TO-RUN.md#run-the-zipkin-example> explains how Zipkin can be added to the example.

## 22.4 Recipe Variations

Integration at the UI level does not require tracing. With a UI integration, what part of the UI belongs to what microservice is transparent. This makes troubleshooting relatively easy.

Zipkin supports many additional [libraries and frameworks<sup>265</sup>](#) so that tracing is also possible in a heterogeneous system.

Spring Cloud Sleuth can also be used, so that only the [trace IDs are transferred<sup>266</sup>](#). If the trace ID is also stored in the logs, then at least all log information for a request can be correlated in the log analysis.

An alternative to Zipkin is [Jaeger<sup>267</sup>](#). It supports [OpenTracing<sup>268</sup>](#), which standardizes tracing data formats and instrumentations. Istio is a service mesh and integrates many technologies useful for operating microservices. [Section 23.5](#) discusses Jaeger and its integration in Istio.

Commercial systems for monitoring such as [Dynatrace<sup>269</sup>](#), [New Relic,<sup>270</sup>](#) or [AppDynamics<sup>271</sup>](#) also offer similar features for monitoring distributed systems.

## 22.5 Conclusion

Tracing makes it possible to track calls across microservices. For synchronous microservices ([chapter 13](#)), tracing is especially useful if the microservices communicate frequently with each other. However, too much communication should be avoided in a microservices architecture because it is a sign of too many dependencies, which also have a detrimental effect on changeability and performance. In spite of the high demands placed on the operation of microservices, tracing is not absolutely necessary when the architecture is good enough to handle most requests in a single microservice.

The alternative to Zipkin is mainly consolidated logging ([chapter 21](#)) with a trace ID for each request. In this case, no special server is needed for tracing. Instead, only the trace ID has to be transferred in each request.

### Advantages

- With Spring Cloud, Zipkin is easy to integrate into the infrastructure.
- Zipkin allows extensive analyses.
- The correlation between log entries also makes logs easier to analyze.

### Challenges

- Tracing is only necessary when the dependencies between the components are complex.
- Infrastructure for tracing has to be set up and operated.

<sup>265</sup>[http://zipkin.io/pages/existing\\_instrumentations.html](http://zipkin.io/pages/existing_instrumentations.html)

<sup>266</sup>[http://cloud.spring.io/spring-cloud-static/spring-cloud-sleuth/1.2.5.RELEASE/single/spring-cloud-sleuth.html#\\_only\\_sleuth\\_log\\_correlation](http://cloud.spring.io/spring-cloud-static/spring-cloud-sleuth/1.2.5.RELEASE/single/spring-cloud-sleuth.html#_only_sleuth_log_correlation)

<sup>267</sup><https://www.jaegertracing.io/>

<sup>268</sup><http://opentracing.io/>

<sup>269</sup><https://www.dynatrace.com/platform/offerings/dynatrace/>

<sup>270</sup><https://newrelic.com/>

<sup>271</sup><https://www.appdynamics.com/>

# 23 Recipe: Service Mesh Istio

Service meshes are a type of infrastructure that supports typical challenges of microservices.

In this chapter, readers learn:

- What service meshes are.
- How service meshes solve most of the challenges of microservice with no impact on the code.
- What Istio as the most popular service mesh provides and how it can be used.

## 23.1 What Is a Service Mesh?

The last chapters have shown the technical challenges associated with microservices. [Chapter 4](#) explains that Docker simplifies the packaging and deployment of microservices. [Chapter 17](#) shows that Kubernetes provides service discovery, load balancing and routing for microservice. Kubernetes works with any type of microservice and has no impact on the code. However, the support for resilience is limited.

To fully support the operation of microservices, infrastructure for monitoring ([chapter 20](#)), log analysis ([chapter 21](#)) and tracing ([chapter 22](#)) also has to be provided.

So although a platform such as Kubernetes provides a lot of features, it still is somewhat incomplete.

### Istio

*Istio* supports these features for the operation of microservices. It also supports:

- For operation, Istio integrates technologies for monitoring, log analysis and tracing. The data for these features is measured by Istio, eliminating the need to add any code to the microservice. That way, Istio transparently solves the main issues for the operation of microservices systems.
- *Mutual TLS* (Transport Layer Security) adds encryption to the communication between the microservices. It also distributes certificates to the microservice. That way, each microservice can be authenticated. So even if an attacker is able to communicate with a microservice, he or she won't have such a certificate and the attack can easily be defended against. Istio also supports other security features such as authorization.
- Istio adds advanced features for *routing* to Kubernetes. That allows A/B testing for example. Using this approach, the requests of some users are forwarded to one version of the system, while the requests of other users are forwarded to a different version. That allows to find out which version is more attractive or generates more revenue. It is also possible to send the traffic to the new and old version of a microservice at the same time to make sure that the new version behaves like the old version (mirroring).

- Istio supports *resilience*. A timeout can be added to a request and requests can be retried. Also, a circuit breaker can be used to protect microservices from overload.

Istio does not solve all challenges of microservices. A different infrastructure such as Kubernetes still has to provide basic features for example, for deployment and service discovery.

Istio is licensed under the liberal Apache license. It allows users to freely modify and distribute the code. Google and IBM started the project in partnership with the team for the Envoy proxy at Lyft.

## 23.2 Example

The example in this chapter contains the same microservices as the example in the Atom chapter (see [section 12.2](#)). However, they use a custom data format instead of Atom.

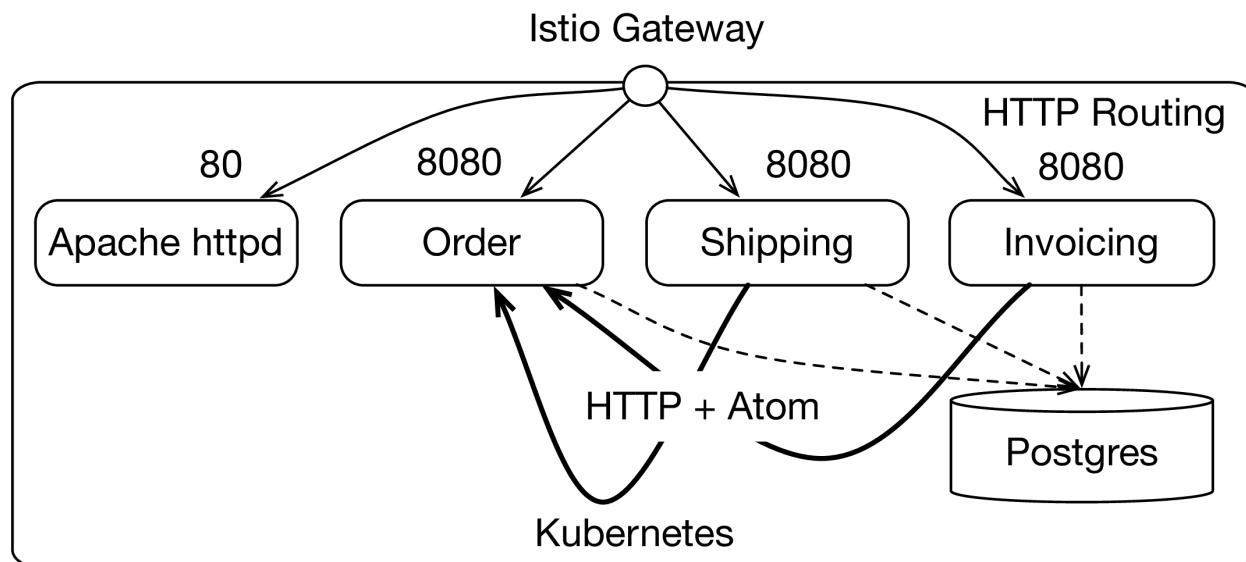


Fig. 23-1: Overview of the Service Mesh Example

[Figure 23-1](#) shows the structure of the example:

- Istio provides the *Ingress Gateway*. It forwards HTTP requests to the microservices. It is similar to a Kubernetes Ingress (see [section 17.2](#)). However, the Istio Gateway supports Istio's features mentioned previously, such as monitoring or advanced routing.
- *Apache httpd* provides a static HTML page that serves as the home page for the example. The page has links to each microservice. Apache httpd is configured by a Kubernetes deployment. That deployment creates a Kubernetes replica set with one Kubernetes pod. A Kubernetes service and a route for the Ingress gateway ensure access to the Apache httpd.
- *Order*, *shipping*, and *invoicing* are microservices. Shipping and invoicing poll data about the orders from the order microservice. They use a feed as described in [chapter 12](#). However, the

data format is custom and based on JSON as described in [section 12.3](#). The feed contains a simple JSON documents with a list of link to the individuals orders. The microservices each have a deployment, a Kubernetes service and a route for the Ingress gateway like the Apache httpd just mentioned.

- All three microservices use the same *Postgres database*. However, they use different database schemas. So each microservice can change its schema and therefore its domain model without any impact on the other microservices. A Kubernetes deployment installs the Postgres database, and a Kubernetes service ensures access to it.

So for load balancing, service discovery and deployment the example relies on Kubernetes with the concepts described in [chapter 17](#).

The reasons for choosing the example in this chapter are:

- The example uses REST. Some of Istio's features support HTTP and HTTP/2 which REST uses.
- It is based on an asynchronous architecture. A synchronous REST system would be an even better example for Istio's features. However, the asynchronous architecture benefits resilience and solves a core challenge for microservices.

## Run the Example

[Section 0.4](#) describes what software has to be installed for starting the example.

You can access the code for the example at <https://github.com/ewolff/microservice-istio>. The [documentation](#)<sup>272</sup> explains in detail how to install the required software and how to run the example.

The following steps are necessary for running the example:

- [Minikube](#)<sup>273</sup> is a minimal Kubernetes version that must be installed. Instructions for this can be found at <https://github.com/kubernetes/minikube#installation>.
- [kubectl](#)<sup>274</sup> is a command line tool for handling Kubernetes and also has to be installed. Its installation is described at <https://kubernetes.io/docs/tasks/tools/install-kubectl/>.
- [Download](#)<sup>275</sup> and [install](#)<sup>276</sup> Istio. It is enough to install Istio *without* mutual TLS authentication between sidecars.
- Change to the directory `microservice-istio-demo` and run `./mvnw clean package` or `mvnw.cmd clean package` (Windows) to compile the Java code. [Appendix B](#) discusses Maven in more detail and contains hints on how to troubleshoot the build.
- Configure Docker so that it uses the Kubernetes cluster. This is required to install the Docker images: `minikube docker-env`(MacOS or Linux) or `minikube.exe docker-env` (Windows) tells you how to achieve that.

---

<sup>272</sup><https://github.com/ewolff/microservice-kubernetes/blob/master/HOW-TO-RUN.md>

<sup>273</sup><https://github.com/kubernetes/minikube>

<sup>274</sup><https://kubernetes.io/docs/user-guide/kubectl-overview/>

<sup>275</sup><https://istio.io/docs/setup/kubernetes/download-release/>

<sup>276</sup><https://istio.io/docs/setup/kubernetes/quick-start/>

- Run `docker-build.sh` in the directory `microservice-istio-demo`. It builds the images and uploads them into the Kubernetes cluster. [Appendix C](#) discusses the docker command line tool in more detail and shows some ways to troubleshoot it. If you are not on a system with a shell that can run the script, you can issue the commands in the script manually.
- Make sure that the Istio containers are automatically injected when the pods are started:  
`kubectl label namespace default istio-injection=enabled`
- Deploy the infrastructure for the microservices using `kubectl apply -f infrastructure.yaml` in the directory `microservice-kubernetes-demo`. This creates the Apache httpd server, the Postgres server and the Istio gateway. It also adds a route from the Istio gateway to the static HTML pages stored in the Apache httpd server.
- At this point, you can deploy the microservices using `kubectl apply -f microservices.yaml`. This configuration contains the deployment, services, and routes for the microservices order, invoicing, and shipping.
- To use the demo, you need to figure out the URL of the Istio gateway. The shell script `ingress-url.sh` does that.
- Open the URL in a web browser. The static overview page with links to the microservices should appear. You should be able to enter an order in the order microservice. After a while, the shipment and the invoice should appear in the other microservices. The microservices poll every 30 seconds, so there might be a slight delay.

If you run into any problems, refer to the [online documentation<sup>277</sup>](#). It contains even more extensive documentation and some strategies for troubleshooting.

## Adding another Microservice

There is another microservice in the sub directory `microservice-istio-bonus`. This microservice has a completely separate build. A separate build might be the better option compared to the shared build of the other microservices. Microservices should be separately deployable. Because the build is the first step in a deployment pipeline, the build should ideally be separate, too.

To add the microservice, you can do the following:

- Change to the directory `microservice-istio-demo` and run `./mvnw clean package` or `mvnw.cmd clean package` (Windows) to compile the Java code.
- Run `docker-build.sh` in the directory `microservice-istio-bonus`. It builds the images and uploads them into the Kubernetes cluster.
- Deploy the microservice with `kubectl apply -f bonus.yaml`.
- You can remove the microservice again with `kubectl delete -f bonus.yaml`.

## Adding a Microservice with Helm

The configuration of the microservice in `bonus.yaml` is very similar to the configuration of the other microservices. It makes little sense to repeat the common parts of the configuration for every microservice. [Helm<sup>278</sup>](#) is a package manager for Kubernetes that supports templates for Kubernetes

<sup>277</sup><https://github.com/ewolff/microservice-istio/blob/master/HOW-TO-RUN.md>

<sup>278</sup><https://helm.sh/>

configurations. Helm can make it easier to add a new microservice.

- First you need to [install Helm<sup>279</sup>](#) into the Kubernetes cluster.
- The directory `spring-boot-microservice` contains a Helm chart for the microservices in this project. With Helm, there is no need to run `kubectl apply -f bonus.yaml`. Instead, `helm install --set name=bonus spring-boot-microservice/` is enough to deploy the microservice. For a different microservice, just provide a different name. So the Helm chart provides a generic mechanism to deploy microservices. A Helm deployment is also called a “release.”

```
[~/microservice-istio]helm install --set name=bonus  spring-boot-microservice/
NAME: waxen-newt
LAST DEPLOYED: Wed Feb 13 14:25:52 2019
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Service
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
bonus    NodePort   10.108.31.211  <none>        8080:30878/TCP  5s

==> v1beta1/Deployment
NAME      DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
bonus     1         1         1           1          5s

==> v1alpha3/VirtualService
NAME      AGE
bonus    1s

==> v1/Pod(related)
NAME                  READY  STATUS    RESTARTS  AGE
bonus-55d854b9d9-sn4pm  2/2    Running   0          4s
```

- A name is automatically assigned to the Helm release. In this example, the name is `waxen-newt` and it is printed out by Helm. `helm list` provides a list of currently installed releases.
- You can remove the Helm release by using this assigned name – for example `helm delete waxen-newt`.

Helm offers a very easy way to deploy microservices that match the structure of the other microservices. In a system with many microservices, Helm charts simplify deployment of microservices and also make the deployments more uniform.

---

<sup>279</sup>[https://docs.helm.sh/using\\_helm/#installing-helm](https://docs.helm.sh/using_helm/#installing-helm)

## Using the Additional Microservice

The bonus microservice is not included in the static web page, so no link to it exists. However, you can access the bonus microservice via the Ingress gateway. If the Ingress gateway's URL is <http://192.168.99.127:31380/>, you can access the bonus microservice at <http://192.168.99.127:31380/bonus>.

Note that the bonus microservice does not show any revenue for the orders. This is because it requires a field `revenue` in the data that the order microservice provides. That field is currently not included in the data. This shows that adding a new microservice might require changes to a common data structure. Such changes might also impact the other microservices because the other microservices use the shared data structure, too.

## 23.3 How Istio Works

A closer look at the `microservices.yaml` file shows that the deployment for the microservices has no Istio-specific information. This is an advantage that makes it easier to use Istio. It also means that Istio works with any type of microservice no matter what programming language or frameworks are used.

However, Istio supports features such as resilience and monitoring as mentioned. Somehow Istio needs to collect information about the microservices.

### Sidecar

The idea behind a *sidecar* is to add another Docker container to each Kubernetes pod. Actually, if you list the Kubernetes pods with `kubectl get pods`, you will notice that for each pod it says 2/2:

```
[~/microservice-istio/microservice-istio-demo] kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
apache-7f7f7f79c6-jbqx8   2/2     Running   0          8m51s
invoicing-77f69ff854-rpcbk 2/2     Running   0          8m43s
order-cc7f8866-9zbnf      2/2     Running   0          8m43s
postgres-5dddbbf8f-xfng5  2/2     Running   0          8m51s
shipping-5d58798cdd-9jqj8 2/2     Running   0          8m43s
```

So while there is just one Docker container configured for each Kubernetes pod, two Docker containers are in fact running. One container contains the microservice, and the other contains the sidecar that enables the integration into the Istio infrastructure.

Istio automatically injects these containers into each pod. During the installation described previously, `kubectl label namespace default istio-injection=enabled` marked the `default` namespace so that Istio injects sidecars for each pod in that namespace. Namespaces are a concept in Kubernetes to separate Kubernetes resources. With Istio, the `default` namespace contains all Kubernetes that the user provides. The namespace `istio-system` contains all Kubernetes resources that belong to Istio itself.

## Proxy

Just injecting a sidecar is not enough. The Istio sidecar has to get information and metrics about the microservice. Istio routes all outgoing and incoming traffic through a *proxy*. This proxy<sup>280</sup> is an extended version of the Envoy proxy<sup>281</sup>. All network traffic passes through the proxy, so the proxy can modify and measure all traffic. That makes the service mesh transparent and enables its features.

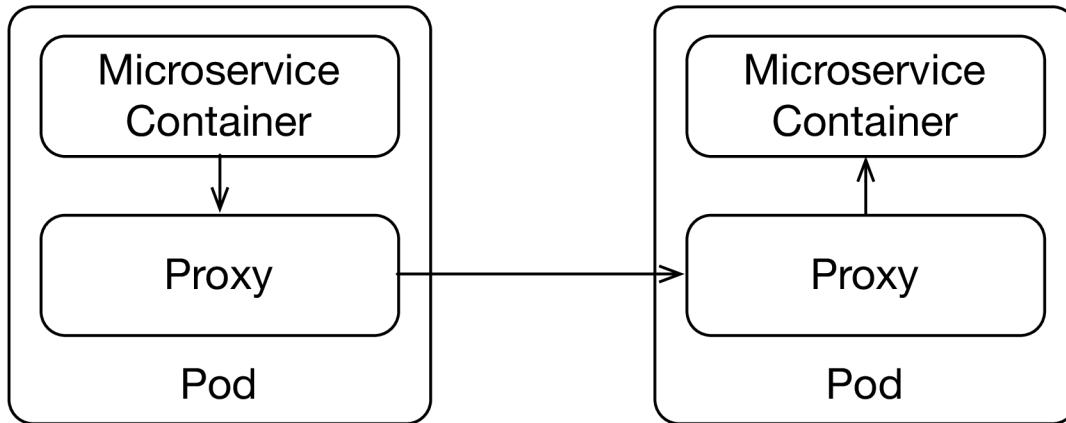


Fig. 23-2: Proxy and Data Plane in Istio

Figure 23-2 shows that all traffic from one microservice container (for example, invoicing) to another container (for example, order) is routed through two proxies. So it seems the microservice is communicating directly with the other microservice but in fact proxies intercept the communication.

The Istio proxy can handle any TCP-based protocol. However, it has specific support for HTTP 1.1, HTTP 2, and gRPC. For those protocols, it can determine whether an operation was successful or not by evaluating, for example, the HTTP status code.

## Data Plane

The proxies are the part of Istio also called the *data plane*. It is responsible for the exchange of data between microservices.

## Control Plane

To do anything meaningful with the intercepted traffic, Istio uses the *control plane*. It is responsible for configuring the proxies in the data plane and processing data from the proxies.

<sup>280</sup><https://github.com/istio/proxy>

<sup>281</sup><https://www.envoyproxy.io/>

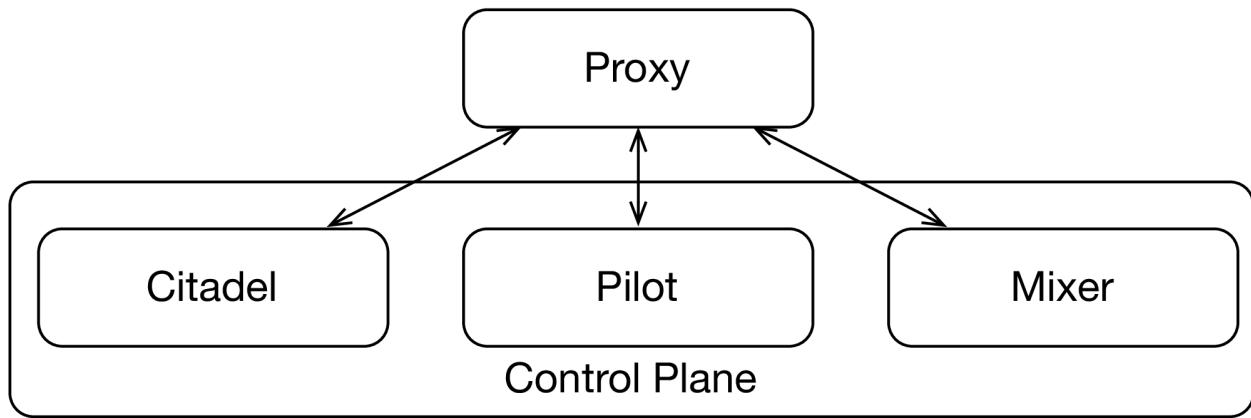


Fig. 23-3: Control Plane in Istio

Figure 23-3 shows that the control plane consists three components:

- *Pilot* converts the routing information in Istio into configuration for the proxies. This enables A/B testing, for example. It also supports resilience with timeouts or circuit breakers, for instance, and abstracts away service discovery mechanisms of the underlying platform. This enables Istio to support other infrastructures besides Kubernetes. So, Istio also supports a basic infrastructure such as Consul rather than Kubernetes.
- For each request, *Mixer* evaluates policies. These are used, for example, to determine what kind of metrics should be measured for a request and what components should receive them. But the policies might also define which microservice can call a specific microservice, or they can enforce quotas.
- *Citadel* manages certificates and cypher keys to enable encryption. It also support authentication based on the identity of a microservice and authentication for end users. An end user is authenticated with a token. Access to specific microservices can be limited to specific users. A [hands-on example<sup>282</sup>](#) is available for this.

So by injecting the proxy into the network traffic between microservices, Istio transparently enables many features that otherwise would need to be implemented in the microservice.

## 23.4 Monitoring with Prometheus and Grafana

As discussed in [chapter 20](#), a microservices system should include a monitoring infrastructure that collects monitoring information from all microservices and makes them accessible. This is required to keep track of the metrics for the huge number of distributed microservices. You can implement alarms and analysis based on these metrics.

<sup>282</sup><https://istio.io/docs/tasks/security/authn-policy/>

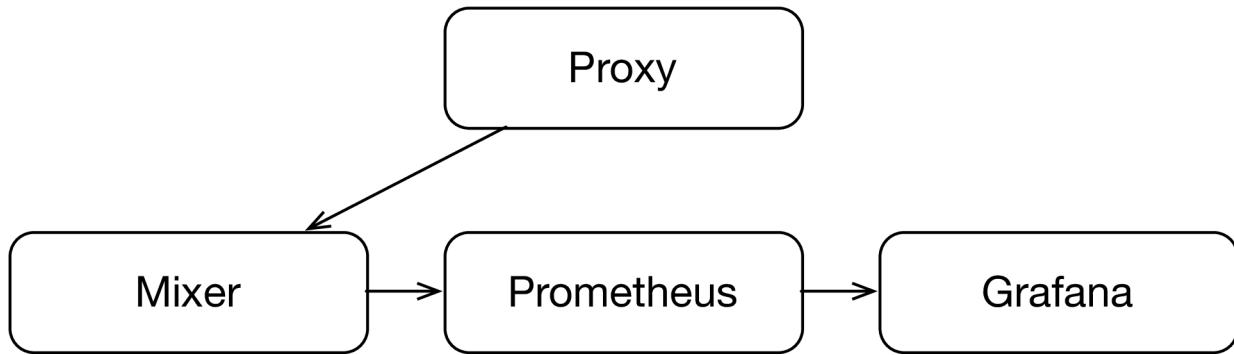


Fig. 23-4: Monitoring in Istio

Figure 23-4 shows how Istio supports monitoring:

- The *proxy* collects metrics such as the duration of a request on the client and server side, the status code and the number of requests.
- This information is collected by *Mixer*.
- *Prometheus*<sup>283</sup> (see chapter 20) then stores these metrics for analysis.
- *Grafana*<sup>284</sup> provides more advanced tools for the analysis of the metrics.

With a default Istio installation, all of these components are configured, installed and ready to be used, requiring no additional effort to enable monitoring for the microservices system.

Of course, this approach supports only metrics that the proxy can measure. This includes all the information about the request, such as its duration or the status code. Also, information about the Kubernetes infrastructure – for example, CPU utilization – can be measured. However, data about the internal state of the microservice is not measured. To get that data, the microservice would need to report it to the monitoring infrastructure.

The metrics are sent asynchronously rather than synchronously; it is acceptable to receive and process the metrics at a later point in time. By doing more communication asynchronously, the additional latency involved with synchronous communication can be reduced.

## Monitoring in the Example

The example uses only the metrics provided by the proxies and the Kubernetes infrastructure for monitoring. This is different from the example in chapter 20. The example in that chapter relies on the microservices to provide the metrics. However, Istio's approach used in this chapter can be a good alternative.

- It has no impact on the code of the microservice whatsoever. Metrics are measured only by the proxy, so any microservice will report the same metrics – no matter in what programming language they are written or which framework they use.

<sup>283</sup><https://prometheus.io/>

<sup>284</sup><https://grafana.com/>

- The metrics give a good impression about the state of the microservice. The metrics show the performance and reliability that a user or client would see. This is enough to ensure that service level agreements and quality of service is met. With an infrastructure such as Kubernetes and Istio, activities such as restarts of failing microservices are automated. So some metrics for which an administrator would restart a service are no longer important.

The metrics provided by Istio might be enough to manage a microservices system. In that case, a macro architecture rule for monitoring (see [section 2.3](#)) isn't necessary; it is still possible to operate the system.

## Prometheus in the Example

Metrics only make sense if there is load on the system. The shell script `load.sh` uses the tool `curl` to request a certain URL 1,000 times. You can start one or multiple instances of this script with the URL of the home page of the shipping microservice to create some load on that service.

The script `monitoring-prometheus.sh` uses `kubectl` to create a proxy for the Prometheus server on localhost. That way, you can access Prometheus with the URL <http://localhost:9090/>. You can use metrics like `istio_requests_total` or `istio_request_bytes_count` as an example for the metrics Istio reports to Prometheus. The metrics are multi-dimensional – that is, they can be summed up by HTTP status code, source, or destination.

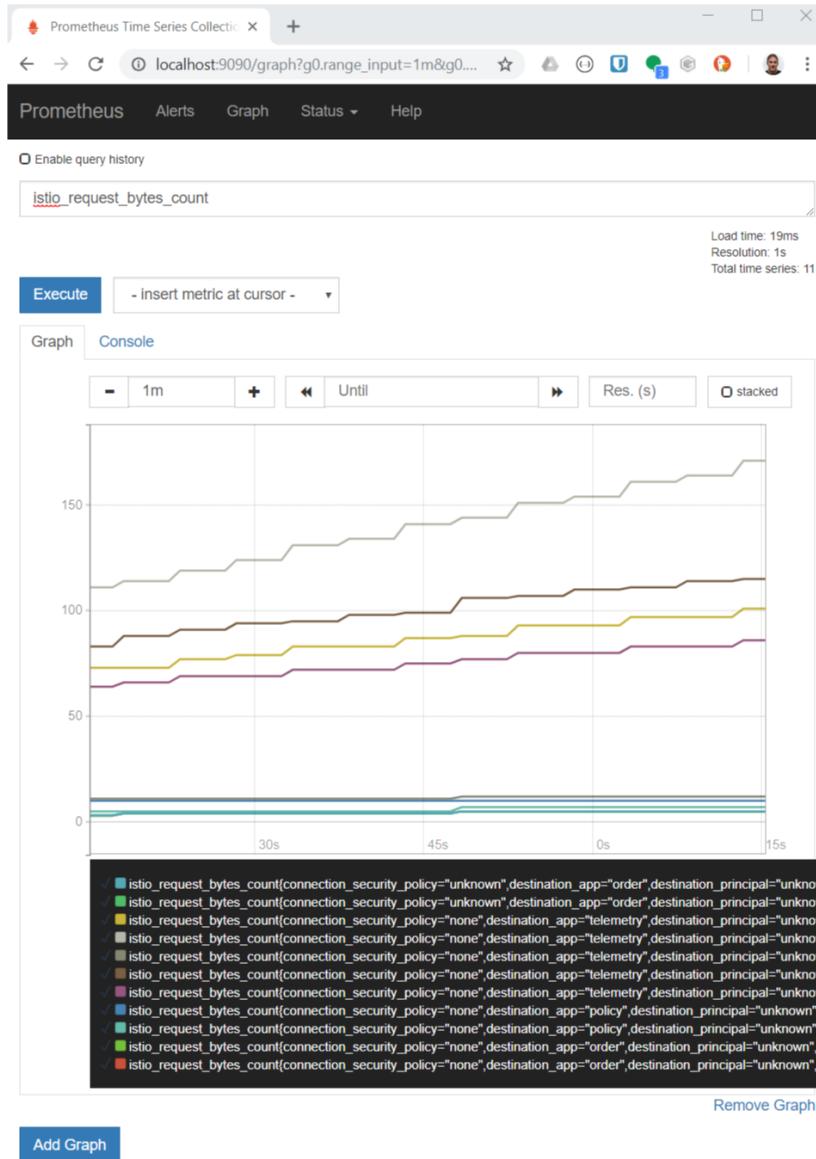


Fig. 23-4: Prometheus with Istio Metrics

For example, Figure 23-4 shows the byte count for requests and the different destinations: the order microservice and also the Istio component that measures the telemetry data. The destination is one dimension of the data. These metrics could be summed up by dimensions such as the destination, to understand which destination receives how much traffic.

## Grafana in the Example

For more advanced analysis of the data, Istio provides an installation of Grafana. The easiest way to use the Grafana installation is to start the shell script `monitoring-grafana.sh`. It creates a proxy on localhost for the Grafana installation. You can then use the URL <http://localhost:3000/> to access the Grafana installation.

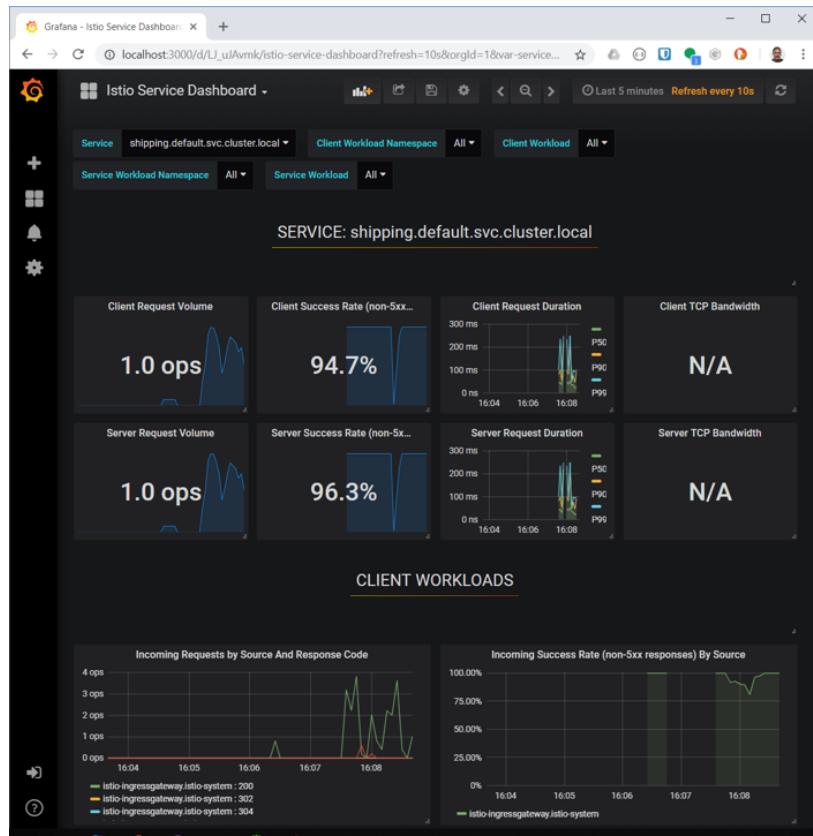


Fig. 23-5: Grafana with Istio Dashboard

The Grafana installation in Istio provides predefined dashboards. Figure 23-5 shows an example of the Istio service dashboard. It uses the shipping microservice. The dashboard shows metrics such as the request volume, the success rate and the duration. This gives a great overview about the state of the service.

Istio supplies dashboards for other things than the microservices. The Istio performance dashboard provides a general overview about the state of the Kubernetes cluster with metrics such as memory consumption or CPU utilization. The Istio mesh dashboard shows a global metric about the number of requests the service mesh processes and their success rates.

## 23.5 Tracing with Jaeger

For tracing, Istio uses [Jaeger<sup>285</sup>](https://www.jaegertracing.io/). This tool is similar to [Zipkin<sup>286</sup>](https://zipkin.io/) (see chapter 22). Jaeger supports collecting tracing information with the Zipkin format but also with the [OpenTracing<sup>287</sup>](https://opentracing.io/) standard. Jaeger is easier to deploy on Kubernetes, for which official templates are available to do the deployment on that infrastructure.

<sup>285</sup><https://www.jaegertracing.io/>

<sup>286</sup><https://zipkin.io/>

<sup>287</sup><https://opentracing.io/>

Tracing solves a common problem in microservices systems. A request to a microservice might result in other requests. Tracing helps to understand these dependencies, thus facilitating root cause analysis.

To understand which incoming request caused which outgoing requests, Jaeger relies on HTTP header. The values in the headers of the incoming requests have to be added to any outgoing request. This means that tracing cannot be transparent to the microservices. They have to include some code to forward the tracing headers from the incoming request to each outgoing request. For Jaeger, these headers are `x-request-id`, `x-ot-span-context`, `x-b3-traceid`, `x-b3-spanid`, `x-b3-parentspanid`, `x-b3-sampled`, and `x-b3-flags`.

## Tracing in the Example

The example uses [Spring Cloud Sleuth<sup>288</sup>](#). This is a powerful library that supports many features for tracing. The example in [chapter 22](#) used Spring Cloud Sleuth to measure and report the tracing information to Zipkin. However, for the example in this chapter, Spring Cloud Sleuth just needs to forward the HTTP headers. So in `application.properties` the parameter `spring.sleuth.propagation-keys` contains the HTTP headers that must be forwarded. The `x-b3-*` headers are automatically forwarded by Spring Cloud Sleuth so just the `x-request-id` and `x-ot-span-context` header have to be configured.

The example in [chapter 22](#) relies on Spring Cloud Sleuth to also publish the trace data to the Zipkin. This is not necessary for the example in this chapter. The Istio proxies forward the information.

Other languages require different means to forward the HTTP headers. So concerning the macro architecture (see [section 2.3](#)), the only necessary rule is to forward the HTTP headers. How the microservices implement this feature is up to them. The easiest way would be to use Spring Boot and Spring Cloud Sleuth. However, there is no need for a strict rule to use the same technology. This ensures that technology freedom is preserved.

To see the tracing information, use `tracing.sh` to start a proxy on localhost. Then you access the Jaeger UI at <http://localhost:16686/>.

---

<sup>288</sup><https://spring.io/projects/spring-cloud-sleuth>

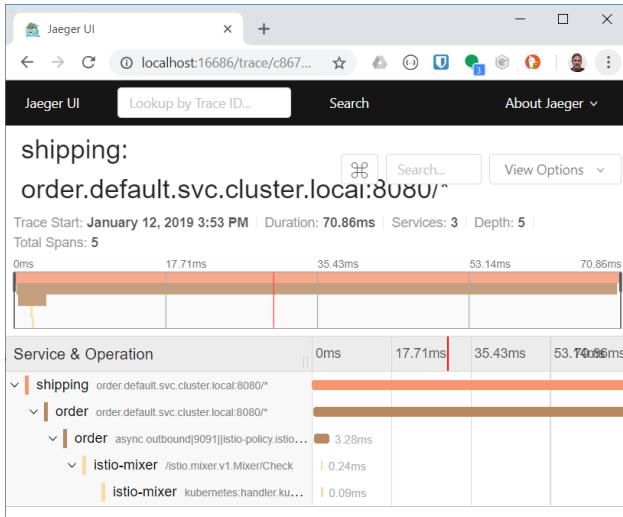


Fig. 23-6: Jaeger Trace

Figure 23-6 shows an example of a trace for a request to the shipping microservice. The user started a poll for new data on the order microservice. Then the service contacted the Istio Mixer to make sure the policies are enforced.

The trace just shows the interaction of two microservices (shipping and order). It is not very complex. Most of the time is spent in the order microservice and so the trace does not provide a lot of useful information to improve performance. The only way to make this request faster is to make the order microservice faster. However, the trace does not show where the order microservice spends its time.

This lack of potential to optimize based on the trace is a result of the architecture. Almost all requests are handled by a single microservice. The trace in the figure is one of the few exceptions but even that example includes only two microservices. Therefore, the trace does not provide a lot of valuable information. However, that also means that the distributed nature of the system has not such a huge impact, and the system is still easy to deal with.

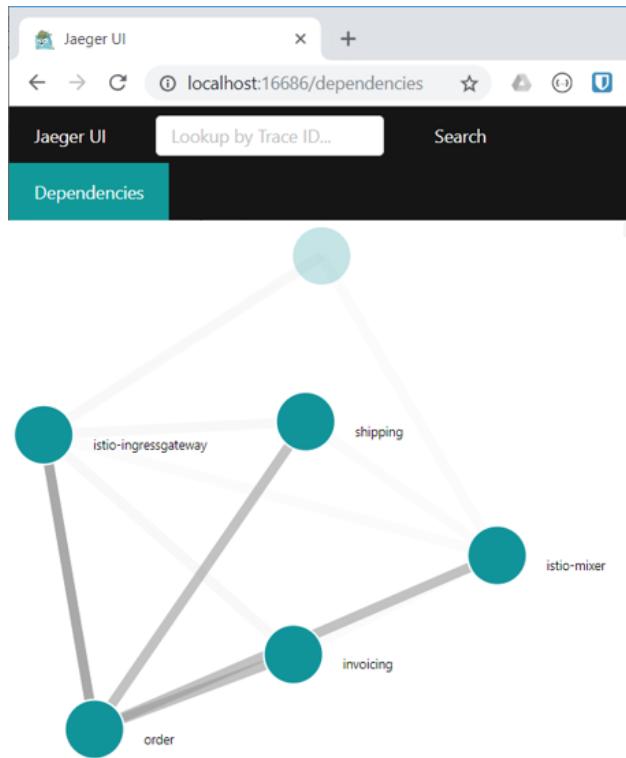


Fig. 23-7: Jaeger Dependencies

Figure 23-7 shows a different type of information Jaeger provides: the dependencies between the microservices. Shipping and invoicing use order to receive the information about the latest orders. Order reports metrics to Mixer. And finally, order is accessed by the Istio gateway when external requests are forwarded to it. This information about dependencies might be useful to get an overview about the architecture of the system.

## 23.6 Logging

Mixer can forward information about each HTTP request to a logging infrastructure. That information can be used to analyze, for example, the number of requests to certain URLs and status codes. A lot of statistics for web sites rely on this kind of information.

The format in the logs can be configured. A log entry may contain any information Mixer received from the request.

However, Istio does not provide an infrastructure to handle logs. To make use of the logging information, some storage for the logs, such as Elasticsearch, is needed. Also, an analysis frontend like Kibana is necessary. This is not part of a standard Istio installation.

This [example<sup>289</sup>](#) discusses how Istio support logs, showing how log information can be composed from Mixer's data. This example outputs the logs to stdout and not to a log infrastructure.

<sup>289</sup><https://istio.io/docs/tasks/telemetry/metrics-logs/>

Also, this [example<sup>290</sup>](#) shows how [Fluentd<sup>291</sup>](#) collects the logs Istio provides from all microservices. The logs are stored in Elasticsearch and evaluated with Kibana.

## Logs in the Example

For the example, a custom log infrastructure was set up. This infrastructure uses Elasticsearch to store logs and Kibana to analyze them – just like the example in [chapter 21](#).

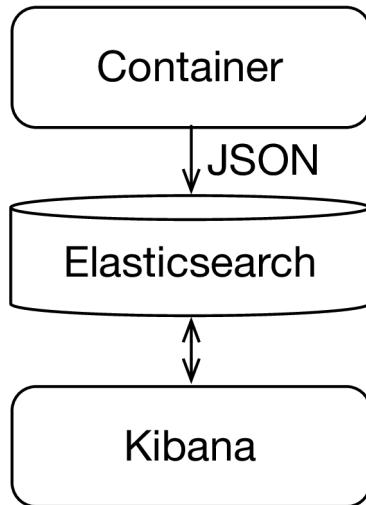


Fig. 23-8: Logging in the Example

[Figure 23-8](#) shows how logging is implemented in the example. Each microservice must directly write JSON data to the Elasticsearch server. So there is no need to write any log files which makes the system easier to handle. The need to parse the log information has also been eliminated; Elasticsearch can directly process it.

The example uses the [Logback<sup>292</sup>](#) Java library. The [Logback Elasticsearch Appender<sup>293</sup>](#) forwards the logs to Elasticsearch. The configuration in the file `logback-spring.xml` defines what information the microservices log.

However, concerning the macro architecture (see [section 2.3](#)), there would only be a rule that requires microservices to log JSON to the Elasticsearch server. There would probably also be a rule about which information should be included in a log entry – for example, the host, severity, and of course the message. Not just Java microservices but also microservices written in a different language or using a different library could also conform to such a macro architecture rule.

The Istio infrastructure could log to the same Elasticsearch instance, too. However, for the example, it was decided that this is not necessary. With the current system, it is easy to find problems in the implementation by searching for log entries with severity error. Also logging each HTTP request

<sup>290</sup><https://istio.io/docs/tasks/telemetry/fluentd/>

<sup>291</sup><https://www.fluentd.org/>

<sup>292</sup><https://logback.qos.ch/>

<sup>293</sup><https://github.com/internetitem/logback-elasticsearch-appender>

adds little value. Information about the HTTP requests is probably already included in the logs of the microservices. This is most likely the case for systems in production environments, too.

Generally speaking, Istio's logging support has the advantage that developers do not have to care about these logs at all. Also, the logs are uniform no matter what kind of technology is used in the microservices and how they log. Enforcing a common logging approach and logging format takes some effort. This is particularly true for microservices that use different technologies. So although Istio's logs might not include information from inside the microservices, they are easy to get. Such a log might be better than no log at all or no uniform log.

## 23.7 Resilience

Resilience means that a microservice should not fail if other microservices fail. It is important to avoid failure cascades that could bring down the complete microservices system.

### Measuring Resilience with Istio

Failure cascades can happen if a called microservice returns an error. It could be even worse if the called microservices does return successfully but takes a long time. In that case, resources such as threads might be blocked while waiting for a reply. In the worst case, all threads end up blocked and the calling microservice fails.

Such scenarios are hard to simulate. Usually the network is reasonably reliable. It would be possible to implement a stub microservice that returns errors, but that would require some effort.

However, Istio controls the network communication through the proxies. It is therefore possible to add delays and errors to specific microservices. Then the other microservices can be checked to see if they are resilient against the delays and failures.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: order-fault
spec:
  hosts:
    - order.default.svc.cluster.local
  http:
    - fault:
        abort:
          percent: 100
          httpStatus: 500
  route:
    - destination:
        host: order.default.svc.cluster.local
```

The previous listing shows the content of the file `fault-injection.yaml` from the example. It makes 100 percent of the calls to the order microservice fail with HTTP status 500. You can apply it to the example with `kubectl apply -f fault-injection.yaml` and remove it again with `kubectl delete -f fault-injection.yaml`.

Actually, the microservice will still work after applying the fault injection. If you add a new order to the system, though, it will not be propagated to shipping and invoicing. You can make those microservices poll the order microservice by pressing the pull button in the web UI of shipping and invoicing. In that case, an error will be shown. So the system is already quite resilient because it uses asynchronous communication. The shipping and invoicing microservices still work. The only exception is the polling of the order microservice. If the shipping microservice would call the order microservice synchronously – for example, to fulfill a request –, the shipping service would fail after the fault injection if no additional logic is implemented to handle such a failure.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: order-delay
spec:
  hosts:
    - order.default.svc.cluster.local
  http:
    - fault:
        delay:
          fixedDelay: 7s
          percent: 100
    route:
      - destination:
          host: order.default.svc.cluster.local
```

Another possibility would be to inject a delay as the previous listing shows. You can apply it to the system with `kubectl apply -f delay-injection.yaml` and remove it again with `kubectl delete -f delay-injection.yaml`. If you make the shipping microservice poll the order microservice, it will take longer but it will work fine. Otherwise the system just works normally. So, asynchronous communication solves the resilience problem for delays also.

It is also possible to limit delays and faults, for example, by using HTTP headers. So you can run specific test requests in the production system and make them fail or hit a delay to understand how the production system would handle such problems. The normal requests would be unaffected.

## Implementing Resilience with Istio

Section 14.5 has described several resilience patterns like timeout or circuit breaker. They can be implemented with a library like Hystrix as described in that section. However, this would require changes to the code and can limit the free choice of technologies.

Because Istio adds proxies to the communication between the microservices, you can add a circuit breaker without changing the code.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: order-circuit-breaker
spec:
  host: order.default.svc.cluster.local
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 1
      http:
        http1MaxPendingRequests: 1
        http2MaxRequests: 1
        maxRequestsPerConnection: 1
    outlierDetection:
      consecutiveErrors: 1
      interval: 1m
      baseEjectionTime: 10m
      maxEjectionPercent: 100
```

The previous listing shows a configuration of a circuit breaker for Istio, which is available in the file `circuit-breaker.yaml` and can be added to the example using `kubectl apply -f circuit-breaker.yaml`. You can remove it with `kubectl delete -f circuit-breaker.yaml`. The configuration has the following settings:

- A maximum of one TCP connection is allowed for the service (`maxConnections`).
- There may be just one request per connection (`maxRequestsPerConnection`).
- In total, just one HTTP 1.1 (`http1MaxPendingRequests`) and one HTTP 2 (`http2MaxRequests`) request might be pending.
- Each minute, each microservice instance is checked (`interval`). If it has returned one error (`consecutiveErrors`) – that an HTTP status 5xx or a timeout – it is excluded from traffic for ten minutes (`baseEjectionTime`). All instances of the microservice might be excluded from traffic in this way (`maxEjectionPercent`).

The goal of the circuit breaker is to protect microservices from too much load. That is the reason for limiting the maximum number of connections. Also, the number of pending requests is limited. This protects a microservice too slow to handle all traffic from overload. And, if an instance has already failed, it is excluded from the work. That gives the instance a chance to recover.

The limits in the example are very low to make it easy to trigger the circuit breaker. In a production environment, the values should be higher. If you use the `load.sh` script to access the order microservice's web UI, you will just need to run a few instances in parallel to receive 5xx error codes returned by the circuit breaker.

If the circuit breaker does not accept a request because of the defined limits, the calling microservice will receive an error. So the calling microservice is not protected from a failing microservice. Quite the contrary: To protect the called microservice instances, the circuit breaker might increase the number of failed requests.

## Retry and Timeout

If a called microservice fails, the calling microservice should not fail. Istio provides two measures ensure this:

- *Retries* repeat the failed requests. If the failure is transient, this can make the request succeed. However, it also adds load to the called microservices. A circuit breaker might be useful to protect it from overload. It is possible to define when a request is considered failed. For example, an HTTP status 5xx might be returned.,
- *Timeouts* make sure that the calling microservice is not blocked for too long. Otherwise, if all threads are blocked, the calling microservice might not be able to accept any more requests.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: order-retry
spec:
  hosts:
    - order.default.svc.cluster.local
  http:
    - retries:
        attempts: 20
        perTryTimeout: 5s
        retryOn: connect-failure,5xx
      timeout: 10s
    route:
      - destination:
          host: order.default.svc.cluster.local
```

The previous listing shows a part of the file `retry.yaml`. It configures retries and timeouts for the order microservice. Calls to the order microservice are retried up to 20 times. For each retry, a timeout of 5 seconds is configured. However, there is also a total timeout of 10 seconds. So if the retries don't succeed after 10 seconds, the call will fail. The Istio's default timeout is 15 seconds.

`retryOn` define when a request is considered failed. In this case any HTTP status code 5xx, or connection failures such as timeouts are considered failed requests.

The rest of the file is not shown in the listing. It is very similar and adds retries to the Istio gateway for the order microservice. The Istio gateway routes user requests from outside the Kubernetes cluster to the correct microservice.

You can configure the order microservice to fail for 50 percent of all requests with `kubectl apply -f failing-order-service.yaml`. This is implemented in the code and triggered by the instance variable `FAILRANDOMLY`. It is set to `true` in the Kubernetes configuration. If you access the web page of the order microservice or make one of the other microservices poll the latest orders, you will notice that about half of the time the requests won't work.

You can configure Istio to retry the requests with `kubectl apply -f retry.yaml`. The system should behave as if the microservice just works normally. The retries are transparent. Another possibility is to change the settings for retries and timeout `per request`<sup>294</sup>.

You can remove the retries with `kubectl delete -f retry.yaml`. The failing microservice can be set to normal with `kubectl apply -f microservices.yaml`.

## Resilience: Impact on the Code

Istio's circuit breaker makes it more likely that a call to a microservice fails. The same is true for a timeout. Retries can reduce the number of failures. But even with Istio's resilience features, there will still be calls to microservices that fail. So while retry, timeout, and circuit breaker do not require any changes to the code, the code still needs to take care of failed requests. Those result in a response with HTTP status code 5xx. How failures are handled is a part of the domain logic. For example, if the warehouse management is down, it might not be possible to determine the availability of certain items. Whether an order should be accepted – even though the availability of the ordered items is unknown –, is a business decision. Maybe it is fine to accept the order and deal with the unavailable items. Maybe this would disappoint customers or violate contracts and is therefore not an option. So there must be code that handles an HTTP status code 5xx and determines whether the order should still be processed.

## 23.8 Challenges

Istio has a lot of advantages but also poses a few challenges.

### Complex Infrastructure

Istio is a huge complex system. It adds a lot of functionalities to Kubernetes. It covers monitoring, tracing, logging, security, and many more features. So it adds another set of technologies to the stack that might be hard to understand in every detail. However, Istio provides a complete solution

<sup>294</sup><https://istio.io/docs/concepts/traffic-management/#fine-tuning>

for many challenges of microservices systems. So, although Istio might appear to be complex, the question is whether there are any simpler alternatives with the same set of features. If you run on Kubernetes and end up building an Istio-like infrastructure yourself, you might be building an even more complex solution than that you need to maintain. It is possible to strip down the Istio installation to the [bare minimum](#),<sup>295</sup> or to exclude specific features that might not be needed.

## Mixer and Proxy: Increased Latency

The call to the proxy adds to the latency. The call goes through the loopback device, so it is not a true network call. It just talks to the local machine with a virtual network round trip.

For each call, Istio also adds a synchronous call to Mixer. This is necessary to check the latest version of the policies against the call. The call to Mixer adds to the latency. However, for a system with asynchronous communication like in the example, this is not a huge problem. Asynchronous calls don't need to be handled in a timely manner because no one waits for the results. For synchronous systems, the latency might be a problem even without the additional call to Mixer.

However, the creators of Istio are well aware of this problem. Therefore, caching and other measures are implemented to make sure that the call to Mixer is fast enough. For example, the metrics are transferred asynchronously to limit how much data is transferred synchronously.

The trace in [figure 23-6](#) includes the round trip to Mixer. In that case it is negligible. However, that might be different under high load. Mixer's design is the result of [experiences at Google](#)<sup>296</sup>. Mixer should actually decrease latency and increase availability compared to a service mesh without Mixer. That is because Mixer is designed with high availability and low latency in mind. It shields the proxies from dealing with the rest of the service mesh. However, a solution based on a library such as Hystrix does not add any latency or availability at all. So from a performance perspective, a library might be better.

## Focus on REST

Istio is only useful if most of the communication goes through the proxies. This is the case for TCP communication. However, Istio has specific support for HTTP, gRPC, and WebSockets. For other traffic, it cannot parse the protocol. Therefore, the proxy cannot determine whether a request was handled successfully or not. As the example in this chapter shows, some features are very useful if the system is designed to do asynchronous communication via REST. For example, tracing and the support for resilience are not that big an advantage for any asynchronous systems. There are no huge call trees to be traced. Resilience is already covered by the asynchronous communication.

If the communication relies on a messaging technology like Kafka (see [chapter 11](#)), a lot of communication happens through the messaging system. So the Istio proxies cannot understand it and handle it as well as REST calls. Istio might still provide a few advantages – for example, for web traffic from users. Prometheus and Grafana can show other metrics – for example, from the messaging system. But the advantage is not as large as for a purely REST-based system.

<sup>295</sup><https://istio.io/docs/setup/kubernetes/minimal-install/>

<sup>296</sup><https://istio.io/blog/2017/mixer-spof-myth/>

## Limited Information

The service mesh uses proxies to collect information about the network traffic between the microservices. This enables the service mesh to work independently from the technologies inside the pods. However, it also limits the information available to the service mesh. For monitoring, the information about the requests and the Kubernetes infrastructure might be enough. For tracing, the microservices have to forward some HTTP headers. So tracing is not fully transparent for the microservices. For logging, Istio can just generate a log entry for each HTTP request. For resilience, circuit breaker, timeout, and retries leave not a lot to be desired. Still, the business logic must decide what should happen if a service is not available.

Of course, it is possible to make the microservice log more information or provide more metrics. But in that case, the microservices would need to be modified. So if you don't want to change the microservices, Istio provides limited information that might be enough for some areas such as monitoring, but might lack in other areas such as logging.

## 23.9 Benefits

Istio solves a lot of challenges of microservices architectures. It covers resilience, tracing, monitoring, security, and logging. All of these challenges are solved independently from the programming language or frameworks. So Istio does not limit the technology freedom. This is clearly preferable over the other alternatives presented in this book. Those alternatives limit the choice of technology or at least have to be supported differently for each used programming language.

Also, Istio works well with Kubernetes (see [chapter 17](#)), which complements Istio by solving deployment, clustering, load balancing, and service discovery. Compared to a Kubernetes cluster without Istio, not a lot changes as far as deployment and configuration of the microservices is involved.

Istio's architecture and implementation is based on Google's experience from operating its huge infrastructure. So although it is a relatively new project, the concepts and previous implementations have proven to be successful in a very large and complex environment. Also, Istio uses quite mature technologies such as Prometheus, Grafana, and so on.

Service meshes are gaining a lot of popularity. Istio is probably the most popular implementation. A popular technology is usually one that involves little risk because the community presents a huge market and will always aim at keeping the technology alive and viable. Istio is supported by Google and IBM, two of the leading companies in the IT area.

## 23.10 Variations

There are alternatives to Istio on Kubernetes:

- Istio can also run with [Docker and Consul](#)<sup>297</sup>. So even if you are not using Kubernetes, you can still use Istio's features. See [chapter 15](#) for more information about Consul.
- [Linkerd](#)<sup>298</sup> is another service mesh. It runs on Kubernetes and is part of the portfolio of the [Cloud Native Computing Platform](#)<sup>299</sup>.
- [Aspen Mesh](#)<sup>300</sup> is a distribution of Istio.
- Cloud providers also offer service meshes. Azure has the [Service Fabric Mesh](#)<sup>301</sup> and Amazon Web Services the [AWS App Mesh](#)<sup>302</sup>. Google provides support for [Istio in the Google Cloud](#)<sup>303</sup>.

## 23.11 Experiments

Istio provides extensive documentation. You can use it to familiarize yourself with features of Istio that this chapter does not discuss:

- Istio provides support for security. See [the security task](#)<sup>304</sup> for some hands-on exercises for this feature. The exercises cover encrypted communication, authentication and authorization.
- Istio also supports advanced routing. For example, [the traffic shifting task](#)<sup>305</sup> shows how to do A/B testing with Istio. The [mirroring task](#)<sup>306</sup> shows mirroring. With mirroring, two versions of the microservice receive the traffic. Mirroring can be used to make sure that the new and the old version behave in the same way.
- Istio can also create graphs of the [dependencies between the microservices](#)<sup>307</sup>. Istio even provides a specific tool to visualize the service mesh called Kiali, see the [Kiali task](#)<sup>308</sup>.

## 23.12 Conclusion

Service meshes solve many challenges for microservices systems. They support monitoring, tracing, logging, and resilience. Although they provide many features, service meshes do not limit the free technology choice for implementing microservices. As the example in this chapter shows, this works quite well for monitoring. For tracing the microservices need to be modified. For logging, Istio is of little use. Besides providing the information, Istio also includes tools like Prometheus, Grafana, Jaeger, or Kiali to visualize and store information about the microservices system. So Istio is a pretty complete solution for the operation of microservices.

<sup>297</sup><https://istio.io/docs/setup/consul/>

<sup>298</sup><https://linkerd.io/>

<sup>299</sup><https://cncf.io/>

<sup>300</sup><https://aspenmesh.io/>

<sup>301</sup><https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-mesh-overview>

<sup>302</sup><https://aws.amazon.com/app-mesh/>

<sup>303</sup><https://cloud.google.com/istio/>

<sup>304</sup><https://istio.io/docs/tasks/security/>

<sup>305</sup><https://istio.io/docs/tasks/traffic-management/traffic-shifting/>

<sup>306</sup><https://istio.io/docs/tasks/traffic-management/mirroring/>

<sup>307</sup><https://istio.io/docs/tasks/telemetry/servicegraph/>

<sup>308</sup><https://istio.io/docs/tasks/telemetry/kiali/>

Istio's concepts are based on Google's experience with large container systems, and so the idea of a service mesh has proven itself. Istio integrates most of the technologies and only adds some glue, making it a not very risky technology. The components Istio uses are mature and used in many systems.

However, Istio is not just powerful but also complex. To solve this, Istio can be stripped down. It is possible to exclude each part of the system from the installation. The question is whether any simpler alternative exists. The challenges Istio solves must be dealt with in a microservices system. Istio's complexity is probably not a problem of Istio but rather of the microservices themselves.

Istio adds an overhead to the communication between microservices. That might impact performance but is probably acceptable. Microservices communicate through the network anyway. Adding an additional, relatively small overhead to that communication is not a significant change.

## Advantages

- Complete solution for the main challenges of microservices
- No impact on the technology used by the microservice
- No changes to the code of the microservices necessary
- Concept has proved itself at Google
- Good integration with Kubernetes

## Challenges

- Huge and complex
- Adds overhead to the communication between the microservices

# 24 And Now What?

This book showed different concepts and recipes for different technologies. A section on “variations” was included in each chapter. From the concepts, recipes, and their variations, you now have to create your own menu.

Here are some tips:

- *Adapting* and *combining* the recipes is a must. Each project is different and needs its own technology solution.
- The most interesting thing about the selected technology stack is the *reason for the choice*. No one can relieve the project of the responsibility for this technology choice. That’s why it must be well considered in terms of the reasoning given for the decision.
- The introduction of microservices is an architectural change. It is often accompanied by organizational changes and the introduction of many new technologies. No project can cope with *too many changes at once*. Therefore, as few new technologies as possible should be introduced simultaneously, with simple technologies preferred. Finally, the introduction of technologies can take place gradually. The complete technology stack may not yet be necessary for the first microservice.
- The *migration* of an existing system has a lot of influence on the decisions made when implementing the microservices architecture. Sticking with technologies and architectures already used in the old system can simplify migration.
- The *skills* of the team members involved are also an important influencing factor. If no one has experience with a technology, that’s a risk.
- The examples from this book facilitate familiarization with the technologies. However, for a production environment, factors such as *fail-safety*, *load balancing*, and *clusters* must be considered.

The overview of the recipes has hopefully made it easier to get started in the world of microservices – and maybe it was fun, too. I wish you every success with microservices!

# Appendix A: Installation of the Environment

- The source code of the examples is available on Github. For access, version control *git* must be installed; see <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>. If the installation was successful, a call of *git* in the command prompt will work.
- The examples are implemented in Java. Therefore, *Java* has to be installed. Instructions can be found at [https://www.java.com/en/download/help/download\\_options.xml](https://www.java.com/en/download/help/download_options.xml). Because the examples have to be compiled, a JDK (Java Development Kit) has to be installed. The JRE (Java Runtime Environment) is not enough. When the installation is completed, it should be possible to start *java* and *javac* in the command prompt.
- The examples run in Docker containers. This requires an installation of *Docker Community Edition*; see <https://www.docker.com/community-edition/>. Docker can be called with *docker*. This should work without errors after the installation.
- The examples require a lot of memory in some cases. Therefore, *Docker* should have about 4 GB available. Otherwise, Docker containers could terminate due to lack of memory. Under Windows and macOS, you can find the settings for this in the Docker application under Preferences/Advanced. If there is not enough memory, Docker containers are terminated. This is shown by the entry *killed* in the logs of the containers.
- After installing Docker you should be able to call *docker-compose*. If *Docker Compose* cannot be invoked, a separate installation is necessary; see <https://docs.docker.com/compose/install/>.

Details regarding Docker are presented in [chapter 5](#).

# Appendix B: Maven Commands

Maven is a build tool. The configuration for a project is stored in a `pom.xml` file. [Section 5.3](#) contains a listing of such a file for a Spring Boot project. <http://start.spring.io/> offers a simple possibility for generating new Spring Boot projects with suitable `pom.xml` files. To do so, the user just has to enter some settings on the web page. Then the web page creates the project with a `pom.xml`.

Maven can combine multiple projects to a [multi module project<sup>309</sup>](#). In this case, the definitions meant to apply to all modules are stored in a single `pom.xml`. All modules reference this `pom.xml`.

The `pom.xml` is stored in a directory. The modules are saved in a sub directory. They have their own `pom.xml` with the information specific to the respective module.

On the one hand, you can start Maven for the entire project in the directory containing the `pom.xml`. In this case, Maven builds the entire project with all its modules. On the other hand, you can start Maven in the directory for a specific module. Then the Maven commands relate to this one module.

## Directories

A Maven module has a fixed file structure.

- The directory `main` contains all files of the module.
- The directory `test` comprises files which are needed only for tests.

Beneath these directories is likewise a standardized directory structure.

- `java` contains the Java code.
- `resources` contains resources adopted into the application.

## Maven Wrapper

After the [installation<sup>310</sup>](#), you can use Maven by starting `mvn`. The rest of this appendix assumes such a Maven installation.

Instead of installing Maven, you can use the [Maven Wrapper<sup>311</sup>](#). In that case, a script is created that downloads and installs Maven. Then `./mvnw` (Linux, macOS) or `./mvnw.cmd` (Windows) must be used to execute Maven. All examples for the book include a Maven wrapper, so this approach can be used, too.

<sup>309</sup><https://maven.apache.org/guides/mini/guide-multiple-modules.html>

<sup>310</sup><https://maven.apache.org/install.html>

<sup>311</sup><https://github.com/takari/maven-wrapper>

## Commands

The most important commands for Maven are:

- `mvn package` downloads all dependencies from the Internet, compiles the code, executes the tests, and creates an executable JAR file. The result is provided in the sub directory `target` of the respective module. `mvn package -Dmaven.test.skip=true` does not execute the tests. `mvn package -DdownloadSources=true -DdownloadJavadocs=true` downloads the source code and the JavaDoc of the dependent libraries from the Internet. The JavaDoc contains a description of the API. Development environments can display JavaDoc and the library source code for the user.
- `mvn test` compiles and tests the code, but does not create a JAR.
- `mvn install` adds a step to `mvn package` by copying the JAR files into the local repository in the `.m2` directory in the home directory of the user. This allows other projects and modules to declare the module as a dependency in `pom.xml`. However, this is not necessary for the examples, so that `mvn package` is enough.
- `mvn clean` deletes all results of preceding builds. Maven commands can be combined. `mvn clean package` thus compiles everything anew because the results of the old builds have been deleted prior to the new build.

The result of the Maven build is a JAR (Java Archive). The JAR contains all components of the application including all the libraries. Java directly supports this file format. Therefore, it is possible to start a microservice with `java -jar target/microservice-order-0.0.1-SNAPSHOT.jar`.

## Troubleshooting

When `mvn package` does not work:

- Try out `mvn clean package` to delete all old build results prior to the new build.
- Use `mvn clean package package -Dmaven.test.skip=true` to skip the tests.
- The tests might fail because a server is still running on your machine on port 8080. Make sure this is not the case.

# Appendix C: Docker and Docker Compose Commands

Docker Compose is used for the coordination of multiple Docker containers. Microservices systems usually consist of many Docker containers. Therefore, it makes sense to start and stop the containers with Docker Compose.

## Docker Compose

Docker Compose uses the file `docker-compose.yml` to store information about the containers. The [Docker documentation](#)<sup>312</sup> explains the structure of this file. [Section 4.6](#) contains an example of a Docker Compose file.

Upon starting, `docker-compose` outputs all possible commands. The most important commands for Docker Compose are:

- `docker-compose build` generates the Docker images for the containers with the help of the Dockerfiles referenced in `docker-compose.yml`.
- `docker-compose pull` downloads the Docker images referenced in `docker-compose.yml` from Docker hub.
- `docker-compose up -d` starts the Docker containers in the background. Without `-d`, the containers start in the foreground so that the output of all Docker containers happens on the console. It is not particularly clear which output originates from which Docker container. The option `--scale` can start multiple instances of a service – for example, `docker-compose up -d --scale order=2` starts two instances of the `order` service. The default value is one instance.
- `docker-compose down` stops and deletes the containers. In addition, the network and the Docker file systems are deleted.
- `docker-compose stop` stops the containers. Network, file systems, and containers are not deleted.

## Docker

At startup, `docker` outputs all valid commands without parameters.

Tip: Tab-pressing completes names and IDs of containers and images.

Here is an overview of the most important commands. The container `ms_catalog_1` is used as an example.

---

<sup>312</sup><https://docs.docker.com/compose/compose-file/>

## State of a Container

- `docker ps` displays all running Docker containers. `docker ps -a` also shows stopped Docker containers. The containers as well as the images have a hexadecimal ID and a name. `docker ps` outputs all this information. For other commands, containers can be identified by name or hexadecimal ID. For the example in this book, the containers have names like `ms_catalog_1`. This name consists of a prefix `ms` for the project, the name of the service `catalog`, and the sequence number `1`. The name of the container is often confused with the name of the image (for example, `ms_catalog`).
- `docker logs ms_catalog_1` shows the previous output of the container `ms_catalog_1`. `docker logs -f ms_catalog_1` also displays all other outputs that the container still outputs.

## Lifecycle of a Container

- `docker run ms_catalog --name="container_name"` starts a new container with the image `ms_catalog`, which gets the name `container_name`. The parameter `--name` is optional. The container then executes the command stored in the `CMD` entry of the `Dockerfile`. But you can execute a command in a container with `docker run <image> <command>`. `docker run ewolff/docker-java /bin/ls` executes the command `/bin/ls` in a container with the Docker image `ewolff/docker-java`. So the command displays the files in the root directory of the container. If the image does not yet exist locally, it is automatically downloaded from the Docker hub on the Internet. When the command has been executed, the container shuts itself down.
- `docker exec ms_catalog_1 /bin/ls` executes `/bin/ls` in the running container `ms_catalog_1`. Thus, with these commands, you can start tools in an already running container. `docker exec -it ms_catalog_1 /bin/sh` starts a shell and redirects input and output to the current terminal. This way, you have a shell in the Docker container and can interactively work with the container.
- `docker stop ms_catalog_1` stops the container. It first sends a `SIGTERM` so that the container can shut down cleanly, and then a `SIGKILL`.
- `docker kill ms_catalog_1` terminates execution of the container with a `SIGKILL`. But the container is still there.
- `docker rm ms_catalog_1` permanently deletes the container.
- `docker start ms_catalog_1` restarts the previously stopped container. Because the data is not deleted when the container was stopped, all data is still available.
- `docker restart ms_catalog_1` restarts the container.

## Docker Images

- `docker images` displays all Docker images. The images have a hexadecimal ID and a name. For other commands, images can be identified by both mechanisms.

- `docker build -t=<name> <path>` creates an image with the name `name`. The `Dockerfile` has to be stored in directory `path`. When no version is indicated, the image gets the version `latest`. As an alternative, the version can also be indicated in the format `-t=<name:version>`. [Section 4.5](#) describes the format of the `Dockerfiles`.
- `docker history <image>` shows the layers of an image. For each layer, the ID, the executed command, and the size of the layer appear. The image to be displayed can be identified by its name if only one version of the image with that name exists. Otherwise, the name and version must be specified via `name:version`. Of course, it is also possible to use the hexadecimal ID of the image.
- `docker rmi <image>` deletes an image. As long as a container is still using the image, it cannot be deleted.
- `docker push` and `docker pull` store Docker images in a registry or load them from a registry. If no other registry is configured, the public Docker hub is used.

## Cleaning up

Several commands are available to clean up the Docker environment .

- `docker container prune` deletes all stopped containers.
- `docker image prune` deletes all images that do not have a name.
- `docker network prune` deletes all unused Docker networks.
- `docker volume prune` deletes all Docker volumes not used by a Docker container.
- `docker system prune -a` deletes all stopped containers, all unused networks and all images not used by at least one container. So all that remains is what the currently running containers need.

## Troubleshooting

If an example does not work:

- Are all containers running? `docker ps` displays the running containers, and `docker ps -a` shows the running and the terminated ones.
- You can display logs with `docker logs`. This works also for terminated containers. The term `Killed` in the logs denotes that too little memory is available. Under Windows and macOS, you can find the settings for this in the Docker application under Preferences/ Advanced. Docker should have about 4 GB assigned.
- In case of more complex problems, you can start a shell in the container with `docker exec -it ms_catalog_1 /bin/sh` and examine the container more closely.