Microservices

Flexible Software Architectures

Eberhard Wolff

Microservices

Flexible Software Architectures

Eberhard Wolff

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

This version was published on 2016-01-10

* * * * *

* * * * *

Table of Contents

1 Preface

Even though Microservices are a new term, they have haunted me already for a long time. In 2006 Werner Vogels (CTO, Amazon) gave a talk at the JAOO conference presenting the Amazon Cloud and Amazon's partner model. In his talk he mentioned the CAP theorem, today the basis for NoSQL. In addition he was talking about small teams, which develop and run services with their own databases. This type of organization is nowadays called DevOps, and the architecture is known as Microservices.

Later I was asked to develop a strategy for a client allowing him to integrate modern technologies into his existing application. After a few attempts to integrate the new technologies directly into the Legacy code, we finally built a new application with a completely different modern technology stack alongside the old one. The old and the new application were only coupled via HTML links and via a shared database. Except for the shared database this is in essence a Microservices approach. That happened in 2008.

Already in 2009 another client had divided his complete infrastructure into REST services, which were each developed by individual teams. This is also called Microservices today. Many other companies with a web-based business model had already implemented similar architectures at the time. Lately I realized in addition that Continuous Delivery influences the software architecture. Also there Microservices offer many advantages.

This is the reason for writing this book: Microservices constitute an approach a number of people have already been pursuing for a long time, among them many very experienced architects. Like every other approach to architecture Microservices for sure cannot solve every problem, however this concept represents an interesting alternative to existing approaches.

1.1 Overview of Microservice

Microservice: Preliminary Definition

The focus of this book are Microservices – an approach for the modularization of software. Modularization in itself is nothing new. For quite some time large

systems have been divided into small modules to facilitate the implementation, understanding and further development of software.

The new aspect is that Microservices use modules, which run as distinct processes. This approach is based on the philosophy of UNIX, which can be reduced to three aspects:

- One program should only fulfill one task, but this it should do really well.
- Programs should be able to work together.
- A universal interface should be used. In UNIX this is provided by text streams.

The term Microservice is not firmly defined. Chapter 4 provides a more detailed definition. However, the following criteria can serve as first approximation:

- Microservices are a modularization concept. Their purpose is to divide large software systems into smaller parts. Thus they influence the organization and development of software systems.
- Microservices can be deployed independently of each other. Changes to one Microservice can be taken into production independently of changes to other Microservices.
- Microservices can be implemented in different technologies. There is no restriction on the programming language or the platform for each Microservice.
- Microservices possess their own data storage: a private database – or a completely separate schema in a shared database.
- Microservices can bring their own support services along, for example a search engine or a specific database. Of course, there is a common platform for all Microservices – for example virtual machines.
- Microservices are self-contained processes – or virtual machines e.g. to bring the supporting services along.
- Accordingly, Microservices have to communicate via the network. To do so Microservices use protocols, which support loose coupling such as REST or messaging.

Deployment Monoliths

Microservices are the opposite of Deployment Monoliths. A Deployment Monolith is a large software system, which can only be deployed in one piece. It has to get in its entirety through all phases of the Continuous Delivery pipeline

such as deployment, the test stages and release. Due to the size of Deployment Monoliths these processes take longer than for smaller systems. This reduces flexibility and increases process costs. Deployment Monolith can have a modular structure internally – still, all modules have to be brought into production simultaneously.

1.2 Why Microservices

Microservices allow to divide software into modules and thereby improve the software changeability.

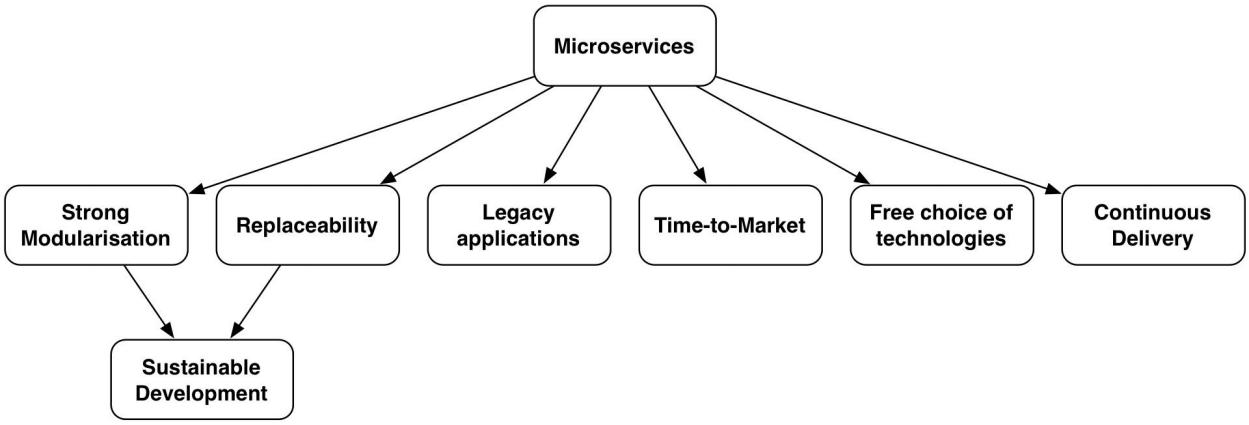


Fig. 1: Advantages of Microservices

Microservices offer a number of important advantages:

Strong Modularization

Microservices are a strong modularization concept. Whenever a system is built from different software components such as Ruby GEMs, Java JARs, .NET Assemblies or Node.js NPMs, unwished for dependencies can easily creep in. Somebody references a class or function in a place where it is not supposed to be used. After a short while so many dependencies will have accumulated that the system can no longer be serviced or further developed.

Microservices, in contrast, communicate via explicit interfaces, which are realized using mechanisms like messages or REST. Accordingly, the technical hurdles for the use of Microservices are higher. Thus unwanted dependencies can hardly arise. In principle it should be possible to achieve also a high level of modularization in Deployment Monoliths. However, practical experience teaches that the architecture of Deployment Monoliths progressively deteriorates over time.

Easy Replaceability

Microservices can more easily be replaced. Other components utilize a Microservice via an explicit interface. Whenever a service offers the same interface, it can replace the Microservice. The new Microservice does neither need to use a part of the code basis nor the technologies of the old Microservice. Such like restrictions often prevent the modularization of legacy systems.

Small Microservices further facilitate replacements. Such replacements are often neglected during the development of software systems. Who likes to take into consideration how the just built system can be replaced in the future? The easy replaceability of Microservices reduces in addition the costs of incorrect decisions. When the decision for a technology or approach is limited to a Microservice, this Microservice can be completely rewritten if need arises.

Sustainable Development

The strong modularization and the easy replaceability allow for sustainable software development. Most of the time working on a new project is quite simple. Upon longer project run time productivity decreases. One of the reasons is the erosion of architecture. Microservices counteract this erosion due to the strong modularization. Being bound to outdated technologies and the difficulties associated with the removal of old system modules constitute additional problems. Microservices, which are not linked to a specific technology and can by replaced one by one, overcome these problems.

Further Development of Legacy Applications

Starting with a Microservices architecture is easy and provides immediate advantages when working with old systems: Instead of having to add to the old and hard to understand code base the system can be enhanced with a Microservice. The Microservice can act on specific requests while leaving all others to the legacy system. It can modify requests prior to their processing by the legacy system. In this manner replacing the complete functionality of the legacy system can be circumvented. In addition, the Microservice is not bound to the technology stack of the legacy system, but can be developed using modern approaches.

Time-to-Market

Microservices allow for a better time-to-market. As mentioned before, Microservices can be brought into production on a one-by-one basis. If teams

working on a large system are responsible for one or several Microservices and if features require only changes to these Microservices, each team can develop and bring features into production without time consuming coordination with other teams. In this manner many teams can work on numerous features in parallel and bring more features into production within a certain time than would have been possible with a Deployment Monolith. Microservices help scaling agile processes to large teams by dividing the large team into small teams each dealing with their own Microservices.

Independent Scaling

Each Microservice can be scaled independently of other services. This obliterates the need to scale the whole system when it is only a few functionalities that are used intensely. This will often be a decisive simplification.

Free Choice of Technologies

When developing Microservices there are no restrictions in regards to the usage of technologies. This allows to test a new technology within a single Microservice without affecting other services. Thereby the risk associated with the introduction of new technologies and new versions of already used technologies is decreased as these new technologies are introduced and tested in a confined environment keeping potential costs low. In addition it is possible to use specific technologies for specific functionalities, for example a specific database. The risk is small as the Microservice can easily be replaced or removed. The new technology is confined to one or few Microservices. This reduces the potential risk and enables independent technology decisions for different Microservices. Moreover, it facilitates the decision to try out and evaluate new, highly innovative technologies. This increases the productivity of developers and prevents that the technology platform becomes outdated. In addition, the use of modern technologies will attract qualified developers.

Continuous Delivery

Microservices are advantageous for Continuous Delivery. They are small and can be deployed independently of each other. Realizing a Continuous Delivery pipeline is simple due to the size of a Microservice. The deployment of a single Microservice is associated with less risk than the deployment of a large monolith. It is also easier to assure the safe deployment of a Microservice, for instance by running different versions in parallel. For many Microservice users Continuous Delivery is the main reason for the introduction of Microservices.

All these reasons argue for the introduction of Microservices. Which of these reasons are the most important will depend on the context. Scaling agile processes and Continuous Delivery are often crucial from a business perspective. Chapter 5 describes the advantages of Microservices in detail and deals also with prioritization.

Challenges

However, there is no light without shadow. Accordingly Chapter 6 will discuss the challenges posed by the introduction of Microservices and how to deal with them. In short, the main challenges are the following:

Relationships are Hidden.

The architecture of the system consists of the relationships between the services. However, it is not evident which Microservice calls which other Microservice. This makes working on the architecture challenging.

Refactoring is Difficult.

The strong modularization leads also to disadvantages: Refactorings, which move functionalities between Microservices, are difficult to perform. And, once introduced, it is hard to change the Microservices-based modularization of a system. However, these problems can be lessened by smart approaches.

Domain Architecture is Important.

The modularization into Microservices for the different domains is important as it determines how teams are divided. Problems at this level influence also the organization. Only a solid domain architecture can ensure the independent development of a Microservice. As it is difficult to change the once established modularization, mistakes can be hard to correct later on.

Running Microservices is Complex.

A system comprised of Microservices has many components, which have to be deployed, controlled and run. This increases the complexity for operations and the number of runtime infrastructures used by the system. Microservices necessitate the automatization of operations as operating the platform is otherwise too laborious.

Distributed Systems are Complex.

The complexity the developers are facing increases: A Microservice-based system is a distributed system. Calls between Microservices can fail due to network problems. Calls via the network are slower and have a smaller bandwidth than calls within a process.

Part I: Motivation and Basics

This part of the book conveys what Microservices are, why they are interesting and where they are of use. Practical examples demonstrate the effects of Microservices in different scenarios. Chapter 2 explains the structure of the book. To illustrate the importance of Microservices chapter 3 contains detailed scenarios where Microservices can be used.

2 Introduction

This chapter focuses on the book itself: Section 2.1 explains briefly the book concept. Section 2.2 describes the audience for which the book was written. Section 2.3 provides an overview of the different chapters and the structure of the book. Section 2.5 describes paths through the book for different audiences. Section 2.6 finally contains the acknowledgements. Errata, links to examples and additional information can be found at http://microservices-book.com/ . The example code is available at https://github.com/ewolff/microservice/ .

2.1 Overview of the Book

This book provides a detailed introduction to Microservices. Architecture and organization are the main topics. However, technical implementation strategies will not be neglected. A complete example of a Microservice-based system demonstrates a concrete technical implementation. Technologies for Nanoservices illustrates that modularization does not stop with Microservices. The book provides all necessary information in order to enable readers to start using Microservices.

2.2 For Whom is the Book Meant?

The book addresses managers, architects and developers who want to introduce Microservices as an architectural approach.

Managers

Microservices can profit from the mutual support of architecture and organization Microservices offer. In the introduction managers get to know the basic ideas behind Microservices. Afterwards they can focus on the organizational effects of utilizing Microservices.

Developers

Developers are provided with a comprehensive introduction to the technical aspects and can acquire the necessary skills to use Microservices. A detailed example of a technical implementation of Microservices as well as numerous

additional technologies, for example for Nanoservices, facilitate grasping the basic concepts.

Architects

Architects get to know Microservices from an architecture perspective and can at the same time deepen their understanding of the associated technical and organizational issues.

The book highlights possibilities for experiments and additional information sources. So the interested reader can test her/his new knowledge practically and delve deeper into subjects that are of relevance to her/him.

2.3 Chapter Overview

Part I

The first part of the book explains the motivation for using Microservices and the foundation of the Microservices architecture. The preface (chapter 1) already presented the basic properties as well as advantages and disadvantages of Microservices. Chapter 3 presents two scenarios for the use of Microservices: an E-Commerce application and a system for signal processing. This part conveys first insights into Microservices and already points out contexts for applications.

Part II

Part II does not only explain Microservices in detail, but also deals with their advantages and disadvantages:

- Chapter4 investigates the **definition** of the term Microservices" from three perspectives: the size of a Microservice, Conway's Law, which states that organizations can only create specific software architectures, and finally from a technical perspective based on Domain-Driven Design and *Bounded Context*.
- The **reasons** for using Microservices are detailed in chapter 5. Microservices do not only have technical, but also organizational advantages, and also from a business perspective there are good reasons for turning to Microservices.
- The unique **challenges** posed by Microservices are discussed in chapter 6. Among these are technical challenges as well as problems related to architecture, infrastructure and operation.

- Chapter 7 aims at defining the differences between Microservices and **SOA (Service-Oriented Architecture)**. At first sight both concepts seem to be closely related. However, a closer look reveals a plethora of differences.

Part III

Part III deals with the application of Microservices and demonstrates how the advantages that were described in part II can be obtained and how the associated challenges can be solved.

- Chapter 8 describes the **architecture of Microservice-based systems**. In addition to domain architecture comprehensive technical challenges are discussed.
- Chapter 9 presents the different possibilities for the **integration** of and the **communication** between Microservices. This includes not only a communication via REST or messaging, but also an integration of UIs and the replication of data.
- Chapter 10 shows possible **architectures for Microservices** such as CQRS, Event Sourcing or hexagonal architecture. Finally suitable technologies for typical challenges are addressed.
- **Testing** is the main focus of chapter 11. Tests have to be as independent as possible to allow for the independent deployment of the different Microservices. Nevertheless the tests have not only to check the individual Microservices, but also the system in its entirety.
- Operation and **Continuous Delivery** are addressed in chapter 12. Microservices generate a huge number of deployable artefacts and thus increase the demands on the infrastructure. This is a substantial challenge when introducing Microservices.
- Chapter 13 illustrates how Microservices also influence the **organization**. After all, Microservices are an architecture, which is supposed to influence and improve the organization.

Part IV

The last part of the book shows in detail and at the code level how Microservices can be technically implemented:

- Chapter 14 contains an exhaustive example for a Microservices architecture based on Java, Spring Boot, Docker and Spring Cloud. This chapter aims at providing an application, which can be easily run, illustrates the concepts

behind Microservices in practical terms and offers a starting point for the implementation of a Microservices system and experiments.
- Even smaller than Microservices are the Nanoservices, which are presented in chapter 15. Nanoservices exact specific technologies and a number of compromises. The chapter discusses different technologies in the context of their advantages and disadvantages.
- Chapter 16 demonstrates in the end how Microservices can be adopted.

2.4 Essays

The book contains assays, which were written by Microservices experts. The experts were asked to record their main findings regarding Microservices on approximately two pages. Sometimes these assays complement book chapters, sometimes they focus on other topics, and sometimes they contradict passages in the book. This illustrates that there is in general no single right answer when it comes to software architectures, but rather a collection of different opinions and possibilities. The essays offer the unique opportunity to get to know different view points in order to subsequently develop an opinion.

2.5 Paths Through the Book

The book offers suitable content (Tab. 1) for each type of audience. Of course, everybody can and should read also chapters that are primarily meant for people with a different type of job. Nevertheless the chapters are focussing on topics that are most relevant for a certain audience as indicated below.

Tab. 1: Paths through the book

Chapter	Developer	Architect	Manager
3 - Microservice Scenarios	X	X	X
4 - What are Microservices?	X	X	X
5 - Reasons for Using Microservices	X	X	X
6 - Challenges Regarding Microservices	X	X	X
7 - Microservices and SOA		X	X
8 - Architecture of Microservice-based Systems		X	
9 - Integration and Communication	X	X	
10 - Architecture of Individual Microservices	X	X	
11 - Testing Microservices and Microservice-based Systems	X	X	

12 - Operations and Continuous Delivery of Microservices	X	X	
13 - Organizational Effects of a Microservices-based Architecture			X
14 - Example for a Microservice-based Architecture	X		
15 - Technologies for Nanoservices	X	X	
16 - How to start with Microservices?	X	X	X

Readers who only want to obtain an overview are advised to concentrate on the summary section at the end of each chapter. People who want to gain first of all practical knowledge should commence with chapters 14 and 15, which deal with concrete technologies and code.

The instructions for experiments, which are given in the sections “Try and Experiment”, help to deepen the understanding by doing practical exercises. Whenever a chapter is of particular interest for the reader, he/she is encouraged to complete the related exercises to get a better grasp on the topics presented in the respective chapter.

2.6 Acknowledgement

I would like to thank everybody with whom I have discussed Microservices and all the people who asked questions or worked with me - way too many to list them all. The interactions and discussions were very fruitful and fun!

I would like to mention especially Jochen Binder, Matthias Bohlen, Merten Driemeyer, Martin Eigenbrodt, Oliver B. Fischer, Lars Gentsch, Oliver Gierke, Boris Gloger, Alexander Heusingfeld, Christine Koppelt, Andreas Krüger, Tammo van Lessen, Sascha Möllering, André Neubauer, Till Schulte-Coerne, Stefan Tilkov, Kai Tödter, Oliver Wolf and Stefan Zörner.

My employer innoQ has also played an important role throughout the writing process. Many discussions and suggestions of my innoQ colleagues are reflected in the book.

Finally I would like to thank my friends and family and especially my wife whom I have often neglected while working on the book. In addition I would like to thank

her for the English translation of the book.

Of course, my thanks go also to all the people who have been working on the technologies that are mentioned in the book and thus have laid the foundation for the development of Microservices. Special thanks also to the experts who shared their knowledge of and experience with Microservices in the essays.

Leanpub has provided me with the technical infrastructure to create the translation. It has been a pleasure to work with it and it is quite likely that the translation would not exist without Leanpub.

Last but not least I would like to thank dpunkt.verlag and René Schönfeldt who supported me very professionally during the genesis of the original German version.

3 Microservice Scenarios

This chapter will present a number of scenarios in which Microservices can be useful. Section 3.1 focuses on the modernization of a legacy web application. This scenario is the most common context for Microservices. A very different scenario is discussed in section 3.2. In this case a signaling system is supposed to be developed as distributed system based on Microservices. Section 3.3 formulates a conclusion and invites the readers to judge for themselves on the usefulness of Microservices in the presented scenarios.

3.1 Modernizing an E-Commerce Legacy Application

Scenario

The Big Money Online Commerce inc. runs an E-commerce shop, which is the main source of the company revenue. It is a web application offering many different functionalities such as user registration and administration, product search, an overview of orders and the ordering process – the central feature of an E-commerce application.

This application is a Deployment Monolith: It can only be deployed in its entirety. Whenever a feature is changed, the entire application needs to be deployed anew. The E-Commerce shop works together with other systems – for instance with accounting and logistics.

Reasons to Use Microservices

The Deployment Monolith once started out as a well-structured application. However, over the years more and more dependencies between the individual modules creeped in. For this reason the application is nowadays hardly maintainable and changeable. In addition the original architecture is not suited any more for the current requirements. Product search for instance has been greatly modified as the Big Money Online Commerce inc. attempts to outperform its competitors especially in this area. Likewise more and more possibilities have been generated for clients to solve problems by themselves without the assistance of a client service. This helped the company to reduce costs. Accordingly, these

two modules became very large with a very complex internal structure and many dependencies on other modules that had not been planned for originally.

Slow Continuous Delivery Pipeline

Big Money has decided to use Continuous Delivery and has established a Continuous Delivery pipeline. This pipeline is complicated and slow as the complete Deployment Monolith needs to be tested and brought into production. Some of the tests run for hours. A faster pipeline would be highly desirable.

Parallel Work Complicated

There are teams working on different new features. However, the parallel work is complicated: The software structure just doesn't really support it. The individual modules are not well enough separated and have too many interdependencies. As everything can only be deployed together, the entire Deployment Monolith has to be tested. Deployment and testing phase constitute a bottle neck. Whenever a team is having a release in the deployment pipeline, which is creating a problem, all other teams have to wait until the problem has been fixed and the change has been successfully deployed. Moreover, the access to the Continuous Delivery pipeline has to be coordinated. Only one team at a time can be doing testing and deployment. Thus it has to be regulated which team can bring which change into production at which time.

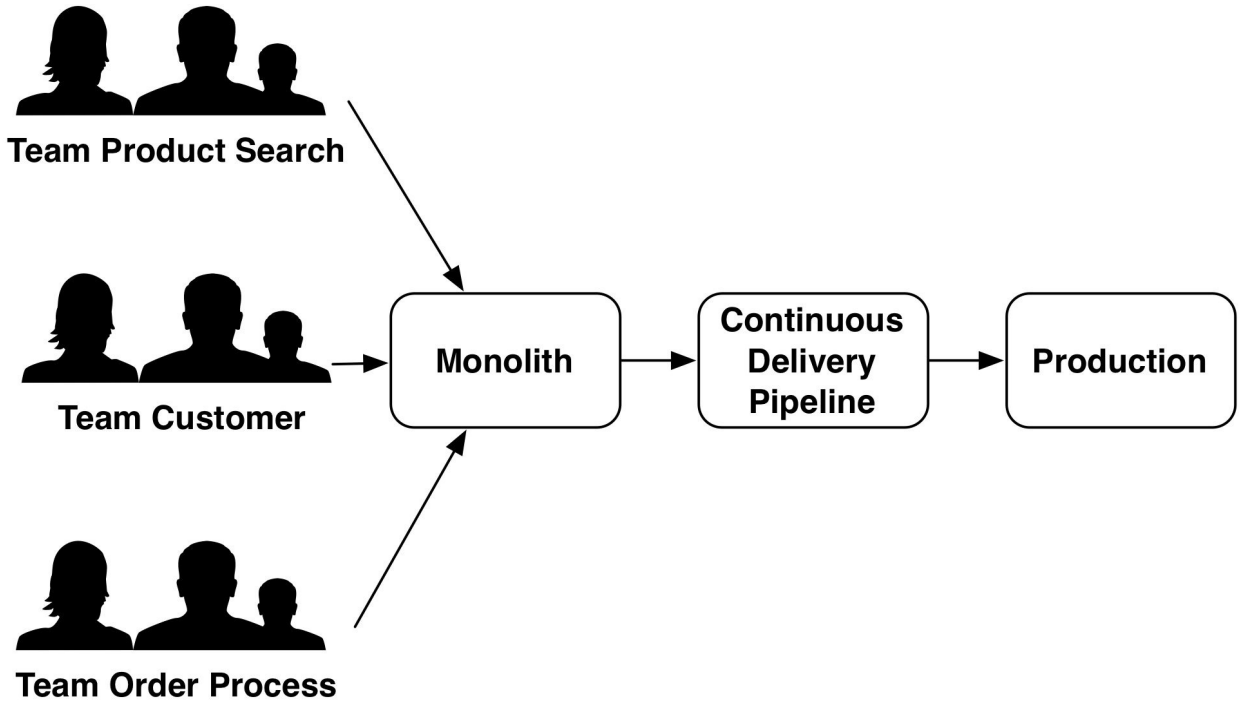
Bottleneck During Testing

In addition to deployment also the tests have to be coordinated. When the Deployment Monolith runs in an integration test, only the changes made by one team are allowed to be contained in the test. There were attempts to test several changes at once. However, in that case it was very hard to discern the origin of errors so that error analyses were long and complex.

One integration test requires approximately one hour. Thus about six integration tests are feasible per working day as errors have to be fixed and the environment has to be set up again for the next test. In the case of ten teams one team can bring one change into production every two days on average. However, often a team also has to do error analysis, which lengthens integration. For that reason some teams use feature branches in order to separate themselves from integration: They perform their changes on a separate branch in the version control system. Integrating these changes into the main branch later on often causes problems: Changes are erroneously removed again upon merging or the software suddenly contains errors, which are caused by the separated development process and only

show up after integration. These errors can only be eliminated in lengthy processes after integration.

Consequently, the teams slow each other down due to the testing. Although each team develops its own modules, they all work on the same code basis so that they impede each other. As a consequence of the shared Continuous Delivery pipeline and the ensuing need for coordination the teams are neither able to work independently of each other nor in parallel.



....

Fig. 2: Teams slow each other down due to the Deployment Monoliths.

Approach

Because of the many problems Big Money Online Commerce inc. decided to split off small Microservices from the Deployment Monolith. The Microservices each implement one feature such as the product search and are developed by individual teams. Each team has the complete responsibility for an individual Microservice starting from requirements engineering up to running the application in production. The Microservices communicate with the Monolith and other Microservices via REST. The client GUI is also divided between the individual Microservices based on use cases. Each Microservice delivers the HTML pages for its use cases. Links are allowed between the HTML pages of the Microservices. However, it is not allowed to access the database tables of the other Microservices or the Deployment Monolith. Integration of services is exclusively done via REST or via links between the HTML pages.

The Microservices can be deployed independently of each other. This allows to deliver changes in a Microservice without the need to coordinate with other Microservices or teams. This greatly facilitates parallel work on features while reducing coordination efforts.

The Deployment Monolith is subject to far fewer changes due to the addition of Microservices. For many features changes to the Monolith are not necessary anymore. Thus, the Deployment Monolith is more seldom deployed and changed. Originally, it was planned to completely replace the Deployment Monolith at some point. However, meanwhile it seems more likely that the Deployment Monolith will just be deployed less and less frequently as most changes take place within the Microservices. Thus the Deployment Monolith does not disturb work any more. To replace it entirely is in the end not necessary and also does not appear sensible in economic terms anymore.

Challenges

Implementing Microservices creates additional complexity in the beginning: All the Microservices need their own infrastructure. In parallel the Monolith has still to be supported.

The Microservices comprise a lot more servers and thus pose very different challenges. Monitoring and log file processing have to deal with the fact that the data originate from different servers. Thus information has to be centrally consolidated. Besides a substantially larger number of servers has to be handled – not only in production, but also in the different test stages and team environments. This is only possible with good infrastructure automatization. It is not only necessary to support different types of infrastructure for the Monolith and the Microservices, but also to provide substantially more servers in the end.

Entire Migration Lengthy

The added complexity due to the two different software types will persist for a long time as it is a very lengthy process to completely migrate away from the Monolith. If the Monolith is never entirely replaced, the additional infrastructure costs will remain as well.

Testing Remains a Challenge.

Testing is an additional challenge: Previously the entire Deployment Monolith was tested in the deployment pipeline. These tests are complex and take a long time as all functionalities of the Deployment Monolith have to be tested. If each change to each Microservice is sent through these tests, it will take a long time for each change to reach production. Moreover, the changes have to be coordinated as each change should be tested in isolation so that it is easily discernible in case of errors which change caused them. In that scenario a Microservices-based architecture does not seem to have major advantages over a Deployment

Monolith: While Microservices can in principle be deployed independently of each other, the test stages preceding deployment still have to be coordinated and each change still has to pass through them singly.

Current Status of Migration

Fig. 3 presents the current status: Product search works on an independent Microservice and is completely independent of the Deployment Monolith. Coordination with other teams is hardly necessary. Only in the last stage of the deployment the Deployment Monolith and the Microservices have to be tested together. Each change to the Monolith or any Microservice has to run through this step. This causes a bottleneck. The team “Customer” works together with the team “Order Process” on the Deployment Monolith. In spite of Microservices these teams still have to closely coordinate their work. For that reason the team “Order Process” has implemented its own Microservice, which comprises part of the order process. In this part of the system changes can be introduced faster than in the Deployment Monolith - not only due to the younger code basis, but also because it is no longer necessary to coordinate with the other teams.

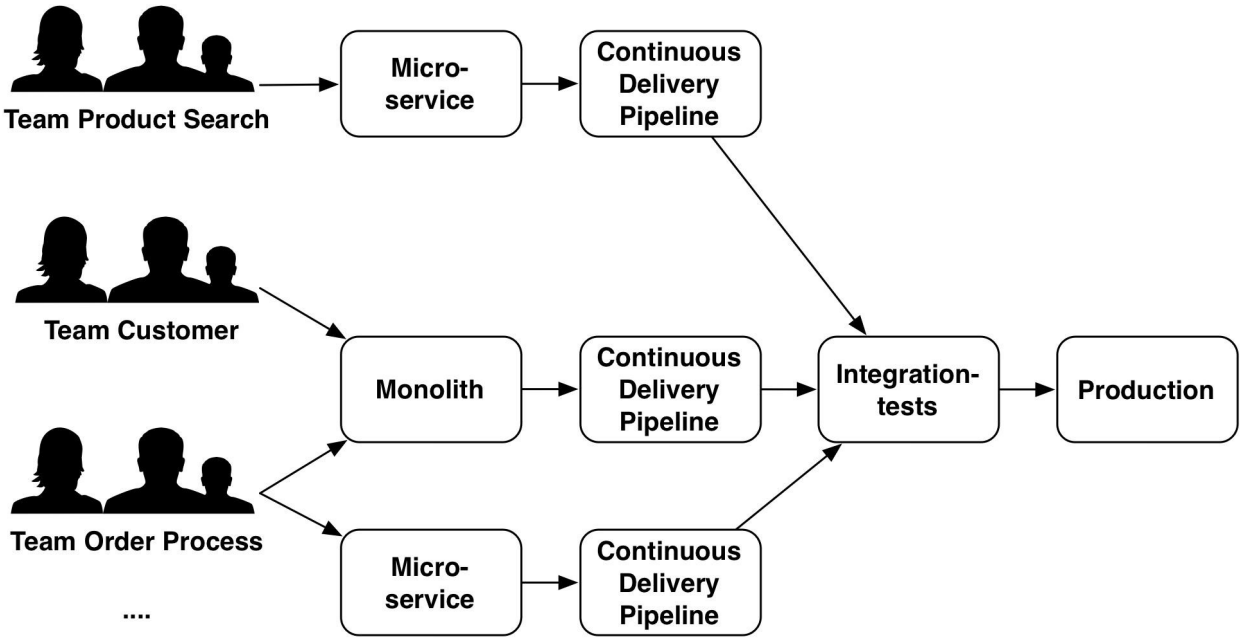


Fig. 3: Independent work through Microservices

Creating Teams

For the teams to be able to work independently on features it is important to create teams according to functionalities such as product search, customer or order process. If teams are instead created along technical layers such as UI, Middle Tier or database, each feature requires the involvement of all the teams as a feature normally comprises changes to UI, Middle Tier and database. Thus to minimize coordination efforts between the teams, the best approach is to create teams around features like product search. Microservices support the independence of the teams by their own technical independence from each other. Consequently, teams need to coordinate less in respect to basic technologies and technical designs.

The tests have also to be modularized. Each test should ideally deal with a single Microservice. In that case it is sufficient to perform the test upon changes in the respective Microservice. In addition it might be possible to implement the test rather as unit test than as integration test. This progressively shortens the test phase in which all Microservices and the Monolith have to be tested together. This reduces the coordination problems for the final test phase.

Migrating to a Microservices-based architecture created a number of performance problems and also some problems upon network failures. However, these problems could be solved after some time.

Advantages

Thanks to the new architecture changes can be deployed much faster. A team can bring a change into production within 30 minutes. The Deployment Monolith on the other hand is deployed only weekly due to the not yet fully automated tests.

Deploying the Microservices is not only much faster, but also in other respects much more comfortable: Less coordination is required. Errors are more easily found and fixed because developers still know very well what they have been working on as it was only 30 minutes ago.

In summary the goal was attained: The developers can introduce more changes to the E-Commerce shop. This is possible because the teams need to coordinate their work less and because the deployment of a Microservice can take place independently of the other services.

The possibility to use different technologies was sparingly used by the teams: The previously used technology stack proved sufficient, and the teams wanted to avoid the additional complexity caused by the use of different technologies. However, the long needed search engine for the product search was introduced. The team responsible for product search was able to implement this change on its own. Previously the introduction of this new technology had been prohibited because the associated risk had been considered too great. In addition some teams meanwhile have new versions of the libraries of the technology stack in production as they needed the bug fixes of the more recent version. This did not require any coordination with the other teams.

Conclusion

Replacing a Monolith via the implementation of Microservices is nearly a classical scenario for the introduction of Microservices. It requires a lot of effort to keep developing a Monolith and to add new features to it. The complexity of the Monolith and consequently the problems caused by it progressively increase over time. Its complete replacement by a newly written system is very difficult. The software has to be replaced in one go which is very risky.

Rapid and Independent Development of new Features

Especially in the case of companies like Big Money Online Commerce inc. the rapid development of new features and the parallel work on several features are vital for economic success. Only by providing state of the art features customers

can be won and kept from changing to other companies. The promise to develop more features faster renders Microservices highly attractive for many use cases.

Influence on the Organization

The presented example illustrates also the influence of Microservices on the organization. The teams work on their own Microservices. As the Microservices can be developed and deployed independently of each other, the work of the different teams is no longer linked. In order to keep it that way a Microservice may not be changed by several teams in parallel. The Microservices architecture requires a team organization corresponding to the different Microservices: Each team is responsible for one or several Microservices, which implement an isolated functionality. This relationship between organization and architecture is especially important in the case of Microservices-based architectures. Each team takes care of all issues revolving around “its” Microservices from requirements engineering up to operation monitoring. Of course, especially for operation the teams can use common infrastructure services for logging and monitoring.

And finally: If the goal is to achieve a simple and fast deployment in production, just including Microservices into the architecture will not be sufficient. The entire Continuous Delivery pipeline has to be checked for potential obstacles and these have to be removed. This is illustrated by the tests in the presented example: Testing all Microservices together should be reduced to the essential minimum. Each change has to run through an integration test together with the other Microservices, but this test must not require a lot of time to avoid a bottleneck in integration tests.

Amazon Has Been Doing It for a Long Time

The scenario presented here is very similar to what Amazon has been doing already for a very long time – and for the discussed reasons: Amazon wants to be able to rapidly and easily implement new features on its website. In 2006 Amazon did not only present its Cloud platform, but also discussed how it develops software. Essential features are:

- The application is divided into different services.
- Each services provides a part of the website. For instance there is a service for searching and another one for recommendations. In the end the individual services are presented together in the UI.
- There is always one team responsible for one service. The team takes care of developing new features as well as of operating the service. The idea is:

"You build it – you run it!"
- The Cloud platform i.e. virtual machines constitute the common foundation of all services. Apart from that there are no further standards. Thus the teams are very free in their choice of technologies.

By introducing this type of architecture Amazon implemented fundamental characteristics of Microservices already in 2006. Moreover Amazon introduced DevOps by having teams consisting of operation experts and developers. This approach necessitates that the deployments occur largely in an automated fashion as the manual construction of servers is not feasible in Cloud environments – thus Amazon also implemented at least one aspect of Continuous Delivery.

Conclusion: Microservices have been used by some companies for quite some time already – especially by companies having an internet-based business model. Thus the approach has already proven its practical advantages in real life. In addition Microservices display synergy effects with other modern approaches such as Continuous Delivery, Cloud or DevOps.

3.2 Developing a New Signaling System

Scenario

Searching airplanes and ships which have gone missing is a complex task. Rapid action can save lives. Therefore different systems are required. Some provide signals such as radio or radar signals. These signals have to be recorded and processed. Radio signals for example can be used to obtain a bearing, which subsequently has to be checked against radar-based pictures. Finally humans have to further evaluate the information. The data analyses as well as the raw data have to be provided to the different rescue teams. Signal inc. builds systems for exactly these use cases. The systems are individually assembled, configured and adapted to the specific needs of the respective client.

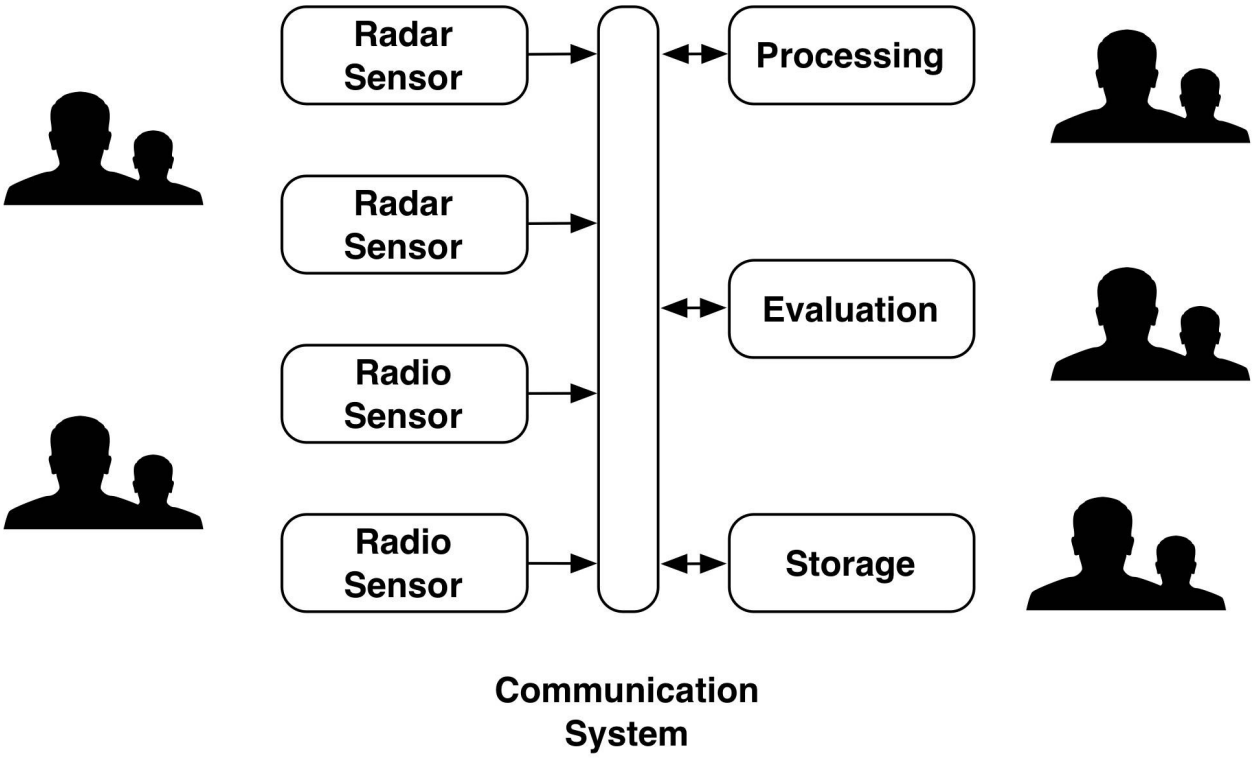


Fig. 4: Overview of the Signaling System

Reasons to Use Microservices

The system is composed of different components, which run on different computers. The sensors are distributed all over the area to be monitored and are provided with their own servers. However, these computers are not supposed to handle the more detailed data processing or to store the data. Their hardware is not sufficiently powerful for that. Besides data privacy considerations render such an approach very undesirable as well.

Distributed System

For these reasons the system has to be a distributed system. The different functionalities are distributed within the network. The system is unreliable as individual components and the communication between components can fail.

It would be possible to implement a large part of the system within a Deployment Monolith. However, upon closer consideration the different parts of the system have to fulfill very different demands. Data processing requires rather a lot of CPU and an approach that allows numerous algorithms to process the data. For such purposes there are solutions, which read events out of a data or event stream and process them. Data storage requires a very different focus: Basically, the data have to be maintained within a data structure, which is suitable for different data

analyses. Modern NoSQL databases are well suited for this. Recent data are more important than old data. They have to be accessible faster while old data can even be deleted at some point. To be finally analyzed by experts the data have to be read from the database and processed.

Technology Stack per Team

Each of the discussed tasks poses different challenges. Consequently, each requires not only a well adapted technology stack but also a dedicated team consisting of technical experts for the respective task. In addition people are needed who decide which features Signal inc. will bring to the market and in line with that define new requirements for the systems. Systems for processing and sensors are individual products, which can be positioned on the market independently of each other.

Integration of Other Systems

An additional reason for the use of Microservices is the possibility to easily integrate other systems. Sensors and computing units are also provided by other companies. The ability to integrate such solutions is a frequent requirement in client projects. Microservices allow the easy integration of other systems as the integration of different distributed components is anyhow a core feature of a Microservices-based architecture.

For these reasons the architects of Signal inc. decided to indeed implement their system as a distributed system. Each team must implement its respective domain in several small Microservices. In this way the exchangeability of the Microservices will be further improved, and the integration of other systems will be even more facilitated.

Only the communication infrastructure to be used by all services for their communication with each other is predetermined. The communication technology supports many programming languages and platforms so that there are no limitations as to which concrete technology is used. To allow for a flawless communication the interfaces between the Microservices have to be clearly defined.

Challenges

A failure of communication between the different Microservices presents an important challenge. The system has to stay useable even if network failures occur. This necessitates the use of technologies, which can handle such failures.

However, technologies alone will not solve this problem. It has to be decided as part of the user requirements what should happen if a system fails. If for instance old data are sufficient, caches can be helpful. In addition it can be possible to use a simpler algorithm, which does not require calls to other systems.

High Technological Complexity

The technological complexity of the entire system is very high. Different technologies are employed to fulfill the demands of the different components. The teams working on the individual systems can make largely independent technology decisions. This allows them to always implement the most suitable solution.

Unfortunately, this means as well that developers can no longer change easily between teams. For example when there was a lot of work for the data storage team, developers from other teams could hardly help out as they were not even proficient in the programming languages the data storage team was using and did not know the specific technologies such as the used database.

It certainly is a challenge to run a system comprising such a plethora of technologies. For this reason there is one standardization in this area: All Microservices must be able to be run in a largely identical manner. They are virtual machines so that their installation is fairly simple. Furthermore, the monitoring is standardized, which determines data formats and technologies. This allows for the central monitoring of the applications. In addition to the typical operational monitoring there is also monitoring of application-specific values and finally an analysis of log files.

Advantages

In this context the main advantage offered by Microservices is the good support for the distributed nature of the system. The sensors are at different locations so that a centralized system is anyhow not sensible. The architecture has adapted to this fact by further dividing the system into small Microservices, which are distributed within the network. This enhanced the exchangeability of the Microservices. Besides the Microservices approach supports the technology diversity, which characterizes this system.

In this scenario time-to-market is by far not as important as in the other scenario. It would also be hard to implement as the systems are installed at different clients and cannot be easily reinstalled. However, some ideas from the Continuous

Delivery field are used: For instance the largely uniform installation and the central monitoring.

Verdict

Microservices are a very suitable architecture design for the presented scenario. The system can profit from the fact that typical problems can be solved during implementation by established approaches from the Microservices field – for example technology complexity and platform operation.

Still this scenario would hardly be immediately associated with the term “Microservice”. This leads to the following conclusions:

- Microservices have a wider application than is apparent at first glance. Also outside of web-based business models Microservices can solve many problems – even if those issues are very different from the ones found in web companies.
- Indeed many projects from different fields have been using Microservice-based approaches for quite some time, even if they do not call them by this name or implement them only partially.
- With the help of Microservices these projects can use technologies, which are currently created in the Microservice field. In addition they can profit from the experiences made in this field, for instance in regards to architecture.

3.3 Conclusion

This chapter presented two very different scenarios from two completely distinct business areas: a web system with a strong focus on rapid time-to-market and a system for signal processing, which is inherently distributed. The architecture principles are very similar for the two systems although originating from different reasons.

In addition there are a number of common approaches, among those the creation of teams according to Microservices, the demands in regards to infrastructure automatization as well as other organizational topics. However, in other areas there are also differences. For the signaxsling system it is essential to have the possibility to use different technologies as this system anyhow has to employ a number of different technologies. For the web system this aspect is not as

important. Here, the independent development, the fast and easy deployment and finally the better time-to-market are the decisive factors.

Essential Points

- Microservices offer a plethora of advantages.
- In the case of web-based applications Continuous Delivery and short time-to-market can constitute an important motivation for the use of Microservices.
- However, there are also very different use cases for which Microservices as distributed systems are extremely well suited.

Part II: Microservices: What, Why and Why Not?

This part of the book discusses the different facettes of Microservice-based architectures to present the diverse possibilities offered by Microservices. Advantages as well as disadvantages are addressed so that the reader can evaluate what can be gained by using Microservices and which points require special attention and care during the implementation of Microservice-based architectures.

Chapter 4 explains the term “Microservice” in detail. The term is dissected from different perspectives, which is essential for an in depth understanding of the Microservice approach. Important aspects are the size of a Microservice, Conway’s Law as organizational influence and Domain-Driven Design resp. *Bounded Context* from a domain perspective. Furthermore, the chapter addresses the question whether a Microservice should contain a UI. Chapter 5 focuses on the advantages of Microservices taking alternatingly a technical, organizational and business perspective. Chapter 6 deals with the associated challenges in the areas of technology, architecture, infrastructure and operation. Chapter 7 distinguishes Microservices from SOA (Service-Oriented Architecture). By making this distinction Microservices are viewed from a new perspective which helps to further clarify the Microservices approach. Besides Microservices have been frequently compared to SOAs.

Afterwards the third part of the book will introduce how Microservices can be implemented in practice.

4 What are Microservices?

Section 1.1 provided an initial definition of the term “Microservice”. However, there are also different possibilities to define Microservices. The different definitions illustrate the different characteristics of Microservices and thereby explain for which reasons the use of Microservices is advantageous. At the end of the chapter the reader should have his/her own definition of the term “Microservice” – depending on the individual project scenario.

The chapter discusses the term “Microservice” from different perspectives:

- Section 4.1 focuses on the size of Microservices.
- Section 4.2 sets Microservices, architecture and organization into relation by using the law of Conway.
- Finally section 4.3 presents a fachliche division of Microservices based on Domain-driven Design (DDD) and Bounded Context.
- Section 4.4 explains why Microservices should contain a UI.

4.1 Size of a Microservice

The name “Microservices” betrays already that the size of the service matters, obviously Microservices are supposed to be small.

One possibility to define the size of a Microservice is to count the Lines of Code (LoC). However, such an approach entails a number of problems:

- It depends on the programming language used. Some languages require more code than others to express the same content – and Microservices are explicitly not supposed to predetermine the technology stack. Accordingly, defining Microservices based on this metric is not very useful.
- Finally Microservices represent an architecture approach. Architectures, however, should follow the conditions in the domain rather than adhering to technical metrics such as LoC. Also for this reason attempts to determine size based on code lines should be viewed critically.

In spite of the voiced criticism LoC can be an indicator for a Microservice. Still, the question as to the ideal size of a Microservice remains. How many LoC may a Microservice have? Even if there are no absolute standard values, there are nevertheless influencing factors, which argue for larger or smaller Microservices.

Modularization

One factor is the modularization. Teams develop software in modules to be better able to deal with its complexity: Instead of having to understand the entire software a developer only needs to understand the module he is working on as well as the interplay between the different modules. This is the only way for a team to work productively in spite of the enormous complexity of a typical software system. In daily life there are often problems as modules get larger than originally planned. This makes them hard to understand and hard to maintain as changes require an understanding of the software. Thus it is very sensible to keep Microservices as small as possible. On the other hand Microservices in contrast to many other approaches to modularization have an overhead:

Distributed Communication

Microservices run within independent processes. Therefore communication between Microservices is distributed communication via the network. To this type of system the "First Rule of Distributed Object Design" applies. This rule states that systems should not be distributed if it can be avoided. The reason behind this is that a call on another system via the network is orders of magnitude slower than a direct call within the same process. In addition to the pure latency time serialization and deserialization of parameters and results are time-consuming. These processes do not only take a long time, but also cost CPU capacity.

Moreover, distributed calls might fail because the network is temporarily unavailable or the called server cannot be reached – for instance due to a crash. This increases complexity when implementing distributed systems as the caller has to deal with these errors in a sensible manner.

Experience teaches that Microservice-based architectures work in spite of these problems. When Microservices are designed to be especially small, the amount of distributed communication increases and the overall system gets slower. This argues for larger Microservics. When a Microservice contains a UI and fully implements a specific part of the domain , it can do without calling on other Microservices in most cases because all components of this part of the domain are

implemented within one Microservice. The wish to prevent distributed communication is another reason to build systems according to the domain.

Sustainable Architecture

Microservices use distribution also to design architecture in a sustainable manner through distribution into individual Microservices: It is much more difficult to use a Microservice than a class. The developer has to deal with the distribution technology and has to use the Microservice interface. In addition he might have to make preparations for tests to include the called Microservice or replace it with a stub. Finally, he has to contact the team responsible for the respective Microservice.

To use a class within a Deployment Monolith is much simpler – even if the class belongs to a completely different part of the Monolith and falls within the responsibility of another team. However, as it is so simple to implement a dependency between two classes, unintended dependencies tend to accumulate within Deployment Monoliths. In the case of Microservices dependencies are harder to implement, which prevents the creation of unintended dependencies.

Refactoring

However, the boundaries between Microservices create also challenges, for instance during refactoring. When it becomes apparent that a certain functionality is not fitting well within its present Microservice, it has to be moved to another Microservice. If the target Microservice is written in a different programming language, this transfer corresponds ultimately to a new implementation. Such problems do not arise when functionalities are moved within a Microservice. This factor argues also rather for larger Microservices. This topic is the focus of Section 8.3.

Team Size

The independent deployment of Microservices and the division into teams result in an upper limit for the size of an individual Microservice. A team should be able to implement features within a Microservice independently of other teams and to bring them also independently into production. In this way the architecture enables the scaling of development without requiring too much coordination effort from the teams.

A team has to be able to implement features independently of the other teams. Therefore at first glance it seems like the Microservice should be large enough to

allow for the implementation of different features. When Microservices are smaller, a team can be responsible for several Microservices, which together allow the implementation of a domain. A lower limit for the Microservice size does not result from the independent deployment and the division into teams.

However, an upper limit does result from it: When a Microservice has reached a size that prevents its further development by a single team, it is too large. For that matter a team should have a size that is especially well suited for agile processes, i.e. typically three to nine people. Thus a Microservice should in no case grow so large that three to nine people cannot develop it further by themselves. In addition to sheer size the number of features to be implemented in an individual Microservice plays an important role. Whenever a large amount of changes is necessary within a short time, a team can be rapidly overloaded. Section 13.2 highlights alternatives to allow several teams to work on the same Microservice. However, in general a Microservice should never grow so large that several teams are necessary to work on it.

Infrastructure

Another important factor influencing the size of a Microservice is the infrastructure. Each Microservice has to be able to be deployed independently. It must have a Continuous Delivery pipeline and an infrastructure for running the Microservice, which has to be present not only in production, but also during the different test stages. Also databases and application servers might belong to infrastructure. Moreover, there has to be a build system for the Microservice. The Microservice code has to be versioned independently of other Microservices. Thus a project within version control has to exist for the Microservice.

Depending on the effort that is necessary to provide the required infrastructure for a Microservice, the sensible size for a Microservice can vary. When a small Microservice size is chosen, the system is distributed into many Microservices thus requiring more infrastructures. In the case of larger Microservices the system contains overall fewer Microservices and consequently requires fewer infrastructures.

Build and deployment of Microservices should anyhow be automated. Nevertheless it can be laborious to provide all necessary infrastructure components for a Microservice. Once setting up the infrastructure for new Microservices is automated, the expenditure for providing infrastructures for additional Microservices decreases. This allows to further reduce the

Microservice size. Companies, which have been working with Microservices for some time, usually simplify the creation of new Microservices by providing the necessary infrastructure in an automated manner.

Besides there are technologies, which allow to reduce the infrastructure overhead to such an extent that substantially smaller Microservices are possible – in that case, however, with a number of limitations. Such Nanoservices are discussed in chapter 15.

Replaceability

A Microservice should be as easy to replace as possible. Replacing a Microservice can be sensible when its technology is outdated or the Microservice code is of such a bad quality that it cannot be developed any further. The replaceability of Microservices is an advantage as compared to monolithic applications, which hardly can be replaced at all. When a Monolith cannot be maintained anymore, its development has either to be continued in spite of the associated high costs or a likewise cost-intensive migration has to take place nevertheless. The smaller a Microservice is, the easier it is to replace it by a new implementation. Above a certain size a Microservice can hardly be replaced anymore because replacing it then poses the same challenges as for a Monolith. Replaceability thus limits the size of a Microservice.

Transactions and Consistency

Transactions possess the so-called ACID characteristics:

- **Atomicity** indicates that a given transaction is either executed completely or not at all. In case of an error all changes are reversed again.
- **Consistency** means that data are consistent before and after the execution of a transaction – validations for instance are not violated.
- **Isolation** indicates that the operations of transactions are separated from each other.
- **Durability** indicates permanence: Changes to the transaction are stored and are still available after a crash.

Within a Microservice changes to a transaction can take place. Moreover, the consistency of data in a Microservice can be guaranteed very easily. Beyond an individual Microservice this gets difficult. In that case an overall coordination is necessary. Upon the rollback of a transaction all changes to all Microservices would have to be reversed. This is laborious and hard to implement as there has

to be a guarantee that the decision whether changes have to be reversed is delivered. However, communication within networks is unreliable. Until it is decided whether a change may take place, further changes to the data are barred. For once additional changes have taken place, it might not be possible anymore to reverse a certain change. However, when Microservices are kept from introducing data changes for some time, the system throughput is reduced.

However, when communicating via messaging systems, transactions are possible (compare Section 9.4). With such an approach transactions are also possible without a close link between the Microservices.

Consistency

In addition to transactions data consistency is important. An order for instance has finally to be recorded as revenue. Only then will revenue and order data be consistent. Data consistency can only be achieved through close coordination. Data consistency can hardly be guaranteed across Microservices. This does not mean that the revenue for an order will not be recorded at all. However, it will likely not happen at the exact same time point and maybe not even within one minute of order processing as the communication occurs via the network - and is consequently slow and unreliable.

Data changes within a transaction and data consistency are only possible when all concerned data is part of the same Microservice. Therefore they determine the lower size limit for a Microservice: When transactions are supposed to encompass several Microservices and data consistency is required across several Microservices, the Microservices have been designed too small.

Compensation Transactions Across Microservices

At least in the case of transactions there is an alternative: If a data change has to be rolled back in the end, compensation transactions can be used for that.

A classical example for a distributed transaction is a travel booking, which consists of a hotel, a rental car and a flight. Either everything has to be booked together or nothing at all. Within real systems and also within Microservices this functionality is divided into three Microservices because the three tasks are very different. Inquiries are sent to the different systems whether the desired hotel room, the desired rental car and the desired flight are available. If that is the case, everything is reserved. If now, for instance, the hotel room suddenly becomes unavailable, the reservation for the flight and the rental car has to be cancelled.

However, in the real world the concerned companies will likely demand a fee for the booking cancellation. Due to that the cancellation is not only a technical event happening in the background like a transaction rollback, but a business process. This is much easier to represent with a compensation transaction. With this approach transactions across several elements in Microservice environments can also be implemented without the presence of a close technical link. A compensation transaction is just a normal service call. Technical as well as business reasons can argue for the use of mechanisms like compensation transactions for Microservices.

Summary

In conclusion the following factors influence the size of a Microservice (compare Fig. 5):

- The team size sets an upper limit: A Microservice should never be that large that several teams are required to work on it. Eventually, the teams are supposed to work and bring software into production independently of each other. This can only be achieved when each team works on a separate deployment unit – i.e. a separate Microservice. However, one team can work on several Microservices.
- Modularization further limits the size of a Microservice: The Microservice should preferably be of a size that allows a developer to understand and further develop it. Even smaller is of course better. This limit is below the team size: Whatever one developer can still understand, a team should still be able to develop further.
- Replaceability declines with the size of the Microservice. Therefore replaceability can influence the upper size limit for a Microservice. This limit lies below the one set by modularization: When somebody is able to replace a Microservice, this person has first of all to be able to understand the Microservice.
- A lower limit is set by infrastructure: If it is too laborious to provide the necessary infrastructure for a Microservice, the number of Microservices should be kept rather small – consequently the single Microservices are then rather larger.
- Similarly, distributed communication increases with the number of Microservices. Also for this reason the size of Microservices should not be set too small.
- Consistency of data and transactions can only be ensured within a Microservice. Therefore Microservices may not be that small that

consistency and transactions comprise several Microservices.

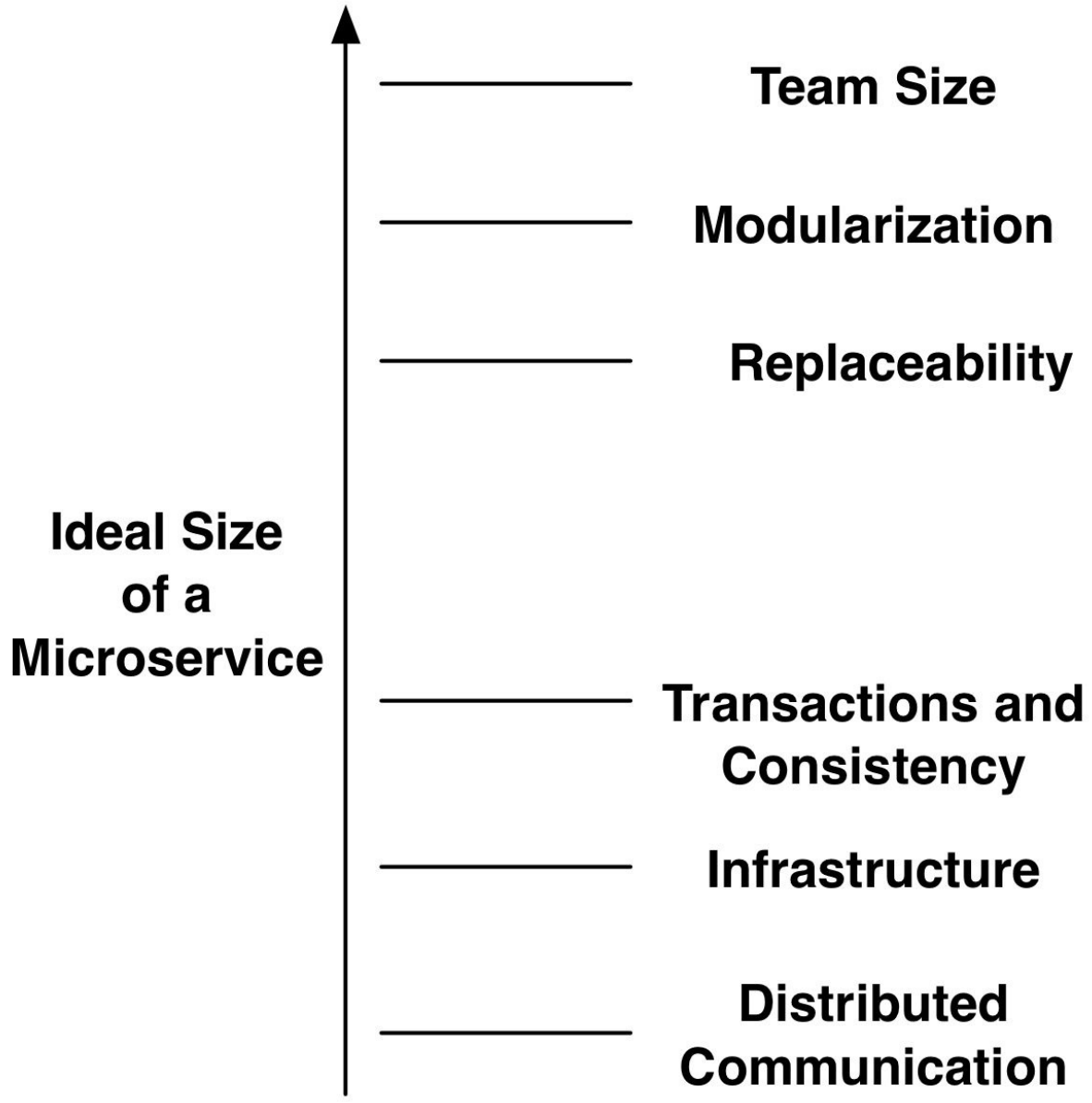


Fig. 5: Factors Influencing the Size of a Microservice

These factors do not only influence the size of Microservices, but they also reflect a certain idea of Microservices. According to this idea the main advantages of Microservices are independent deployment and the independent work of the different teams, and in addition the replaceability of Microservices. The optimal size of a Microservice can be deduced from these desired features.

However, there are also other reasons for Microservices. When Microservices are, for instance, introduced because of their independent scaling, a Microservice

size has to be chosen that ensures that each Microservice is a unit, which has to scale independently.

How small or large a Microservice can be, cannot be deduced solely from this lineup. This also depends on the technology. Especially the effort necessary for providing infrastructure for a Microservice and for the distributed communication depends on the utilized technology. Chapter 15 looks at technologies, which make the development of very small services possible – denoted as Nanoservices. These Nanoservices have different advantages and disadvantages as Microservices, which, for instance, are implemented using technologies presented in Chapter 14.

Thus, there is no ideal size. The actual Microservice size will depend on the technology and the use case of an individual Microservice.

Try and Experiment

How large is the effort for the deployment of a Microservice in your language, platform and infrastructure?

- Is it just a simple process? Or a complex infrastructure containing application servers or other infrastructure elements?
- How can the effort for the deployment be reduced so that smaller Microservices become possible?

Based on this information you can define a lower limit for the size of a Microservice. Upper limits depend on team size and modularization – also in those terms you should think of appropriate limits.

4.2 Conway's Law

Conway's Law was coined by the American computer scientist Melvin Edward Conway and indicates:

> Any organization, that designs a system (defined broadly), will produce a design whose structure is a copy of the organization's communication structure.

It is important to know that this law is not only meant to apply to software, but to any kind of design. The communication structures, which Conway mentions, do not

have to be identical with the organization chart. Often there are informal communication structures, which also have to be considered in this context. In addition the geographical distribution of teams can influence communication. After all it is much simpler to talk to a colleague who works in the same room or at least in the same office than with one working in a different city or even in a different time zone.

Reasons for the Law

The reasons behind the Law of Conway derive from the fact that each organizational unit designs a specific part of the architecture. If two architecture parts have an interface, coordination in regards to this interface is required – and, consequently, a communication relationship between the organizational units, which are responsible for the respective architecture parts.

From the Law of Conway it can also be deduced that design modularization is sensible. Via such a design it is ensured that not every team member has to constantly coordinate with every other team member. Instead the developers working on the same module can closely coordinate their efforts, while team members working on different modules only have to coordinate when they develop an interface – and even then only in regards to the specific design of this interface.

However, the communication relationships extend beyond that. It is much easier to collaborate with a team within the same building than with a team located in another city, another country or even within a different time zone. Therefore architecture parts having numerous communication relationships are better implemented by teams, which are geographically close to each other, as it is easier for them to communicate with each other. In the end the Law of Conway does not focus on the organization chart, but on the real communication relationships.

By the way, Conway postulated that a large organization has numerous communication relationships. Thus communication becomes more difficult or even impossible in the end. As a consequence the architecture can be more and more affected and finally break down. In the end having too many communication relationships is a real risk for a project.

The Law as Limitation

Normally the Law of Conway is viewed as a limitation, especially from the perspective of software development. Let us assume that a project is modularized according to technical aspects (Fig. 6). All developers with UI focus are grouped into one team, the developers with backend focus are put into a second team, and data bank experts make up the third team. This distribution has the advantage that all three teams consist of experts for the respective technology. This makes it easy and transparent to realize this type of organization. Moreover, this distribution appears also logical. Team members can easily support each other, and the technical exchange is also facilitated.

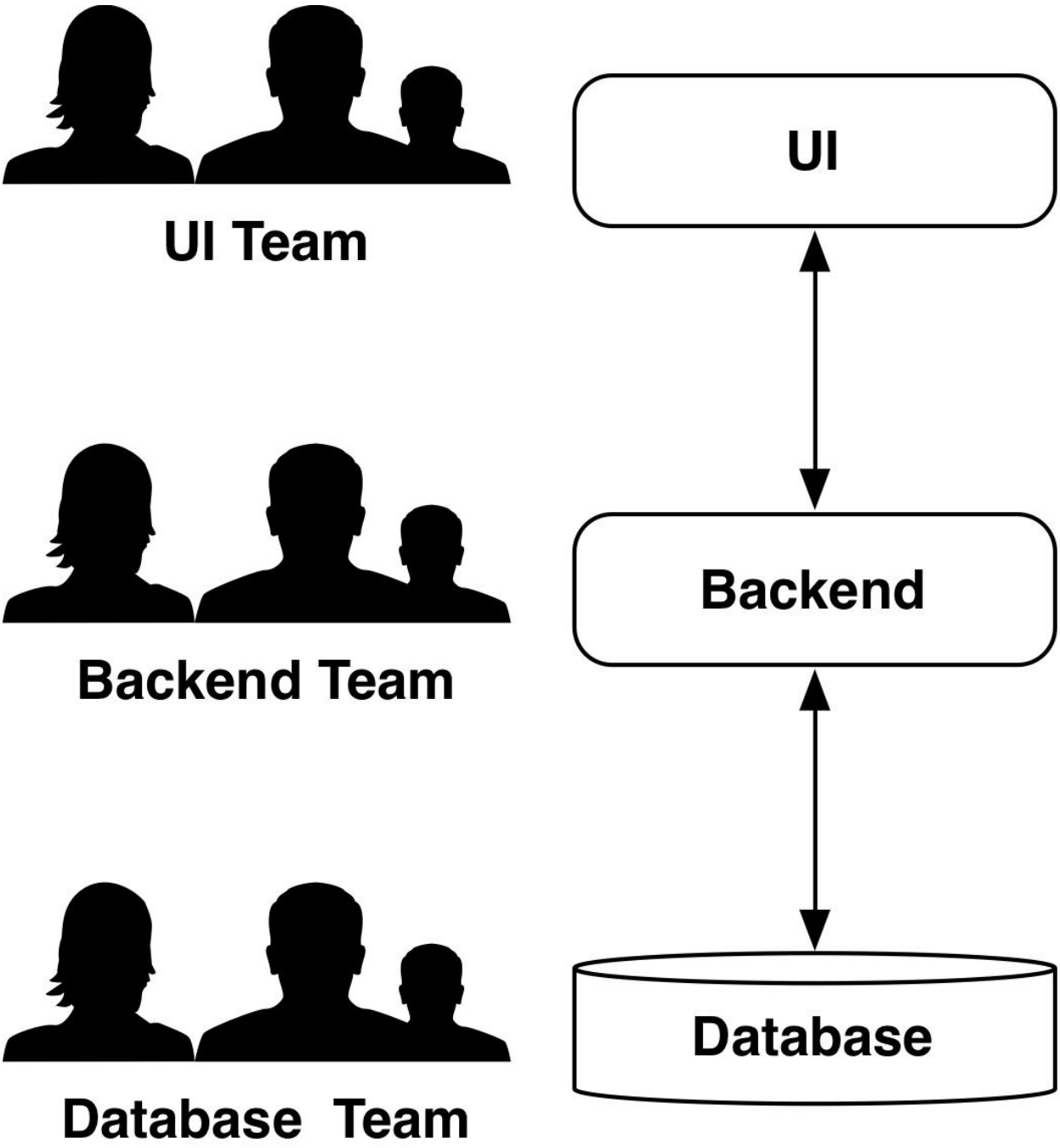


Fig. 6: Technical Project Distribution

According to the Law of Conway it follows from such a distribution that the three teams will implement three technical layers: a UI, a backend and a database. The chosen distribution corresponds to the organization, which is in fact sensibly built. However, it has a decisive disadvantage: A typical feature requires changes to UI, backend and database. The UI has to render the new features useable for the clients, the backend has to implement the logic, and the database has to create

structures for the storage of the respective data. This results in the following disadvantages:

- The person wishing to have the feature implemented has to talk to all three teams.
- The teams have to coordinate their work and create new interfaces.
- The work of the different teams has to be coordinated in a manner that ensures that their efforts temporally fit together. The backend, for instance, cannot really work without getting input from the database – and the UI cannot work without input from the backend.
- When the teams work in sprints, these dependencies cause time delays: The database team generates in its first sprint the necessary changes, within the second sprint the backend team implements the logic, and in the third sprint the UI is dealt with. In this way it takes three sprints to implement a single feature.

In the end this approach creates a large amount of dependencies as well as a high communication and coordination demand. Thus this type of organization does not make much sense if the main goal is to implement new features as rapidly as possible.

Many teams following this approach do not realize its impact on architecture and do not consider this aspect further. This type of organization focuses rather on the aspect that developers with similar skills should be similarly positioned within the organization. In this way the organization turns into an obstacle for a design driven by the domain like Microservices whose development is opposed by the division of teams into technical layers.

The Law as Enabler

However, the law of Conway can also be used to support approaches like Microservices. If the goal is to develop individual components as independently of each other as possible, the system can be distributed into domain components. Based on these domain components teams can be created. Fig. 7 illustrates this principle: There are individual teams for product search, clients and order process. These teams work on the respective components, which can be technically divided into UI, backend and database. By the way, the domain components are not explicitly named in the figure as they are identical with the team names. Components and teams are synonymous. This approach corresponds to the idea of so-called cross functional teams, as proposed by methods with

Scrum. These teams should encompass different roles so that they can cover a large task spectrum. Only a team designed along such principles can be in charge of a component – from engineering requirements via implementation up to operation.

The division into technical artifacts and the interface between the artifacts can then be settled within the teams. In the easiest case a developer has only to talk to the developer sitting next to him to do so. Between teams coordination is more complex. However, inter-team coordination is not required very often since features are ideally implemented by independent teams. Moreover, this approach creates thin interfaces between the components. This avoids laborious coordination across teams to define the interface.

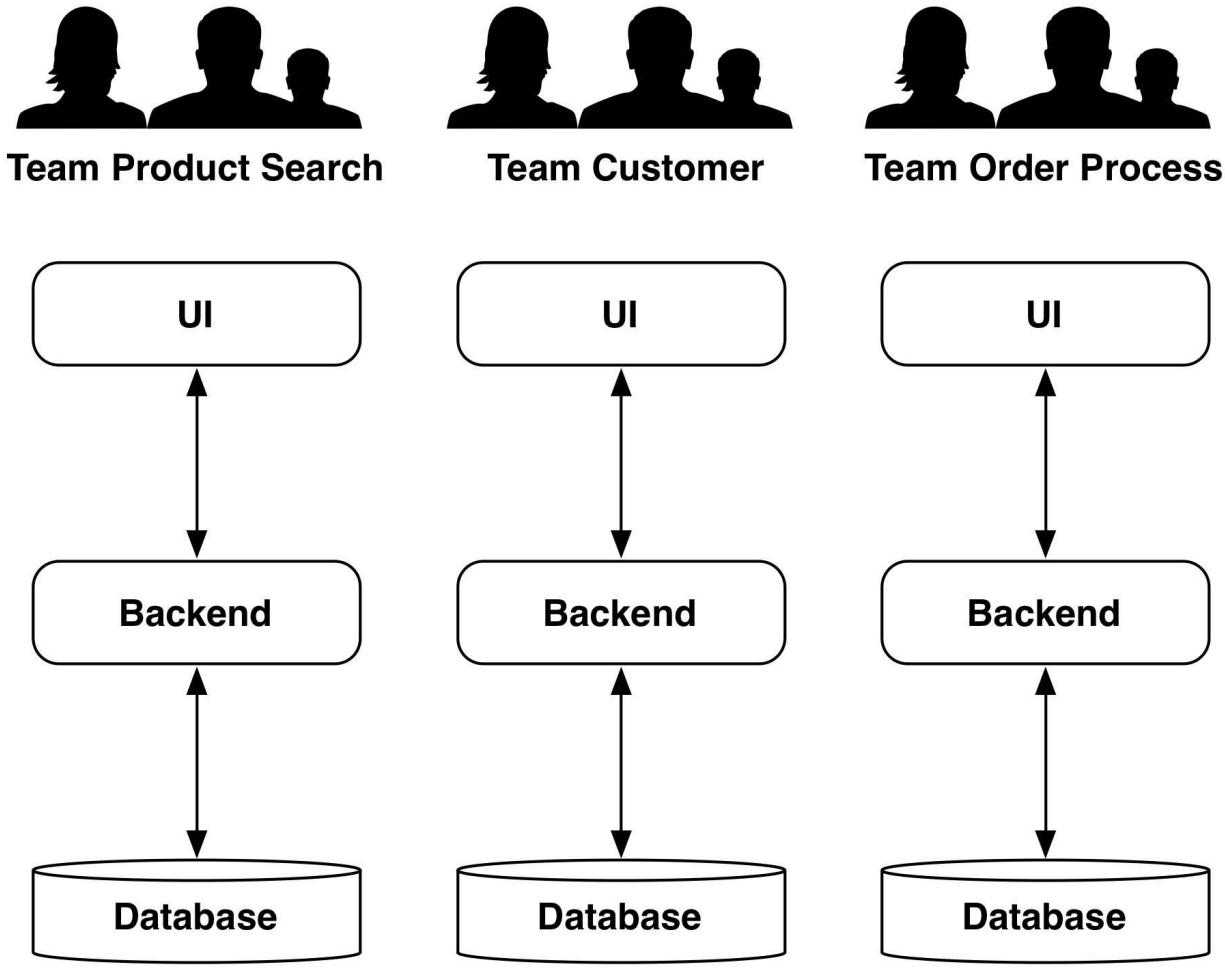


Fig. 7: Project by domains

Eventually, the central point to be derived from Conway's Law is that architecture and organization are just two sides of the same coin. When this insight is cleverly put to use, the system will have a clear and useful architecture for the project. Architecture and organization have the common goal to ensure that teams can work in an unobstructed manner and with as little coordination effort as possible.

The clean distribution of functionalities into components also facilitates maintenance. Since an individual team is responsible for an individual functionality and component, this distribution will have long term stability, and consequently the system will remain maintainable.

The teams need requirements to work upon. This means that the teams need contact persons which define the requirements. This affects the organization beyond the projects as the requirements come from the departments of the enterprise, and also these according to Conway's Law have to correspond to the

team structures within the project and the domain architecture. Conway's Law can be expanded beyond software development to the communication structures of the entire organization including the users. To put it the other way round: The team structure within the project and consequently the architecture of a Microservice system can follow from the organization of the departments of the enterprise.

The Law and Microservices

The previous discussion highlighted the relationship between architecture and organization of a project only in a general manner. It would be perfectly conceivable to align the architecture along functionalities and devise teams, which each are in charge for a separate functionality without using Microservices. In this case the project would develop a Deployment Monolith within which all functionalities are implemented. However, Microservices support this approach. Section 3.1 already discussed that Microservices offer technical independence. In conjunction with the division by domains the teams become even more independent of each other and have even less need to coordinate their work. The technical coordination as well as the coordination concerning the domains can be reduced to the absolute minimum. This makes it far easier to work in parallel on numerous features and to bring the features also in production.

Microservices as technical architecture are especially well suited to support the approach to devise a Conway's Law-based distribution of functionalities. In fact, exactly this aspect is an essential characteristic of a Microservices-based architecture.

However, orienting the architecture according to the communication structures entails that a change to the one also requires a change of the other. This renders architecture changes between Microservices more difficult and makes the overall process less flexible. Whenever one functionality is moved from one Microservice to another, this might have the consequence that another team has to take care of this functionality from that point on. This type of organizational changes render software changes more complex.

As a next step this chapter will address how the distribution by domain can best be implemented. Domain-driven Design (DDD) is helpful for that.

Try and Experiment

Have a look at a project you know:

- What does the team structure look like?
  - Is it technically motivated or by domain?
  - Would the structure have to be changed to implement a Microservices-based approach?
  - How would it have to be changed?
- Is there a sensible way to distribute the architecture onto different teams? Eventually each team should be in charge of independent domain components and be able to implement features relating to them.
  - Which architectural changes would be necessary?
  - How laborious would the changes be?

4.3 Domain-Driven Design and Bounded Context

In his book of the same title Eric Evans formulated Domain-Driven Design (DDD)[1] as pattern language. It is a collection of connected design patterns and supposed to support software development especially in complex domains. In the following text the names of design patterns are written in *italics*.

Domain-Driven Design is important for understanding Microservices as it supports the structuring of larger systems according to domains. Exactly such a model is necessary for the division of a system into Microservices. Each Microservice is meant to constitute a domain, which is designed in such a way that only one Microservice has to be changed in order to implement changes or to introduce new features. Only then is the benefit to be derived from the independent development in different teams maximal as several features can be implemented in parallel without the need for extended coordination.

Ubiquitous Language

DDD defines as basis how a model for a domain can be designed. An essential foundation of DDD is *Ubiquitous Language*. This expression denotes that the software should use exactly the same terms as the domain experts. This applies on all levels: in regards to code and variable names as well as for database schemas. This practice ensures that the software really encompasses and implements the critical domain elements. Let us assume for instance that there are express orders in an E-commerce system. One possibility would be to generate a boolean value with the name “fast” in the order table. This creates the following problem: domain experts have to translate the term “express order”, which they use on a

daily basis, into “order with a specific boolean value”. They might not even know what boolean values are. This renders any discussion of the model more difficult as terms have to be constantly explained and related to each other. The better approach is to call the table within the database scheme “express order”. In that case it is completely transparent how the domain terms are implemented in the system.

Building Blocks

To design a domain model DDD identifies basic patterns:

- *Entity* is an object with an individual identity. In an E-commerce application the customer or the items could be examples for *Entities*. *Entities* are typically stored in databases. However, this is only the technical implementation of the concept *Entity*. An *Entity* belongs in essence to the domain modeling like the other DDD concepts.
- *Value Objects* do not have their own identity. An address can be an example for a *Value Object* as it makes only sense in the context of a specific customer and therefore does not have an independent identity.
- *Aggregates* are composite domain objects. They facilitate the handling of invariants and other conditions. An order for instance can be an *Aggregate* of order lines. This can be used to ensure that an order from a new customer does not exceed a certain value. This is a condition, which has to be fulfilled by calculating values from the order lines so that the order as *Aggregate* can control these conditions.
- *Services* contain business logic. DDD focuses on modeling business logic as *Entities*, *Value Objects* and *Aggregates*. However, logic accessing several such objects cannot be sensibly modeled using these objects. For these cases there are *Services*. The order process could be such a *Service* as it needs access to items and customers and requires the *Entity* order.
- *Repositories* serve to access all *Entities* of a type. Typically there is a persistency technology like a database behind a *Repository*.
- Factories are mostly useful to generate complex domain objects. This is especially the case when these contain for instance many associations.

Aggregates are of special importance in the context of Microservices: Within an *Aggregate* consistency can be enforced. Because of the necessary consistency parallel changes have to be coordinated in an *Aggregate*. Otherwise two parallel changes might endanger consistency. For instance, when two order positions are included in parallel into an order, consistency can be endangered. The order has

already a value of €900 and is maximally allowed to reach €1000. When two order positions of €60 each are added in parallel, both might calculate a still acceptable total value of €960 based on the initial value of €900. Therefore, changes have to be serialized so that the final result of €1020 can be controlled. Accordingly, changes to *Aggregates* have to be serialized. For this reason an *Aggregate* cannot be distributed across two Microservices. In such a scenario consistency cannot be ensured. Consequently, *Aggregates* cannot be divided between Microservices.

Bounded Context

Building Blocks such as *Aggregate* represent for many people the core of DDD. DDD describes in addition with Strategic Design how different domain models interact and how more complex systems can be built up this way. This aspect of DDD is probably even more important than the Building Blocks. In any case it is the concept of DDD, which influences Microservices.

The central element of Strategic Designs is the *Bounded Context*. The underlying reasoning is that each domain model is only sensible in certain limits within a system. In E-commerce for instance number, size and weight of the ordered items are of interest in regards to delivery, as they influence delivery routes and costs. For accounting on the other hand prices and tax rates are relevant. A complex system consists of several *Bounded Contexts*. In this it resembles the way complex biological organisms are built out of individual cells, which are likewise separate entities with their own inner life.

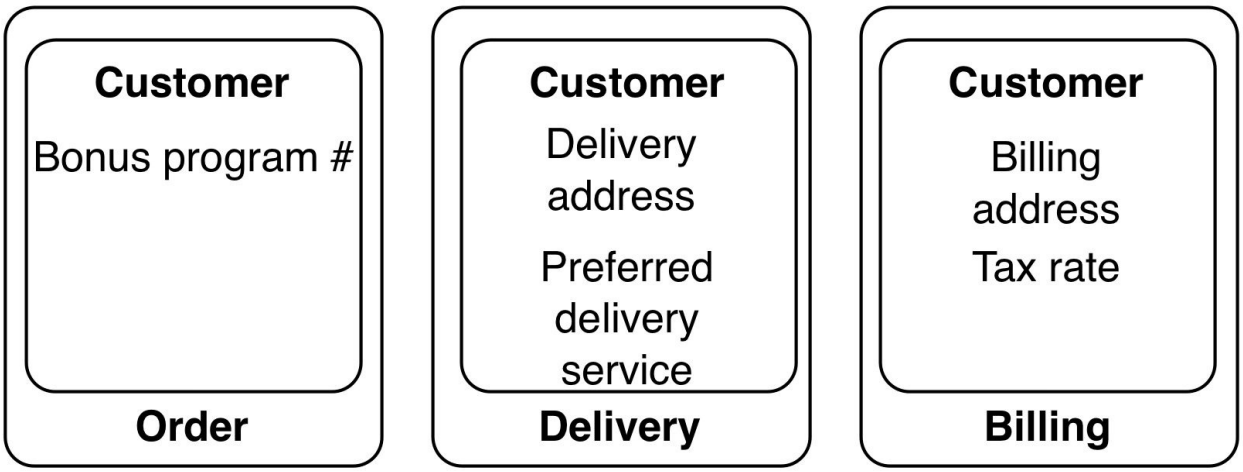


Fig. 8: Project by domains

Bounded Context: An example

The customer from the E-commerce system shall serve as example for a *Bounded Context* (Fig. 8). The different *Bounded Contexts* are Order, Delivery and Billing. The component Order is responsible for the order process. The component Delivery implements the delivery process. The component Billing generates the bills.

Each of these Bounded Contexts requires certain customer data:

- Upon ordering the customer is supposed to be rewarded with points in a bonus program. In this Bounded Context the number of the customer has to be known to the bonus program.
- For Delivery the delivery address and the preferred delivery service of the customer are relevant.
- Finally, for generating the bill the billing address and the tax rate of the customer have to be known.

In this manner each *Bounded Context* has its own model of the customer. This renders it possible to independently change Microservices. If for instance more information regarding the customer is necessary for generating bills, only changes to the *Bounded Context* billing are necessary.

It might be sensible to store basic information concerning the customer in a separate *Bounded Context*. Such fundamental data is probably sensible in many *Bounded Contexts*. To this purpose the *Bounded Contexts* can cooperate (compare below).

A universal model of the customer, however, is hardly sensible. It would be very complex since it would have to contain all information regarding the customer. Moreover, each change to customer information, which is necessary in a certain context, would concern the universal model. This would render such changes very complicated and would probably result in permanent changes to the model.

To illustrate the system setup in the different *Bounded Contexts* a *Context Map* can be used (see section 8.2). Each of the *Bounded Contexts* then can be implemented within one or several Microservices.

Collaboration between *Bounded Contexts*

How are the individual *Bounded Contexts* connected? There are different possibilities:

- In case of a *Shared Kernel* the domain models share some common elements, however, in other areas they differ.
- *Customer/Supplier* means that a subsystem offers a domain model for the caller. The caller in this case is the client who determines the exact setup of the model.
- This is very different in case of *Conformist*: The caller uses the same model as the subsystem, and the other model is thereby forced upon him. This approach is relatively easy – there is no need for translation. One example is

a standard software for a certain domain. The developers of this software likely know a lot about the domain since they have seen many different use cases. The caller can use this model to profit from the knowledge from the modeling.
- The *Anti-corruption Layer* translates a domain model into another one so that both are completely decoupled. This allows the integration of legacy systems without having to take over the domain models. Often data modeling is not very meaningful in legacy systems.
- *Separate Ways* means that the two systems are not integrated, but stay independent of each other.
- In the case of *Open Host Service* the *Bounded Context* offers special services everybody can use. In this way everybody can assemble their own integration. This is especially useful when an integration with numerous other systems is necessary and when the implementation of these integrations is too laborious.
- *Published Language* achieves similar things. It offers a certain domain modeling as common language between the *Bounded Contexts*. Since it is widely used, this language can hardly be changed anymore afterwards.

Bounded Context and Microservices

Each Microservice is meant to model one domain so that new features or changes have only to be implemented within one Microservice. Such a model can be designed based on *Bounded Context*.

One team can work on one or several *Bounded Contexts*, which each serve as foundation for one or several Microservices. Changes and new features are supposed to concern typically only one *Bounded Context* – and thus only one team. This ensures that teams can work largely independently of each other. A *Bounded Context* can be divided into multiple Microservices if that seems sensible. There can be technical reasons for that. For example a certain part of a *Bounded Context* might have to be scaled up to a larger extent than the others. This is simpler if this part is separated into its own Microservice. However, it should be avoided to design Microservices, which contain multiple *Bounded Contexts*, as this entails that several new features might have to be implemented in one Microservice. This interferes with the goal to develop features independently.

Nevertheless, it is possible that a special requirement comprises many *Bounded Contexts* – in that case additional coordination and communication will be required.

The coordination between teams can be regulated via different collaboration possibilities. These influence the independence of the teams as well: *Separate Ways*, *Anti-corruption Layer* or *Open Host Service* offer a lot of independence. *Conformist* or *Customer/Supplier* on the other hand tie the domain models very closely together. For *Customer/Supplier* the teams have to closely coordinate their efforts: The supplier needs to understand the requirements of the customer. For *Conformist* , however, the teams do not need to coordinate: One team defines the model that the other team just uses unchanged. (compare Fig. 9).

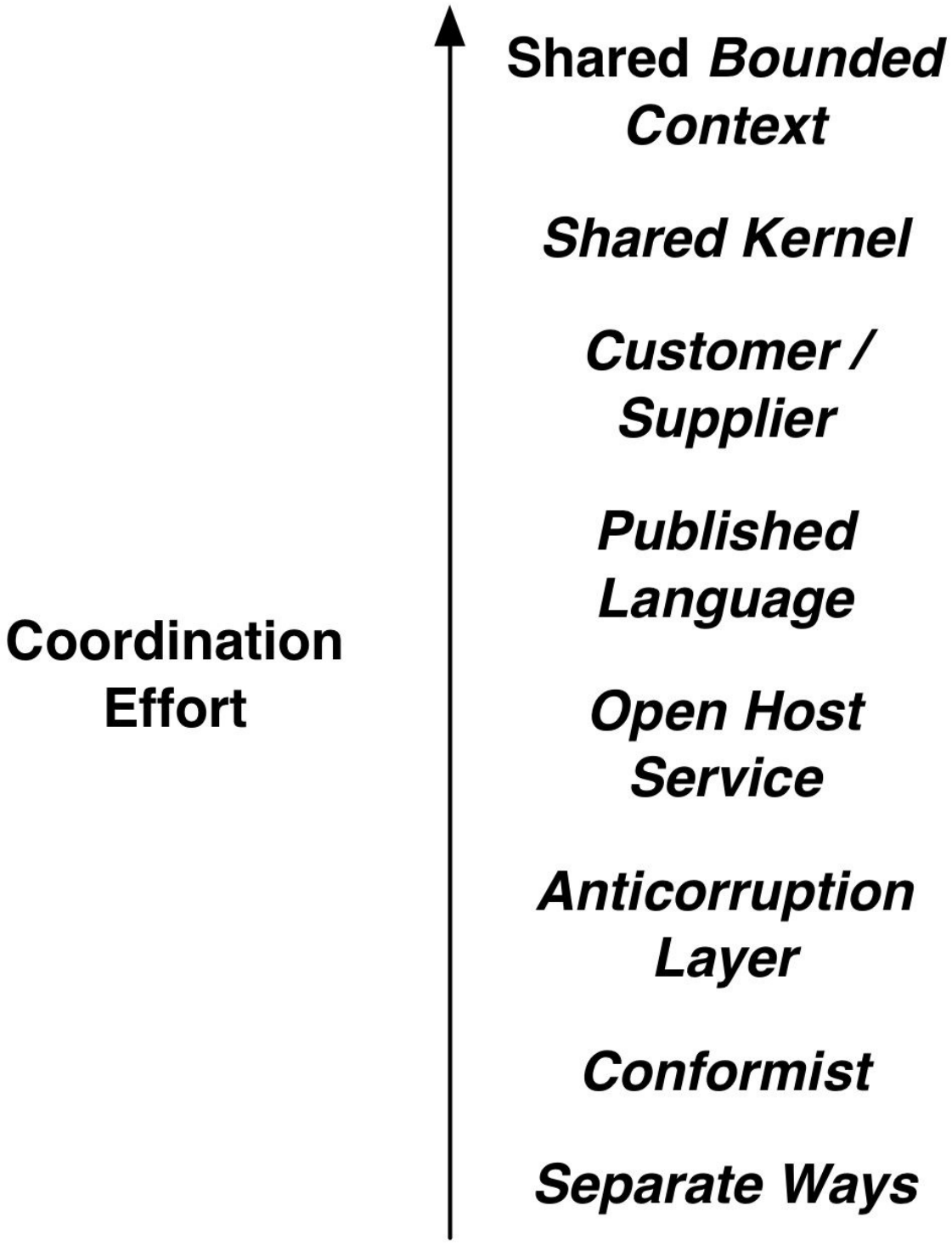


Fig. 9: Communication effort of different collaborations

Like in the case of Conway's Law from section 4.2 it becomes very apparent that organization and architecture are very closely linked. When the architecture enables a distribution of the domains in which the implementation of new features

only requires changes to a defined architecture part, these parts can be distributed to different teams in such a way that these teams can work largely independently of each other. DDD and especially *Bounded Context* demonstrate what such a distribution can look like - and how the parts can work together and how they have to coordinate.

Large-Scale Structure

With Large-Scale Structure DDD also addresses the question how the system in its entirety can be viewed from the different *Bounded Contexts* respectively Microservices.

- A *System Metaphor* can serve to define the fundamental structure of the entire system. For example, an E-commerce system can orient itself according to the shopping process: The customer starts out looking for products, then he/she will compare items, select one item and order it. This can give rise to three Microservices: search, comparison and order.
- *Responsibility Layer* divides the system into layers with different responsibilities. Layers can only call other layers if those are located below them. This does not refer to a technical division into database, UI and logic. In an E-commerce system domain layers might be for example the catalog, the order process and billing. The catalog can call on the order process and the order process can call on the generation of the bill. However, calls into the other direction are not permitted.
- *Evolving Order* suggests not to determine the overall structure too rigidly. It should arise from the individual components in a stepwise manner.

These approaches can provide an idea how the architecture of a system, which consists of different Microservices, can be organized (compare also Chapter 8).

Try and Experiment

Look at a project you know:

- Which *Bounded Contexts* can you identify?
- Generate an overview of the *Bounded Contexts* in a *Context Map*. Compare section 8.2.
- How do the *Bounded Contexts* cooperate? (*Anti-corruption Layer, Customer/Supplier* etc.). Add this information to the *Context Map*.
- Would other mechanisms have been better at certain places? Why?
- How could the *Bounded Contexts* be sensibly distributed to teams so that features are implemented by independent teams?

These questions might be hard to answer as you need to get a new perspective on the system and how the domains are modeled in the system.

Why You Should Avoid a Canonical Data Model (Stefan Tilkov)

by Stefan Tilkov, innoQ

In recent times, I've been involved in a few architecture projects on the enterprise level again. If you've never been in that world, i.e. if you've been focusing on individual systems so far, let me give you the basic gist of what this kind of environment is like: There are lots of meetings, more meetings, and even more meetings; there's an abundance of slide decks, packed with text and diagrams – none of that Presentation Zen nonsense, please. There are conceptual architecture frameworks, showing different perspectives, there are guidelines and reference architectures, enterprise-wide layering approaches, a little bit of SOA und EAI and ESB and Portals and (lately) API talk thrown in for good measure. Vendors and system integrators and (of course) consultants all see their chance to exert influence on strategic decisions, making their products or themselves an integral part of the company's future strategy. It can be a very frustrating, but (at least sometimes) also very rewarding experience: Those wheels are very big and really hard to turn, but if you manage to turn them, the effect is significant.

It's also amazing to see how many of the things that cause problems when building large systems are repeated on the enterprise level. (We don't often make mistakes … but if we do, we make them big!) My favorite one is the idea of establishing *canonical data model* (CDM) for all of your interfaces.

If you haven't heard of this idea before, a quick summary is: Whatever kind of technology you're using (an ESB, a BPM platform, or just some assembly of services of some kind), you standardize the data models of the business objects you exchange. In its extreme (and very common) form, you end up with having just one kind of Person, Customer, Order, Product, etc., with a set of IDs, attributes, and associations everyone can agree on. It isn't hard to understand how that might seem a very compelling thing to attempt: After all, even a non-technical manager will understand that the conversion from one data model to another whenever systems need to talk to each other is a complete waste of time. It's *obviously* a good idea to standardize. Then, anyone who happens to have a model that differs from the canonical one will have to implement a conversion to a and from it just once, new systems can just use the CDM directly, and everyone will be able to communicate without further ado!

In fact, it's a horrible, horrible idea. Don't do it.

In his book on Domain-driven Design, Eric Evans gave a name to a concept that is obvious to anyone who has actually successfully built a larger system: The *Bounded Context*. This is a structuring mechanism that avoids having a single huge model for all of your application, simply because that (a) becomes unmanageable and (b) makes no sense to begin with. It recognizes that a Person or a Contract are different things in different contexts on a *conceptual level*. This is not an implementation problem – it's reality.

If this is true for a large system – and trust me, it is – it's infinitely more true for an enterprise-wide architecture. Of course you can argue that with a CDM, you're only standardizing the interface layer, but that doesn't change a thing: You're still trying to make everyone agree what a concept means, and my point is that you should recognize that not every single system has the same needs.

But isn't this all just pure theory? Who cares about this, anyway? The amazing thing is that organizations are excellent in generating a huge amount of work based on bad assumptions. The CDM (in the form I've described it here) requires coordination between all the parties that use a particular object in their interfaces (unless you trust that someone will be able to just design the right thing from scratch on their own, which you should never do). You'll have meetings with some enterprise architect and a few representatives for specific systems, trying to agree what a customer is. You'll end up with something that has tons of optional attributes because everyone insisted theirs need to be there, and with lots of things

that are kind of weird because they reflect some system's internal restrictions. Despite the fact that it'll take you ages to agree on it, you'll end up with a zombie interface model will be universally hated by everyone who has to work with it.

So is a CDM a universally bad idea? Yes, unless you approach it differently. In many cases, I doubt a CDM's value in the first place, and think you are better off with a different and less intrusive kind of specification. But *if* you want a CDM, here are a number of things you can do to address the problems you'll run into:

- Allow for independent parts to be specified independently. If only one system is responsible for a particular part of your data model, leave it to the people to specify what it looks like canonically. Don't make them participate in meetings. If you're unsure whether the data model they create has a significant overlap with another group's, it probably hasn't.
- Standardize on formats and possibly fragments of data models. Don't try to come up with a consistent model of the world. Instead, create small buildings blocks. What I'm thinking of are e.g. small XML or JSON fragments, akin to microformats, that standardize small groups of attributes (I wouldn't call them business objects).
- Most importantly, don't push your model from a central team downwards or outwards to the individual teams. Instead, it should be the teams who decide to "pull" them into their own context when they believe they provide value. It's not you who's doing the really important stuff (even though that's a common delusion that's attached to the mighty Enterprise Architect title). Collect the data models the individual teams provide in a central location, if you must, and make them easily browsable and searchable. (Think of providing a big elastic search index as opposed to a central UML model).

What you actually need to as an enterprise architect is to get out of people's way. In many cases, a crucial ingredient to achieve this is to create as little centralization as possible. It shouldn't be your goal to make everyone do the same thing. It should be your goal to establish a minimal set of rules that allows people to work as independently as possible. A CDM of the kind I've described above is the exact opposite.

4.4 Microservices with UI?

This book recommends to equip Microservices with a UI. The UI should offer the functionality of the Microservice to the user. In this way all changes in regards to

one functionality can be implemented in one Microservice – regardless of whether they concern the UI, the logic or the database. However, Microservice experts so far have different opinions in regards to the question whether the integration of UI into Microservices is really required. Ultimately, Microservices should not be too large. And when logic is anyhow supposed to be used by multiple frontends, a Microservice consisting of pure logic without UI might be sensible. In addition, it is possible to implement the logic and the UI in two different Microservices, but to have them implemented by one team. This allows to implement features without coordination across teams.

Focusing on Microservices with UI puts the main emphasis on the distribution of the domain logic instead of a distribution by technical aspects. Many architects are not familiar with the domain architecture, which is especially important for Microservices-based architectures. Therefore, a design where the Microservices contain the UI is helpful as a first approach in order to focus the architecture on the domains.

Technical Alternatives

Technically the UI can be implemented as Web UI. When the Microservices have a RESTful-HTTP interface, the Web-UI and the RESTful-HTTP interface are very similar – both use HTTP as protocol. The RESTful-HTTP interface delivers JSON or XML, the Web UI HTML. If the UI is a Single-Page-App, the JavaScript code is likewise delivered via HTTP and communicates with the logic via RESTful HTTP. In case of mobile clients the technical implementation is more complicated. Section 9.1 explains this in detail. Technically a deployable artifact can deliver via an HTTP interface JSON/XML and HTML. In this way it implements the UI and allows other Microservices to access the logic.

Self-Contained System

Instead of calling this approach “Microservice with UI” you can also call it “Self-Contained System” (SCS). SCS define Microservices as having about 100 lines of code, of which there might be more than one hundred in a complete project.

An SCS consists of many of those Microservices and contains a UI. It should communicate with other SCS asynchronously if at all. Ideally each functionality should be implemented in just one SCS and there should be no need for SCSs to communicate with each other. An alternative approach might be to integrate the SCSs at the UI-level.

In an entire system there are then only five to 25 of these SCS. An SCS is something one team can easily deal with. Internally the SCS can be divided into multiple Microservices.

The following definitions result from this reasoning:

- SCS (Self-Contained System) is something a team works on and which represents a unit in the domain architecture. This can be an order process or an registration. It implements a sensible functionality, and the team can supplement the SCS with new features. An alternative name for a SCS is a vertical. The SCS distributes the architecture by domain. This is a vertical design in contrast to a horizontal design. A horizontal design would divide the system into layers, which are technically motivated – for instance UI, logic or persistence.
- A Microservice is a part of a SCS. It is a technical unit and can be independently deployed. This conforms nearly with the Microservice definition put forward in this book. Only the size given in the SCS world rather correspond to what this book denotes as Nanoservices see chapter 15.
- This book refers to Nanoservices as units, which are still individually deployable, but which make technical trade-offs in some areas to further reduce the size of the deployment units. For that reason, Nanoservices do not share all technical characteristics of Microservices.

SCS inspired the definition of Microservices as put forward in this book. Still there is no reason not to separate the UI into a different artifact in case the Microservice gets otherwise too large. Of course, it is more important that the Microservice is small and thus maintainable than to integrate the UI. But UI and logic should at least be implemented by the same team.

4.5 Conclusion

Microservices are a modularization approach. For a deeper understanding of Microservices the different perspectives discussed in this chapter are very helpful:

- Section 4.1 focused on the size of Microservices. But a closer look revealed that the size of Microservices itself is not that important, even though there are influencing factors. However, this perspective provided a first impression on what a Microservice should be. Team size, modularization and replaceability of Microservices each determine an upper size limit. The

lower limit is determined by transactions, consistency, infrastructure and distributed communication.

- Conway's Law (section 4.2) shows that architecture and organization of a project are closely linked – they are nearly synonymous. Microservices can further improve the independence of teams and thus ideally support architectural designs, which aim at the independent development of functionalities. Each team is responsible for a Microservice and therefore for a certain part of a domain so that the teams are largely independent concerning the implementation of new functionalities. Thus, in regards to domain logic there is hardly any need for coordination across teams. The requirement for technical coordination can likewise be reduced to a minimum due to the possible technical independence.
- In section 4.3 Domain-driven Design provides a very good impression as to what the distribution of domains in a project can look like and how the individual parts can be coordinated. Each Microservice can represent a *Bounded Context*. This is a self-contained piece of domain logic with an independent domain model. Between the *Bounded Contexts* there are different possibilities for collaboration.
- Finally section 4.4 demonstrated that Microservices should contain a UI to be able to implement the changes for a functionality really within an individual Microservice. This does not necessarily have to be a deployment unit, however, UI and Microservice should be in the responsibility of one team.

Together these different perspectives provide a balanced picture of what constitutes Microservices and how they can function.

Essential Points

To put it differently: A successful project requires three components:

1. An organization: This is supported by Conway's Law.
2. A technical approach: This can be Microservices.
3. A domain design as offered by DDD and *Bounded Context*.

The domain design is especially important for the long-term maintainability of the system.

Try and Experiment

Look at the three approaches for defining Microservices: size, Conway's Law and Domain-driven Design.

Section 1.2 showed the most important advantages of Microservices. Which of the goals to be achieved by Microservices are best supported by the three definitions? DDD and Conway's Law lead for instance to a better time-to-market.

Which of the three aspects is in your opinion the most important? Why?

1. Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley,2003, ISBN 978-0-32112-521-7↩

5 Reasons for Microservices

Microservices offer many advantages. These are presented in this chapter. A detailed understanding of the advantages allows a better evaluation whether Microservices represent a sensible approach in a given use case. The chapter continues the discussion of section 1.2 and explains the advantages in more detail.

Section 5.1 depicts the technical advantages of Microservices. However, Microservices also influence the organization, as described in section 5.2. Finally, section 5.3 addresses the advantages from a business perspective.

5.1 Technical Benefits

Microservices are an effective modularization concept. Only with distributed communication it is possible to call another Microservice. This does not happen by accident, but a developer has to create the respective possibilities for it within the communication infrastructure. Consequently, dependencies between Microservices do not just creep in unintendedly, but a developer has to generate them explicitly. Without Microservices it easily happens that a developer just uses some class and thereby creates a dependency, which had not been architecturally intended.

Let us assume for instance that in an E-commerce application the product search should be able to call the order process, but not the other way round. This ensures that the product search can be changed without influencing the order process, as the product search does not use the order process. Now a dependency of the product search to the order process is introduced, for instance, because a developer found functionalities there, which were useful for him. Consequently, product search and order process now depend on each other and can only be changed together.

Once undesired dependencies have started to creep into the system, additional dependencies rapidly accrue. The application architecture erodes. This development can normally only be prevented by architecture management tools. Such tools have a model of the desired architecture and discover when a developer has introduced an undesired dependency. The developer then can

immediately remove the dependency again before harm is done and the architecture suffers. Appropriate tools are presented in section 8.2.

In a Microservices-based architecture product search and order process would be separate Microservices. To create a dependency the developer would have to implement it within the communication mechanisms. This is laborious and thus normally does not happen unnoticed, even without architecture management tools. Thus the probability is lower that the architecture erodes on the level of dependencies between Microservices. The Microservice boundaries act like firewalls, which prevent an architecture erosion. Microservices offer a strong modularization as it is difficult to overstep the boundaries between modules.

Replacing Microservices

Working with old software systems poses a big challenge: A further development of the software is difficult due to bad code quality. To replace the software is risky. Often it is unclear how exactly the software is working, and the system is very large. The larger the software system the more laborious is its replacement. When the software is in addition supporting important business processes, it is nearly impossible to change it. The failure of such business processes can have tremendous consequences, and each software change entails the danger of such a failure.

Although this is a central problem, a software architecture is never really aimed at replacing a software. However, Microservices support this goal: They can be replaced individually since they are separate and small deployment units. Therefore, the technical prerequisites for a replacement are better. Eventually it is not necessary to replace a large software system, but only a small Microservice. Whenever necessary, additional Microservices can be replaced.

In case of the new Microservices the developers are not tied to the old technology stack, but free to use other technologies at will. When the Microservice additionally is independent in a domain sense, the logic is easier to understand. The developer does not need to understand the entire system, but just the functionalities of an individual Microservice. Knowledge regarding the domain is a prerequisite for the successful replacement of a Microservice.

Moreover, Microservices keep functioning when another Microservice fails. Even if the replacement of a Microservice leads to the temporary failure of one

Microservice, the system as such can keep operating. This additionally decreases the risk associated with a replacement.

Sustainable Software Development

The start in a new software project is simple: There is not much code yet, the code structure is clean, and the developers make fast progress. Due to architecture erosion and an increasing complexity development can get more difficult over time. At some point, the software turns into a legacy system. As already discussed, Microservices prevent architecture erosion. When a Microservice has turned into a legacy system, it can be replaced. For these two reasons Microservices make a sustainable software development possible. This means that a high productivity can be reached also on the long-term. However, also in a Microservice-based system it can happen that a lot of code has to be newly written. This will of course decrease productivity.

Handling Legacy

Replacing Microservices is only possible if the system is already implemented in a Microservice-based manner. However, also the replacement and amending of existing legacy applications is easier with Microservices. The legacy applications only have to provide an interface, which enables the Microservice to communicate with the legacy application. Comprehensive code changes or the integration of new code components into the legacy system are not necessary. The code level integration is a big challenge in the case of legacy systems, which can be avoided in this manner. Amending the system is especially easy when a Microservice can intercept the processing of all calls and process them itself. Such calls can be HTTP requests for the built-up of web sites or REST calls.

In these instances, the Microservice can complement the legacy system. There are different possibilities for this:

- The Microservice can process certain requests by itself while leaving the others to the legacy system.
- Alternatively, the Microservice can change the requests and afterwards transfer them to the actual application.

This approach is similar to the SOA approach (compare Chapter 7), which deals with the comprehensive integration of different applications. When the applications are distributed into services, these services cannot only be

orchestrated anew, likewise it is possible to replace individual services for instance by Microservices.

An Example for Microservices and Legacy

In a project the goal was to undertake a modernization in an existing Java-E-commerce application. For this purpose, new technologies, for example new frameworks, were to be introduced to enhance future software development productivity. After some time, it turned out that the effort for the integration of the new and old technologies would be tremendous. The new code had to be able to call the old one – and vice versa. This requires technology integration in both directions. Transactions and database connections have to be used jointly. Likewise, the security mechanisms have to be integrated. This integration would also render the development of the new software more complicated and thus endanger the goal of the undertaking.

Fig. 10 shows the solution: The new system was developed completely independent of the old system. The only integration was provided by links, which call certain behaviors in the old software – for instance the addition of items to the shopping cart. The new system also had access to the same database like the old system. In hindsight, a shared database is not a good idea as the database is an internal representation of the data of the old system. When this representation is placed at the disposal of another application, the principle of encapsulation is violated (compare section 10.1). The data structures can hardly be changed anymore as now in addition to the old system also the new system depends on them.

The approach to develop the system separately solved the integration-related problems to a large extent. First of all, developers thereby could use new technological approaches without having to consider the old code and the old approaches. This enabled much more elegant solutions.

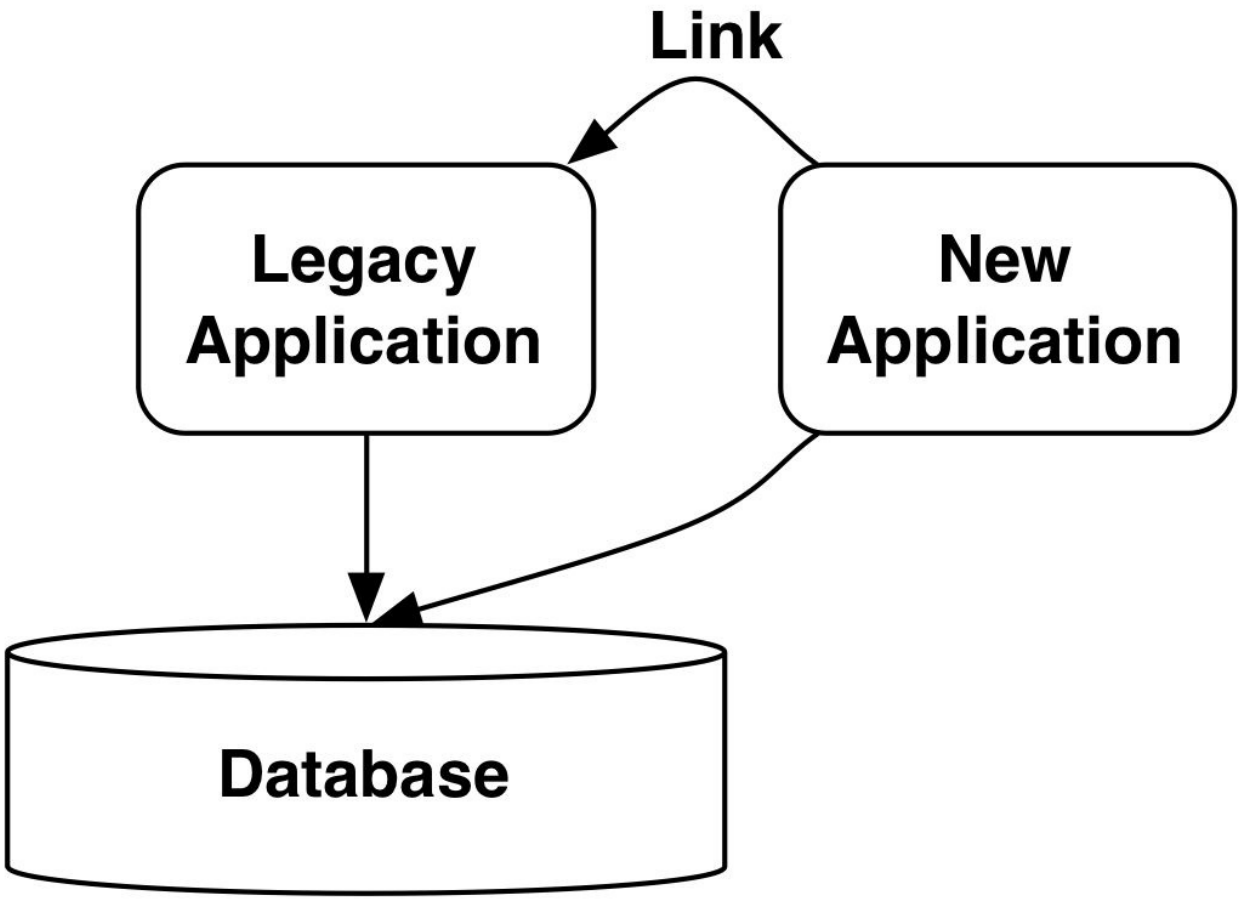


Fig. 10: Example for legacy integration

Continuous Delivery

Continuous Delivery brings software regularly into production thanks to a simple, reproducible process. This is achieved by a Continuous Delivery pipeline (compare Fig. 11):

- In the commit phase the software is compiled, the unit tests are run, and static code analysis might be performed.
- The automated acceptance tests of the next phase ensure that the software is correct concerning the business requirements so that it would be accepted by the customer.
- Capacity tests check whether the software is sufficiently performant to support the expected number of users. These tests are automated as well.
- Explorative tests on the other hand are performed manually and serve to test certain areas of the system such as new features or certain aspects like software security.

- Finally, the software is brought into production. This process is ideally also automated.

Software is promoted through the individual phases: It traverses the individual phases consecutively. For example, a release can successfully pass the acceptance tests. However, the capacitance tests reveal that the software does not meet the requirement regarding the expected load. In this case the software is never going to be promoted to the remaining phases such as explorative tests or even production.

Fig. 11: Continuous Delivery pipeline

A Continuous Delivery pipeline with a full automation is the optimum. However, somehow all software gets into production. Accordingly, the current process can be optimized in a stepwise manner.

Continuous Delivery is especially easy to realize with Microservices. Microservices are independent deployment units. Consequently, they can be brought into production independently of other services. This has tremendous effects onto the Continuous Delivery pipeline:

- The pipeline is faster as only a small Microservice has to be tested and brought into production at one time. This accelerates feedback. Rapid feedback is an essential goal of Continuous Delivery. When it takes weeks for a developer to get to know that his/her code has caused a problem in production, it will be difficult to become acquainted with the code again and to analyze the problem.
- The risk of deployment decreases. The deployed units are smaller, besides Microservice-based systems can even still be used if a number of Microservices fail. And the deployment can more easily be rolled back.
- Measures to further decrease the risk are also easier to implement with smaller deployment units. In case of Blue/Green Deployment for instance a new environment is built up with the new release. This is similar for Canary Releasing: In the case of this approach at first only one server is provided with the new software version. Only when this server runs successfully in production, the new version is rolled out to the other servers. For a Deployment Monolith this approach can be hard or nearly impossible to implement as it requires a lot of resources for the large number of

environments. In case of Microservices the required environments are much smaller and the procedure thus easier.
- Test environments pose additional challenges. When for instance a third party system is used, the environment has to contain also a test version of this third system. In case of smaller deployment units, the demands to the environments are lower. The environments for Microservices only have to integrate the third systems, which are necessary for the individual Microservice. It is likewise possible to test the systems using mocks of the third systems. This facilitates the tests and represents also an interesting method in order to test Microservices independently of each other.

Continuous Delivery is one of the most important arguments for Microservices. Many projects invest in migrating to Microservices in order to facilitate the creation of a Continuous Delivery pipeline.

However, Continuous Delivery is also a prerequisite for Microservices. Without Continuous Delivery pipelines the many Microservices can hardly be brought into production since it is not feasible to bring so many Microservices into production manually. Thus Microservices profit from Continuous Delivery and vice versa.

Scaling

Microservices offer via the network reachable interfaces, which can be accessed for instance via HTTP or via a message solution. Each Microservice can run on one server – or on several. When the service runs on several servers, the load can be distributed onto the different servers. Likewise, it is possible to install and run Microservices on computers having different performance. Each Microservice can implement its own scaling.

In addition, caches can be placed in front of Microservices. For REST-based Microservices it can be sufficient to use a generic HTTP cache. This reduces the effort for such a cache significantly. The HTTP protocol contains a comprehensive support for caching, which is very helpful in this context.

Furthermore, it might be possible to install the Microservices at different locations within the network in order to bring them closer to the caller. In case of world-wide distributed Cloud environments, it does not matter anymore in which computing center the Microservices are running. When the Microservice infrastructure uses several computing centers and processes calls always in the nearest computing center, the architecture can significantly reduce the response

times. Besides, static content can be delivered by a CDN (Content Delivery Network), whose servers are located even closer to the users.

However, the better scaling and the support for caching cannot work miracles: Microservices result in a distributed architecture. Calls via the network are a lot slower than local calls. From a pure performance perspective it might be better to combine several Microservices or to use technologies which focus on local calls (compare chapter 15).

Robustness

Actually, Microservices should be less reliable than other architecture approaches. After all, Microservices are a distributed system. Thus possible network failures add to the usual sources of errors. Moreover, Microservices run on several servers so that there is also a larger probability for hardware failure.

To ensure a high availability, the Microservices-based architecture has to be appropriately designed. The communication between the Microservices has to form a kind of firewall: The failure of a Microservice may not propagate. This prevents that a problem arising in an individual Microservice causes a failure of the complete system.

Accordingly, the Microservice, which is calling, has to somehow keep working upon a failure. One possibility is to assume default values. Alternatively, the failure might lead to a graceful degradation i.e. a somehow reduced service.

It can already be decisive how a failure is dealt with technically: The operation system level timeout for TCP/IP connections is often set to five minutes, for example. If due to the failure of a Microservice requests run into this timeout, the thread is blocked for five minutes. At some point all threads will be blocked. If that happens, the calling system might fail as it cannot do anything else anymore than wait for timeouts. This can be avoided by supplying the calls with shorter timeouts. Such ideas are around much longer than the concept of Microservices. The book “Release It” [1] in detail presents such challenges and approaches for solving them. When these approaches are implemented, Microservice-based systems can tolerate the failure of entire Microservices and thus become more robust than a Deployment Monolith.

In comparison to Deployment Monoliths Microservices have the additional advantage that they distribute the system into multiple processes. These processes

are better isolated from each other. In a Deployment Monolith, which only starts one process, memory leaks or a functionality using up a lot of computing resources can make the whole system fail. Such errors are very often simple programming mistakes or slips. The distribution into Microservices prevents such situations as only a single Microservice would be failing in such a scenario.

Free Technology Choice

Microservices offer technological freedom. Since Microservices communicate only via the network, they can be implemented in any language and platform as long as communication with other Microservices is possible. This free technology choice can be used to test new technologies without running big risks. As a test one can use the new technology in a single Microservice. If the technology does not perform according to expectations, only this one Microservice has to be rewritten. In addition, troubles arising in case of failure will be limited. The free technology choice offers for instance the advantage that developers can really use new technologies in production. This increases motivation and has positive effects on personnel recruitment as developers normally enjoy to use new technologies.

Moreover, in this way the most appropriate technology can be used for each problem. A different programming language or a certain framework can be used to implement specific system parts. It is even possible for an individual Microservice to use a specific database or persistence technology. However, backup and disaster recovery mechanisms have to be implemented for that.

Free technology is an option – it does not have to be made use of. Technologies can also be defined for all Microservices in a project so each Microservice is bound to a specific technology stack. However, Deployment Monolith inherently narrow the choices developers have: For example, in Java applications each library can only be used in one version. Accordingly, not only the libraries to be used, but even the versions have to be set in a Deployment Monolith. Microservices do not impose such technical limitations.

Independence

Decisions regarding technology and putting new versions into production concern only individual Microservices. This makes Microservices very independent of each other. Of course, there has to be a common technical basis. The installation of Microservices should be automated, there should be a Continuous Delivery pipeline for each Microservices, and Microservices should adhere to the monitoring specifications. However, within these parameters Microservices can

implement a practically unlimited choice of technical approaches. Due to the greater technological freedom there is less coordination necessary between Microservices.

5.2 Organizational Benefits

Microservices are an architectural approach and thus should have only advantages for software development and structure. However, due to Conway's Law (compare section 4.2) architecture affects also team communication and thus organization.

First of all Microservices reach a high level of technical independence as the last section (5.1) discussed. When within the organization a team is in full charge of a Microservice, the team can make full use of the technical independence. However, the team has also the full responsibility if a Microservice malfunctions or fails in production.

In this manner Microservices support team independence. The technical basis allows teams to work on the different Microservices with little coordination. This provides the fundament for the independent work of the teams.

In other projects, technology or architecture have to be decided centrally since the individual teams and modules are bound to these decisions due to the technical frame conditions. It might just be impossible to use two different libraries or even two different versions of one library within one Deployment Monolith. Thus, central coordination is mandatory. For Microservices, the situation is different. This allows for self organization. However, a global coordination might still be sensible, for instance to be able to perform an update including all components in case of a security problem with a library.

Teams have more responsibilities: They decide the architecture of their Microservices. They cannot hand over this responsibility to a central architecture. Thus, they also have to carry the consequences since they have the responsibility for the Microservice.

The Scala Decision

In a project employing a Microservice-based approach the central architecture group was supposed to decide whether Scala could be used as programming language by one team. This decision would have transferred the responsibility for the decision to the central architecture group. The group would have had to decide whether the team might solve its problems more

efficiently by using Scala or whether the use of Scala might create additional problems in the end. Eventually, the decision was delegated to the team since the team has to take responsibility for its Microservice. They have to deal with the consequences, if Scala in the end does not fulfill the demands of production or does not support an efficient software development. They have the investment of getting familiar with Scala first and have to estimate whether this effort will pay off in the end. Likewise, they have a problem if suddenly all Scala developers leave the project or change to another team. To delegate the responsibility to the central architecture group is strictly speaking not even possible since the central architecture group is not directly affected by the consequences. Therefore, the team just has to decide by itself. The team has to include all team members into the decision – also the Product Owner, who would for instance suffer in the end in case of a low productivity.

This line of action represents a radical renunciation of old forms of organization, where the central architecture group prescribes the technology stack to be used for everybody. In this type of organization the individual teams are not responsible for decisions and non functional requirements like availability, performance or scalability. In a classical architecture, the non functional properties can only be provided for centrally since they can only be warranted by the common basis of the entire system. When Microservices do not force a common basis anymore, these decisions can be distributed to the teams thus enabling a greater self-reliance and independence.

Smaller projects

Finally, Microservices allow for the distribution of large projects into numerous small projects as the individual Microservices are so independent that a central coordination loses importance. Therefore, a comprehensive project organization is not necessary anymore. Large organizations are problematic as they have a relatively large communication overhead. When Microservices enable the fragmentation of a large organization into several smaller ones, the need for communication decreases. This allows teams to focus more on the implementation of requirements.

Large projects will also fail more frequently. Also from this perspective it is better when a large project can be divided into multiple smaller projects. The smaller extent of the individual projects enables more precise estimations. Better estimations improve planning and decrease risk. And even if the estimation is wrong, the impact of the incorrect decisions is lower. In conjunction with the greater flexibility this can speed up and facilitate the process of decision making – especially as the associated risk is so much smaller.

5.3 Benefits from a Business Perspective

The already discussed advantages from an organizational perspective lead also to business advantages: The projects have a lower risk, and coordination between teams needs to be less intense so that the teams can work more efficiently.

Parallel Work on Stories

The distribution into Microservices enables the parallel work on different stories (compare Fig. 12). Each team works on a story, which only concerns their own Microservice. Consequently, the teams can work independently, and the system as such can be simultaneously expanded at different spots. This eventually scales the agile process. However, scaling does not take place at the level of development processes, but is facilitated by the architecture and the independence of the teams. Changes and deployments of individual Microservices are possible without complex coordination. Therefore, teams can work independently. When a team is slower or encounters obstacles, this does hardly influence the other teams. Thus the risk associated with the project is further reduced.

An unambiguous domain-based design and the assignment of one developer team per Microservice can scale the development or project organization with the number of teams.

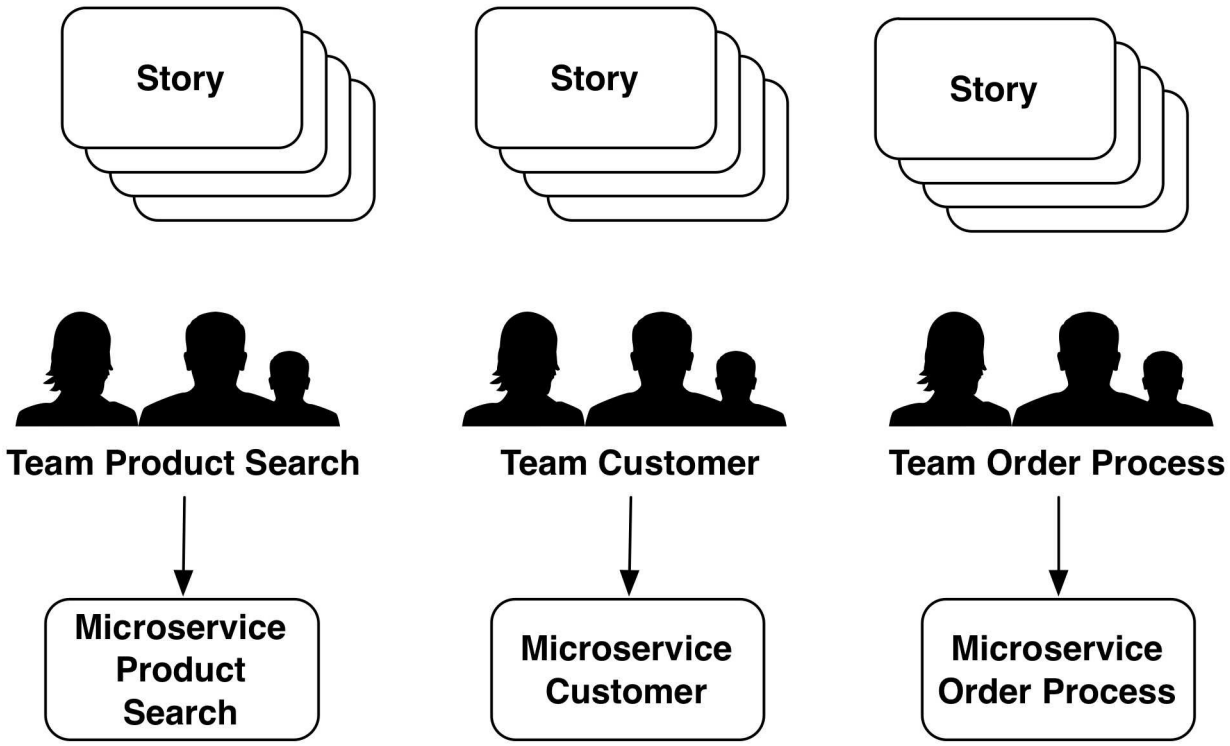


Fig. 12: Example for legacy integration

It is possible that changes concern several Microservices and thus several teams. An example: Only certain customers are allowed to order some products – for instance because of youth protection. In case of the architecture depicted in Fig. 12 changes to all Microservices would be necessary to implement this feature. The Customer Microservice would have to store the data whether a customer is of legal age. Product search should hide or label the products prohibited for underage customers. Finally, the order process has to prevent the ordering of prohibited products by underage customers. These changes have to be coordinated. Coordination is especially required when one Microservice calls another. In that case the called upon Microservice has to be changed first so that the caller can afterwards use the new features.

This problem can certainly be solved. One can reason that the outlined architecture is not optimal. If the architecture is geared to the business processes, the changes can be limited to the order process. Eventually, only the ordering is to be prohibited, not searching. The information whether a certain client is allowed to order or not should also be within the responsibility of the order process. Which architecture and consequently which team distribution is the right one, depends on the requirements and the concerned Microservices and teams.

If the architecture has been selected appropriately, Microservices can well support agility. This is for sure a good reason from a business perspective to use a Microservice-based architecture.

5.4 Conclusion

In summary Microservices lead to the following technical advantages (section 5.1):

- **Strong modularization**: Dependencies between Microservices cannot easily creep in.
- Microservices can be **easily replaced**.
- The strong modularization and the replaceability of Microservices leads to a **sustained speed of development**: The Architecture remains stable, and Microservices, which cannot be maintained anymore, can be replaced. Thus, the quality of the system remains high also on the long run so that the systems stays maintainable.
- **Legacy systems** can be supplemented with Microservices without the need to carry around all the ballast, which has accumulated in the legacy system. Therefore, Microservices are a good approach when dealing with legacy systems.
- Since Microservices are small deployment units, a **Continuous Delivery pipeline** is much easier to set up.
- Microservices can be **scaled** independently.
- If Microservices are implemented in line with established approaches, the system will be more **robust** in the end.
- Each Microservice can be implemented in a different programming language and with a different **technology**.
- Therefore, Microservices are largely **independent** from each other on a technical level.

The technical independence affects the organization (section 5.2): The teams can work independently and on their own authority. There is less need for central coordination. Large projects are replaced by a collection of small projects, which positively affects risk and coordination.

From a business perspective just the effects on risk are already positive (section 5.3). However, is is even more attractive that the Microservice-based architecture

enables the scaling of agile processes without requiring an excessive amount of coordination and communication.

Essential Points

- There are numerous technical advantages – ranging from scalability and robustness to sustainable development.
- The technical independence results in advantages on the organizational level. Teams become independent.
- The technical and organizational advantages taken together result in advantages at the level of business: a lower risk and a faster implementation of more features.

Try and Experiment

Look at a project you know:

Why are Microservices useful in this scenario? Evaluate each advantage by assigning points (1 = no real advantage; 10 = very large advantage). The possible advantages are listed in the conclusion of this chapter.

What would the project look like with or without the use of Microservices?

Develop a discussion of the advantages of Microservices from the perspective of an architect, a developer, a project leader and the customer for the project. The technical advantages will be more of interest for the developers and architects, while the organizational and business advantages matter more for project leaders and customers. Which advantages do you put most emphasis on for the different groups?

Visualize the current domain design in your project or product.

- Which teams are responsible for which parts of the project? Where do you see overlap?
- What should the distribution of teams to product parts and services look like to achieve a largely independent mode of operation?

1. Michael T. Nygard: Release It!: Design and Deploy Production-Ready Software, Pragmatic Programmers, 2007, ISBN 978-0-97873-921-8↩

6 Challenges

The distribution of a system into Microservices entails a higher complexity. This leads to challenges at the technical level (compare section 6.1) – for instance high latency times in the network or the failure of individual services. However, also at the level of software architecture there are a number of things to consider – for instance because of the bad architecture changeability (section 6.2). And finally, there are many more components to be independently delivered so that operation and infrastructure become more complex (section 6.3). These challenges have to be dealt with when introducing Microservices. Measures described in the following chapters show how to appropriately handle these challenges.

6.1 Technical Challenges

Microservices are distributed systems. Calls between Microservices go via the network. This affects the latency and thus the response times of Microservices negatively. The already mentioned first rule for distributed objects states that objects, if possible, should not be distributed (compare section 4.1).

The reason for that is illustrated in Fig. 13 A call has to go via the network to reach the server, is processed there and has to return to the caller. The latency just for network communication can be around 0.5 ms in a computing center (compare here). Within this time a processor running at 3 Ghz can process about 1.5 million instructions. When a computation is redistributed to another node, it should be checked whether local processing of the request might not be faster. The latency can even increase further by parameter marshaling and unmarshaling for a call and the result of a call. On the other hand, network optimizations or connecting nodes to the same network switch can improve the situation.

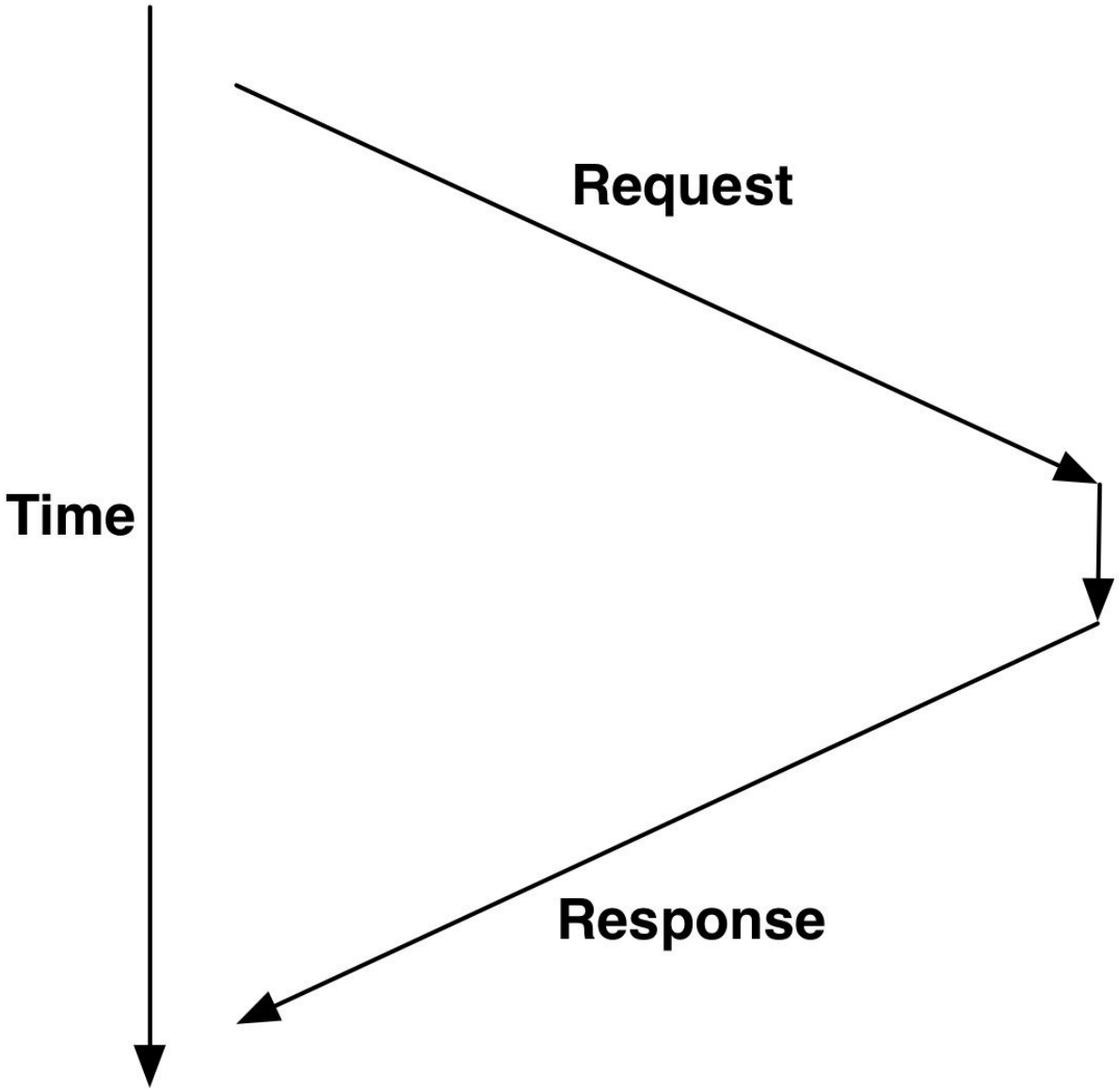


Fig. 13: Latency for a call via the network

The first rule for distributed objects and the warning to be aware of the latency within the network dates back to the time when CORBA and EJB were used. These technologies were often used for distributed Three Tier Architectures (compare Fig. 14). For every client request the web tier implements only the data rendering as HTML page. The logic resides on another server, which is called via the network. The data are deposited in the database and thus on an again other server. When only data are to be shown, there is little happening in the Middle Tier. The data are not processed, just forwarded. For performance and latency, it would be much better to keep the logic on the same server as the web tier. Though

the distribution allows to scale the Middle Tier independently, the system does not get faster this way when it anyhow does not have much to do.

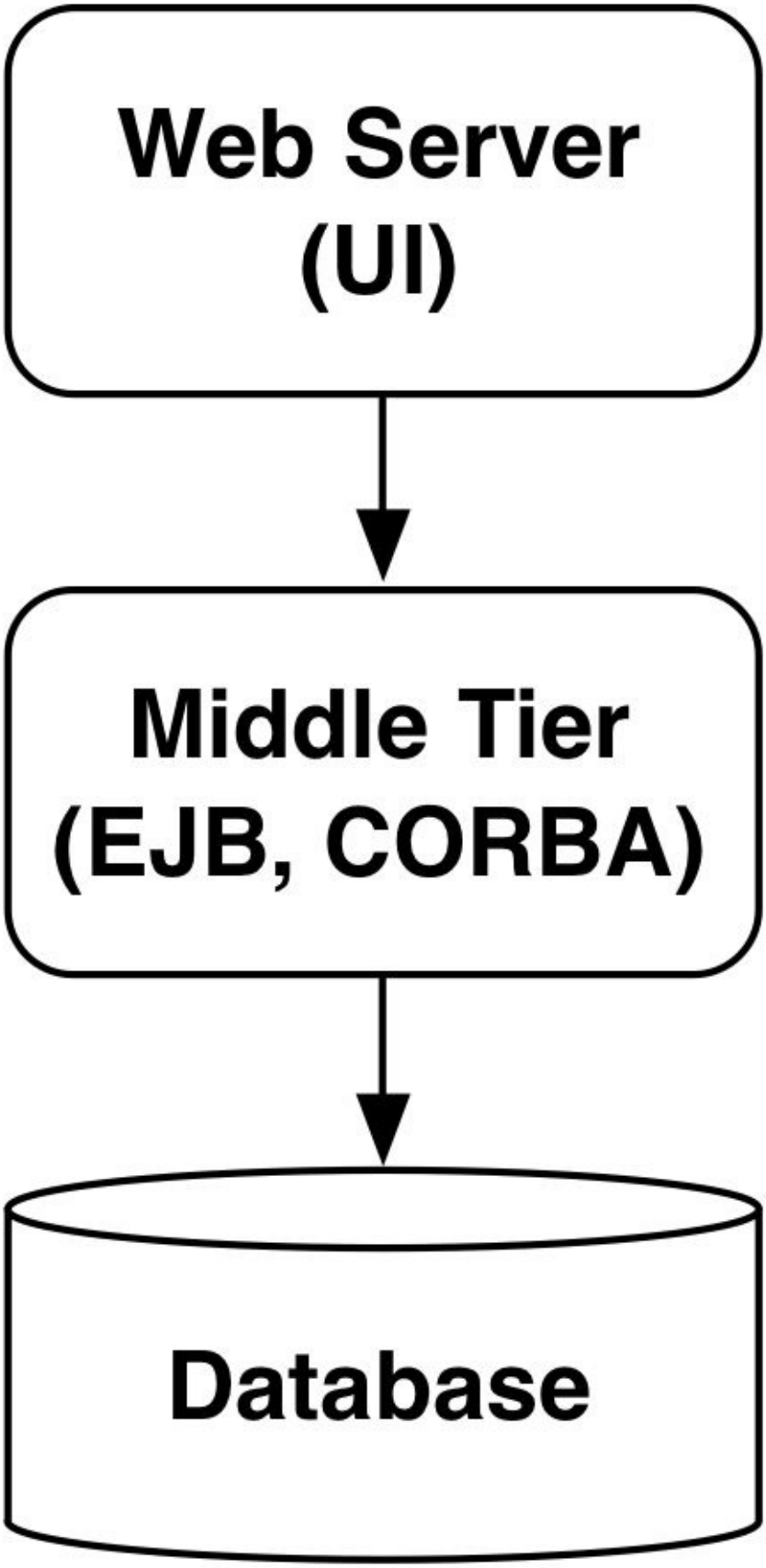

Web Server
(UI)
Middle Tier
(EJB, CORBA)
Database

Fig. 14: Three Tier Architecture

For Microservices the situation is different as the UI is contained in the Microservice. Calls between Microservices only take place when Microservices need functionalities of other Microservices. If that is often the case, this might be a hint that there are architectural problems as the Microservices should be largely independent of each other.

Practically, Microservice-based architectures function in spite of the challenges related to distribution. Still Microservices should not communicate too much with each other in order to improve performance and reduce latency.

Code Dependencies

The great advantage of Microservice-based architectures is the option to independently deploy the individual services. However, this option can be undone by code dependencies. If a library is used by several Microservices and a new version of this library is supposed to be rolled out, a coordinated deployment of several Microservices might be required – precisely the scenario that should be prevented. Something like that can for instance easily occur due to binary dependencies where sometimes different versions are not compatible anymore. The deployment has to be temporally coordinated in a way that all Microservices are rolled out in a certain time interval and in a defined order. Besides the code dependency has to be changed in all Microservices, a process that has likewise to be prioritized and coordinated across all involved teams. A binary level dependency is a very tight technical coupling, which entails a very tight organizational coupling.

Therefore Microservices propagate a Shared Nothing Approach where the Microservices do not possess shared code. Microservices rather accept code redundancy and resist the urge to reuse code in order to avoid a close organizational link.

Code dependencies can be tolerable in certain situations. When a Microservice offers for instance a client library, which supports callers while using this Microservice, this does not necessarily have negative consequences. The library depends on the interface of the Microservice. If the interface is changed in a backward compatible manner, also a caller having an old version of the client library can still use the Microservice. The deployment remains uncoupled. However, the client library can be the starting point to a code dependency. If the

client library contains for instance domain objects, this can be a problem. In fact, if the client library contains the same code for the domain objects, which is also internally used, changes to the internal model will affect the clients. They might have to be deployed again if need be. If the domain object contains logic, this logic can only be modified when all clients are likewise deployed anew. This also violates the idea of independently deployable Microservices.

Consequences of Code Dependencies

Here is an example for the effects of code dependencies: User authentication is a central function, which all services use. A project has developed a service implementing the authentication. Nowadays there are open source projects, which implement such things, (section 8.12) so that an implementation of a home-grown solution is rarely sensible anymore. In that project each Microservice could use a library for facilitating the handling of the authentication service. Accordingly, all Microservices have a code dependency towards the authentication service. Changes to the authentication service might require that the library has to be newly rolled out. This in turn means that all Microservices have to be modified and newly rolled out as well. In addition, the deployments of the Microservices and the authentication service have to be coordinated. This can easily cost a two-digit number of man days. Thus authentication can hardly be changed anymore due to the code dependency. If the authentication service could be deployed just like that and if there were no code dependencies, which couple the deployment of the Microservices and the authentication service, the problem would be solved.

Unreliable Communication

Communication between Microservices occurs via the network and is therefore unreliable. In addition, Microservices can fail. To prevent that a failure of the entire system ensues, the remaining Microservices in such a case have to compensate for the failure of the malfunctioning Microservice and keep being available. However, to achieve this goal the quality of the services has to be degraded i.e. by using default values or limiting the useable functionality (section 10.5).

This problem cannot be completely solved on a technical level: The Microservice availability can for instance be optimized by highly available hardware. But this increases costs. Besides, it is no complete solution: In some respects, it even increases risk. If the Microservice fails despite highly available hardware and the failure propagates across the entire system, a complete failure of the entire system occurs. Thus, the Microservices should rather compensate the failure of another Microservice.

In addition, the threshold between a technical and a domain problem is crossed. An ATM might serve as example: When the ATM cannot retrieve the account balance of the customer, there are two possibilities. The ATM can refuse the withdrawal. Although this is a safe option, it will anger the customer and decrease revenue. Alternatively, the ATM can hand out the money – maybe up to a certain upper limit. Which alternative should be implemented, is a business decision. Eventually, somebody has to decide whether it is preferable to be on the safe side, even if it means to forego some revenue and anger customers, or to run a certain risk to pay out too much money.

Technology Pluralism

The technology freedom of Microservices can result in a project using many different technologies. The Microservices do not need to have a shared technology basis. Accordingly, the complexity of the whole system increases. Each team masters the technologies, which are used in its own Microservice. However, the large number of used technologies and approaches can cause the system as such to reach a level of complexity no individual developer or team can understand anymore. But often such a general understanding is not necessary since each team only needs to understand its own Microservice. Whenever it becomes necessary to have a look at the entire system - be it even only from a certain limited perspective as for instance operations –, the complexity might pose a problem. In such cases, unification can be a sensible counter measure. This does not mean that the technology stack has to be completely uniform, but that certain parts should be uniform or that the individual Microservices should behave in a uniform manner. For instance, a uniform logging framework might be defined or a uniform format for logging, which different logging frameworks might implement differently. Alternatively, a common technical basis like the JVM (Java Virtual Machine) can be decided upon for operational reasons without setting the programming languages.

6.2 Architecture

The architecture of a Microservice-based system distributes the domain-based functionalities among the Microservices. To understand the architecture at this level dependencies and communication relationships between the Microservices have to be known. Analyzing communication relationships is difficult. For large Deployment Monoliths there are tools, which read source code or even only the executable system. Based on this the tools can generate graphs visualizing modules and relationships. This makes it possible to verify the implemented

architecture, adjust it in regards to the planned architecture and to follow the architecture evolution over time. Such overviews are central for architectural work, however, difficult to generate in the case of Microservices as the respective tools are lacking – but there are solutions. Section 8.2 discusses these in detail.

Architecture = Organization

Microservices are based on the idea that organization and architecture are the same. Microservices exploit this circumstance to implement the architecture. The organization is structured in a way, which renders the architecture implementation especially easy. However, this means that an architecture refactoring can entail changes to the organization. This renders architectural changes more difficult. This is not only a problem of Microservices: Conway's Law (section 4.2) applies to all projects. However, other projects often are not aware of the law and its implications. Therefore, they do not use the law productively and cannot estimate the organizational problems caused by architectural changes.

Architecture and Requirements

The architecture influences also the independent development of individual Microservices and the independent streams of stories. When the domain-based distribution of Microservices is not optimal, requirements might not only influence one team and one Microservice, but several. In such cases a larger amount of coordination is necessary between the different teams and Microservices. This influences the productivity negatively and thus undoes one of the essential reasons for the introduction of Microservices.

In case of Microservices the architecture influences not only the software quality, but also the organization and the independent work of the teams and thereby the productivity. Designing an optimal architecture gets even more important since mistakes have far reaching consequences.

Many projects do not pay sufficient attention to domain architecture, often much less than to technical architecture. Besides, most architects are not as experienced with domain architecture as with technical architecture. These circumstances can cause tremendous problems in the case of Microservice-based approaches. The distribution into Microservices and therefore into fields of responsibility for the different teams has to be performed according to domain criteria.

Refactoring

In a single Microservice refactoring is simple since the Microservice is small. It can also be easily replaced and newly implemented.

Between Microservices the situation differs: Transferring functionalities from one Microservice to another is difficult. The functionality has to be moved into a different deployment unit. This is for sure more difficult than moving a functionality within the same unit. Between Microservices technologies are not necessarily uniform. Microservices can use different libraries or even different programming languages. In such cases the functionality has to be moved into a new Microservice. In some cases the functionality must be newly implemented in the technology of the other Microservice and subsequently transferred into this Microservice. However, this is far more complex than moving code within a Microservice.

Agile Architecture

Microservices make it easier to bring as many changes as possible into production in the shortest possible time and to reach a sustainable development speed. This is especially advantageous when there are numerous and hard to predict requirements. This is exactly the environment where Microservices are at home. Changes to a Microservice are also very simple. However, adjusting the architecture of the system as such, for instance by moving around functionalities, is not so simple.

In addition, the architecture of a system is frequently not yet optimal at the first attempt. During implementation the team learns a lot about the domain. In a second attempt, it will be much more capable of designing an appropriate architecture. Most projects suffering from bad architecture had a good architecture at the outset based on the state of knowledge at that time. However, when the project progressed, it became clear that requirements were meant differently and new requirements arose so that the initial architecture stopped fitting. Problems arise when this does not lead to consequences. If the project just continues with a more and more inappropriate architecture, the architecture will not fit at all anymore at some point. This can be avoided by adjusting the architecture step by step to the changed requirements based on the respective state of knowledge. Architecture changeability and architecture adjustment in line with new requirements are central for this. However, architecture changeability at the level of the entire system is a weakness of Microservices while changes within Microservices are very simple.

Summary

When using Microservices, architecture is even more important than in other systems as it influences also the organization and the independent work on requirements. At the same time, Microservices offer many advantages in cases where requirements are unclear and architecture therefore has to be changeable. Unfortunately, the interplay between Microservices is hard to modify since the distribution into Microservices is quite rigid due to the distributed communication between them. Besides, as Microservices can be implemented by the use of different technologies, it gets difficult to move functionalities around. On the other hand, changes to individual Microservices or their replacement are very simple.

6.3 Infrastructure and Operations

Microservices are supposed to be brought into production independently of each other and to be able to use individual technology stacks. Therefore, each Microservice usually resides on its own server. This is the only way to ensure complete technological independence. It is not possible to cope with the required multitude of systems using hardware servers. Even with virtualization the management of such an environment remains difficult. The number of required virtual machines can be higher than otherwise used by an entire business IT. When there are hundreds of Microservices, there are also hundreds of virtual machines needed and for some of them several instances e.g. for load balancing. This requires automation and appropriate infrastructures, which are able to generate a large number of virtual machines.

Continuous Delivery Pipelines

Beyond operation each Microservice requires additional infrastructure: It needs its own Continuous Delivery pipeline so that it can be brought into production independently of the other Microservices. This means that appropriate test environments and automation scripts are necessary. The large number of pipelines causes additional challenges: The pipelines have to be built up and maintained. Furthermore, to reduce expenses they need to be largely standardized.

Monitoring

Each Microservice requires in addition monitoring. This is the only way to recognize problems with the service at runtime. In case of a Deployment Monolith, it is still quite easy to monitor the system. When problems arise, the administrator can log into the system and use specific tools to analyze errors. Microservice-based systems contain so many systems that this approach is not

feasible anymore. Consequently, there has to be a monitoring, which comprises all systems. Thereby, not only the typical information from the operating system and the I/O to the hard disc and to the network should be analyzed, also a view into the application should be possible based on application metrics. This is the only way for developers to find out where the application has to be optimized and where problems exist at the moment.

Version control

Finally, every Microservice has to be stored under version control independent of the other ones. Only software, which is separately versioned, can be brought into production individually. When two software modules are versioned together, they should always be brought into production together. Otherwise a change might have influenced both modules so that in fact both services should be newly delivered. Moreover, if an old version of one of the services is in production, it is not clear whether an update is necessary or whether the new version does not contain changes – after all the new version might only have contained changes in the other Microservice.

For Deployment Monoliths a lower number of servers, environments and projects in version control would be necessary. This decreases complexity. The requirements in regards to operation and infrastructure are much higher in a Microservices environment. To deal with this complexity is the biggest challenge when introducing Microservices.

6.4 Conclusion

This chapter discussed the different challenges associated with Microservices-based approaches. At the technical level (section 6.1) the challenges are mostly a consequence of the fact that Microservices are distributed systems: Due to that, system performance and reliability are more difficult to ensure. In addition, technical complexity increases because of the greater variety of used technologies. Furthermore, code dependencies can render the independent deployment of Microservices impossible.

The architecture of a Microservice-based system (section 6.2) is extremely important due to its impact on the organization and the parallel implementation of multiple stories. At the same time, changes to the interplay of Microservices are hard. Functionalities cannot be easily transferred from one Microservice to another. Classes within a project can often even be moved automatically. Between

Microservices manual work is necessary. The interface to the code changes from local calls to communication between Microservices. This increases the necessary efforts. Finally, Microservices can be written in different programming languages – in such cases to move code entails that it has to be rewritten.

However, changes to system architecture are often necessary because of unclear requirements. Besides, the team permanently improves its knowledge about the system and its domain. Especially in circumstances where the use of Microservices is particularly advantageous because of rapid and independent deployments, architecture should be peculiarly easy to change. Within Microservices changes are indeed easy to implement, however between Microservices they are very laborious.

Finally infrastructure complexity rises due to the larger number of services (section 6.3) since more servers, more projects in version control and more Continuous Delivery pipelines are necessary. This is a central challenge encountered by Microservice-based architectures.

Part III of the book is going to show solutions for these challenges.

Essential Points

- Microservices are distributed systems. This makes them technically more complex.
- A good architecture is very important due to its impact on the organization. While the architecture is easy to modify within Microservices, the interplay between Microservices is hard to change.
- Due to the number of Microservices more infrastructure is necessary e.g. in terms of server environments, Continuous Delivery pipelines or projects in version control.

Try and Experiment

Choose one of the scenarios from chapter 3 or a project you know:

- Which are the challenges to be anticipated? Evaluate these challenges. The conclusion of this chapter highlights the different challenges once again in a compressed manner.
- Which of the challenges poses the biggest risk? Why?
- Are there possibilities to use Microservices in a way which maximizes advantages and avoids disadvantages? For example, heterogeneous technology stacks could be avoided.

7 Microservices and SOA

At first glance Microservices and SOA seem to have a lot in common: Both approaches focus on the modularization of large systems into services. Are SOA and Microservices indeed the same or are there differences? Dissecting this question contributes to an in depth understanding of Microservices. Besides, some ideas from the SOA field are interesting for Microservice-based architectures. A SOA approach can be advantageous when migrating to Microservices. It separates the functionalities of the old applications into services, which can be replaced or supplemented by Microservices.

Section 7.1 defines the term “SOA” as well as the term “service” within the SOA context. Section 7.2 extends this topic by highlighting the differences between SOA and Microservices.

7.1 What is SOA?

SOA (Service-Oriented Architecture) and Microservices share one similarity: They lack an unambiguous definition. Therefore, this section provides only one of the possible definitions. According to other definitions Microservices and SOA are indeed identical approaches. Eventually, both approaches are based on services and the distribution of applications into services.

The term “service” is central for SOA.

A SOA service should have the following characteristics:

- A service should comprise a domain functionality.
- A service can be used independently.
- It is available in the network.
- Each service has an interface. Knowledge about the interface is sufficient to use the service.
- The service can be used via different programming languages and platforms.
- To make it easy to use the service is registered in a directory. Via this directory clients search the service at run time and use it.

- The service should be coarse-grained in order to reduce dependencies. Small services can only implement sensible functionalities together with other services. Therefore, SOA focuses rather on larger services.

SOA services do not need to be newly implemented, but are already present in the company applications. Introducing SOA means to make these services available outside of those applications. Because of the distribution of applications into services their use in different contexts is facilitated. This is supposed to improve the flexibility of the overall IT – that is the goal of SOA. Due to the distribution into individual services it is possible to recycle services during the implementation of business processes. This requires only to orchestrate the individual services.

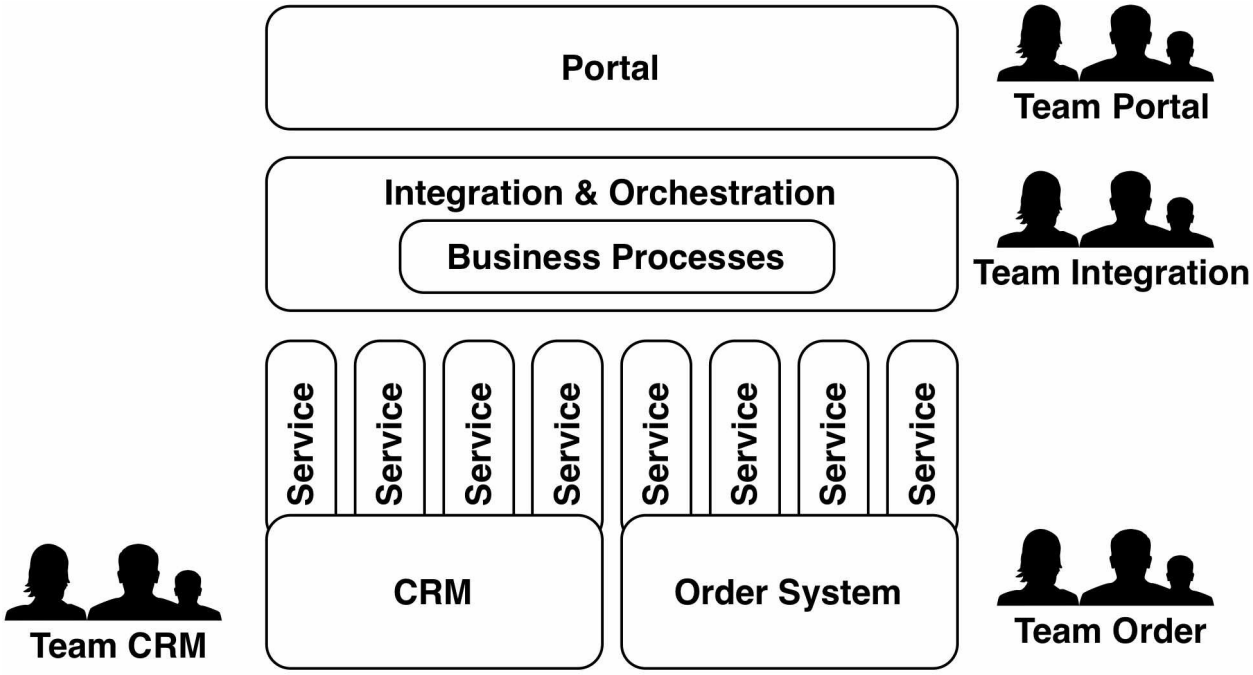


Fig. 15: Overview of a SOA landscape

Fig. 15 depicts a possible SOA landscape. Like the previous examples this example is derived from the E-commerce field. There are different systems in the SOA landscape:

- The *CRM* (Customer Relationship Management) is an application, which stores essential information about the customers. This information comprises not only contact details, but also the history of all transactions with the customer – telephone calls as well as emails or orders. The CRM exposes services, which for instance support the creation of a new customer, provide information about a customer or generate reports for all customers.
- The *Order System* is in charge of order processing. It can receive new orders, provide information about the order status or cancel an order. Also this system provides access to the different functionalities via individual services. These services might have been added as additional interface to the system after the first version was put into production.
- In the scheme CRM and the order system are the only systems. In reality there would be certainly additional systems, for instance to provide the product catalog. However, to illustrate a SOA landscape these two systems suffice.
- For the systems to be able to call each other there is an *integration platform*. This platform allows for the communication between the services. It can newly compose the services by orchestration. Orchestration can be mediated

by a technology, which models business processes and calls the individual services to execute the different processes.

- Therefore, *orchestration* is responsible for coordinating the different services. The infrastructure is intelligent and can react appropriately to the different messages. It contains the model of the business processes and thus an important part of the business logic.
- SOA can be used via a *portal*. The portal is responsible for providing the users with an interface for using the services. There can be different portals: for instance one for the customers, one for the support and one for internal employees. Likewise, the system can be called via rich client applications or mobile Apps. From an architectural perspective this does not make a difference: All such systems use the different services to make them useable for a user. Eventually, all these systems are a universal UI to be able to use all services in the SOA.

Each of these systems can be operated und further developed by an individual team. In the example there could be one team each for the CRM and the order system, and one additional team each for each portal and finally one team taking care of integration and orchestration.

Fig 16 shows how communication is structured in SOA architecture. Users typically work with SOA via the portal. Thereby business processes can be initiated, which then are implemented in the orchestration layer. These processes use the services. Especially when migrating from a Monolith to SOA users might still use a Monolith via its own user interface. However, SOA usually aims for having a portal as central user interface and an orchestration for implementing processes.

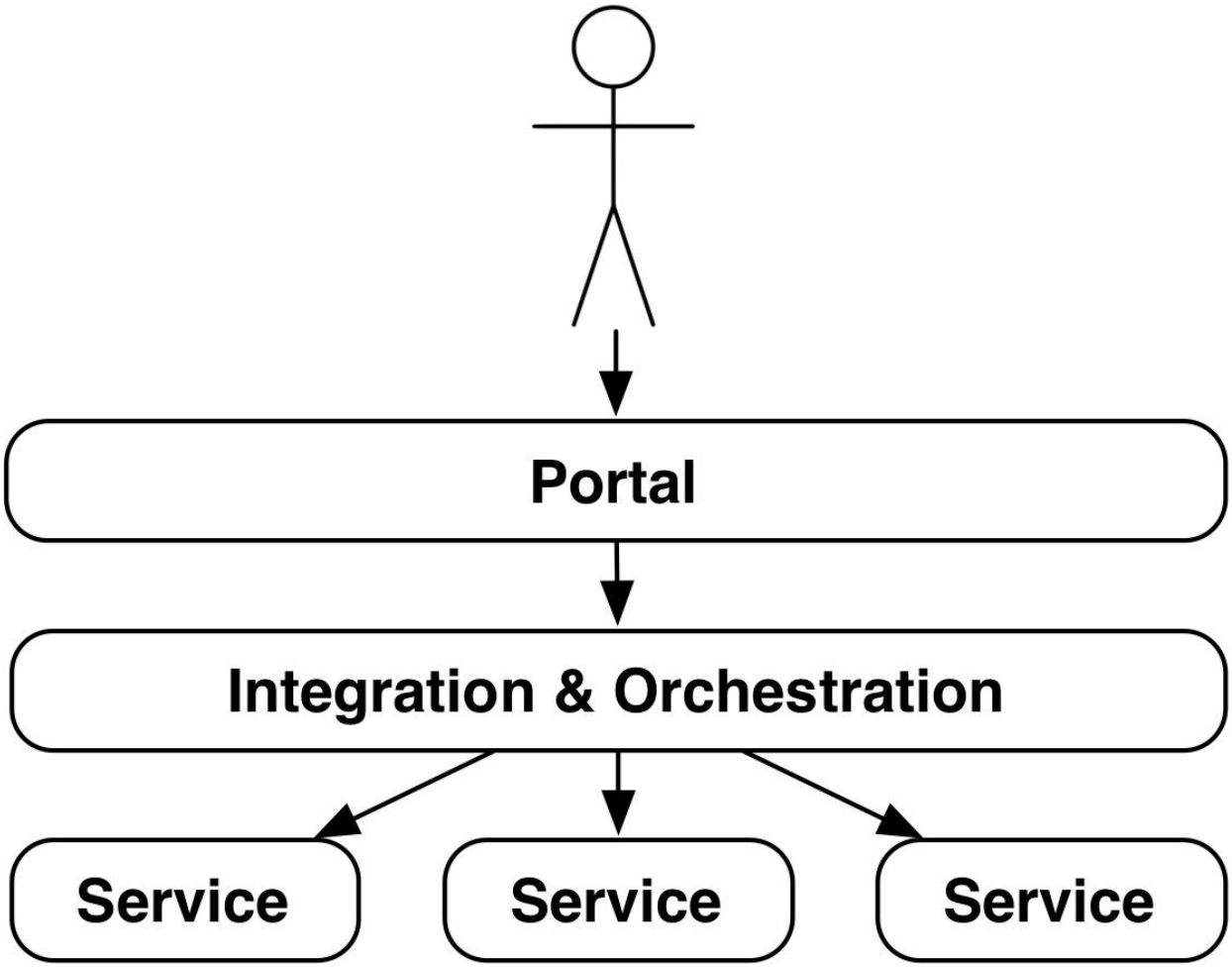


Fig. 16: Communication in a SOA architecture

Introducing SOA

Introducing SOA is a strategic initiative involving different teams. In the end the aim is to distribute the entire company IT into separate services. The distribution supports the composition of services into new functionalities in a better manner. However, this is only possible when all systems in the entire organization have been adjusted. And only when so many services are available that business processes can be implemented by simple orchestration, SOA's advantages are really evident. Therefore, the integration and orchestration technology has to be used in the entire IT to enable service communication and integration. This entails high investment costs as the entire IT landscape has to be changed. This is one of the main points of criticism in regards to SOA.

The services can also be offered to other companies and users via internet or private networks. Thus SOA is well suited to support business concepts, which are based on the outsourcing of services or the inclusion of external services. In

an E-commerce application an external provider could for instance offer simple services like address validation or complex services like a credit check.

Services in a SOA

At least when introducing SOA based on old systems the SOA services are only interfaces of large Deployment Monolith. One Monolith offers several services. The services are built upon the existing applications. Often it is not even necessary to adjust the internals of a system in order to offer the services. Such a service typically does not have a UI, instead it offers only an interface for other applications. A UI exists only for all systems. It is also not part of a service, but independent - for instance in the portal.

In addition, it is possible to implement smaller deployment units in a SOA. The definition of SOA services does not limit the size of the deployment units – quite contrary to Microservices where the size of the deployment units is a defining feature.

Interfaces and Versioning

Service versioning in SOA is a special challenge. Service changes have to be coordinated with the users of the respective service. Because of this coordination requirement, changes to the interface of the services are laborious. Service users are unlikely to adjust their own software if they do not profit from the new interface. Therefore, old interface versions frequently have to be supported as well. This means that numerous interface versions have probably to be supported if a service is used by many clients. This increases software complexity and renders changes more difficult. After all, the correct functioning of the old interfaces has to be ensured upon each new software release. If data are supplemented, challenges arise because the old interfaces do not support these data. This is no problem during reading. However, when writing it can be difficult to create new data sets without the additional data.

External Interfaces

If there are external users outside the company using the service, interface changes get even more difficult. In the worst case the service provider does not even know exactly who is using the service since it is available to anonymous users in the Internet. In that case it is nearly impossible to coordinate changes. Consequently, switching off an old service version then gets nearly unfeasible. This leads to a growing number of interface versions, and service changes get more and more difficult. This problem concerns Microservices as well (compare section 9.6).

The interface users are also facing challenges: If they need an interface modification, they have to coordinate this with the team offering the service. Then the changes have to be prioritized in relation to all other changes and wishes of other teams. As discussed already, an interface change is no easy task. Therefore, it can take quite a long time till changes are in fact implemented. This hampers the further development of the system.

Interfaces Enforce a Coordination of Deployments

After a change to the interface the deployment of the services has to be coordinated. First the service has to be deployed, which offers the new interface versions. Only then the service, which uses the new interface, can be deployed. Since applications are mostly Deployment Monoliths in the case of SOA, several services can always only be deployed together. This renders the coordination of services more difficult. In addition, the risk increases as the deployment of a Monolith takes a long time and is hard to undo – just because the changes are so extensive.

Coordination and Orchestration

Coordinating SOA via an orchestration in the integration layer poses a number of challenges. In a way a Monolith is generated: All business processes are echoed in this orchestration. This Monolith is often even worse than the usual Monoliths as it is using all systems within the enterprise IT. In extreme cases it can happen that the services only undertake the data administration while all logic is found in the orchestration. In such cases the entire SOA is in the end nothing else than a Monolith, which is having its entire logic in the orchestration.

However, even in other settings changes to SOA are not really easy: Domains are divided into services in the different systems and into business processes in orchestration. When a change to a functionality also concerns services or the user interface, things get difficult: Changing the business processes is relatively simple, but changing the service is only possible by writing code and by deploying a new version of the application providing the service. The necessary code changes and the deployment can be very laborious. Thus, the flexibility of SOA is lost, which was meant to arise from a simple orchestration of services. Modifications of the user interface cause changes to the portal or to the other user interface systems and require likewise a new deployment.

Technologies

SOA is an architecture approach and independent of a concrete technology. However, a SOA has to define a common technology for the communication between the services, like Microservices do. In addition, a concrete technology needs to be set for the orchestration of the services. Often introducing a SOA leads to the introduction of complex technologies to allow for the integration and orchestration of the services. There are special products, which support all aspects of SOA. However, they are correspondingly complex, and their features are hardly ever used to full capacity.

This technology can rapidly turn into a bottleneck. Many problems with these technologies are attributed to SOA although SOA could be implemented with other technologies as well. One of the problems is the complexity of the web services protocols. SOA on its own is still quite simple, however, in conjunction with the extensions from the WS-* environment a complex protocol stack arises. WS-* is necessary for transactions, security or other extensions. Complex protocols exacerbate the interoperability – however, interoperability is a prerequisite for a SOA.

An action on the user interface has to be processed by the orchestration and the different services. These are distributed calls within the network with associated overhead and latency. Moreover, this communication runs via the central integration and orchestration technology, which accordingly has to cope with numerous calls.

7.2 Differences Between SOA and Microservices

SOA and Microservices are related: Both aim at splitting applications into services. It is also not easy to distinguish between SOA and Microservices just because of what is happening in the network. After all, in both architecture approaches many services exchange information via the network.

Communication

Like Microservices SOA can be based on asynchronous communication or synchronous communication. SOAs can be uncoupled by sending merely events like “new order”. In such cases every SOA service can react to the event with different logic. One service can write a bill and another can initiate the delivery. The services are strongly uncoupled since they only react to events without knowing the trigger for the events. New services can easily be integrated into the system by likewise reacting to such events.

Orchestration

However, already at the level of integration differences between SOA and Microservices appear: In SOA the integration solution is also responsible for orchestrating the services. A business process is built up from services. In a Microservice-based architecture the integration solution does not possess any intelligence. The Microservices are responsible for communicating with the other services. SOA attempts to use orchestration to gain additional flexibility for the implementation of business processes. This will only work out when services and user interface are stable and do not frequently have to be modified.

Flexibility

For achieving the necessary flexibility Microservices on the other hand exploit the fact that each Microservice can be easily changed and brought into production. When the flexible business processes of SOA are not sufficient, SOA forces the change of services into Deployment Monolith or user interfaces in an additional Deployment Monolith.

Microservices place emphasis on isolation: Ideally a user interaction is completely processed within one Microservice without the need to call another Microservice. Therefore, changes required for new features are limited to individual Microservices. SOA distributes the logic to the portal, the orchestration and the individual services.

Microservices: Project Level

However, the most important difference between SOA and Microservices is the level at which the architecture aims. SOA considers the entire enterprise. It defines how a multitude of systems within the enterprise IT interacts. Microservices on the other hand represent an architecture for an individual system. They are an alternative to other modularization technologies. It would be conceivable to implement a Microservice-based system with another modularization technology and then to bring the system into production as a Deployment Monolith without distributed services. An entire SOA spans the entire enterprise IT. It has to look at different systems. An alternative to a distributed approach is not conceivable. Accordingly, the decision for a Microservice-based architecture can concern and be limited to an individual project while the introduction and implementation of SOA pertains to the entire enterprise.

The SOA scenario depicted in Fig. 15 results in a fundamentally different architecture (compare Fig. 17) if implemented using Microservices:

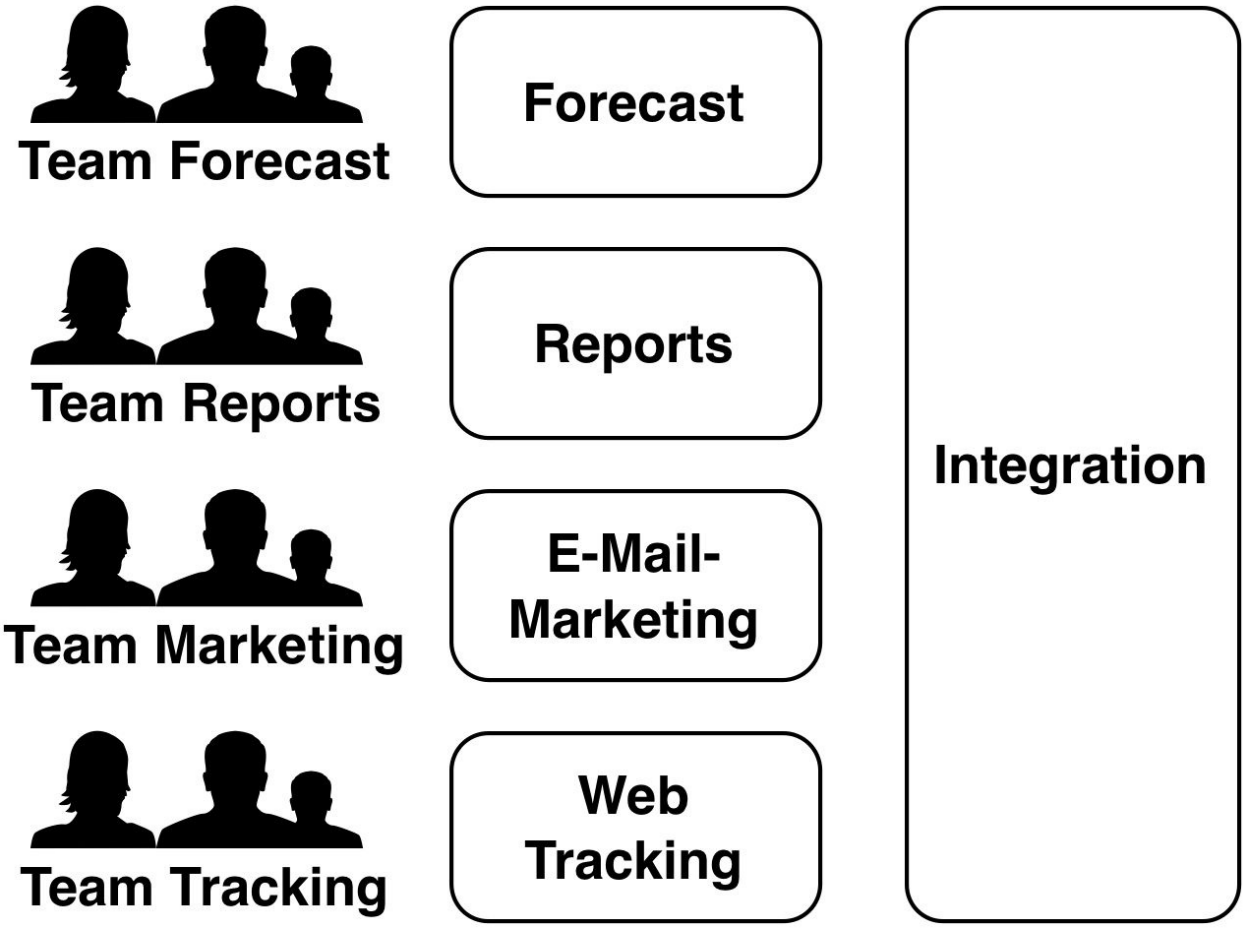


Fig. 17: CRM as Collection of Microservices

- Since Microservices refer to a single system, the architecture does not need to comprise the entire IT with its different systems, but can be limited to an individual system. In Fig. 17 this system is the CRM. Thus, implementing Microservices is relatively easy and not very costly as it is sufficient to implement one individual project rather than to change the entire IT landscape of the enterprise.
- Accordingly, a Microservice-based architecture does not require an integration technology to be introduced and used throughout the company. The use of a specific integration and communication technology is limited to the Microservice system - it is even possible to use several approaches. For instance, a high-performance access also to large data sets can be implemented by replicating the data in the database. For access to other systems again other technologies can be used. In case of SOA all services in the entire company need to be accessible via a uniform technology. This requires a uniform technology stack. Microservices focus on simpler

technologies, which do not have to fulfill as complex requirements as SOA suites.

- In addition, communication between Microservices is different: Microservices employ simple communication systems without any intelligence. Microservices call each other or send messages. The integration technology does not implement an orchestration. A Microservice can call several other Microservices and implement an orchestration on its own. In that case, the logic for the orchestration resides in the Microservice and not in an integration layer. In the case of Microservices the integration solution contains no logic, because it would originate from different domains. This conflicts with the distribution according to domains, which Microservice-based architectures aim at.
- The use of the integration is also entirely different. Microservices avoid communication with other Microservices by having the UI integrated into the Microservice and due to their domain-based distribution. SOA focuses on communication. SOA obtains its flexibility by orchestration - this is accompanied by communication between services. And in the case of Microservices the communication does not necessarily have to be implemented via messaging or REST: An integration at the UI level or via data replication is possible as well.
- CRM as complete system is not really present anymore in a Microservice-based architecture. Instead there is a collection of Microservices, which each cover specific functionalities like reports or forecasting transaction volume.
- While in SOA all functionalities of the CRM system are collected in a single deployment unit, each service is an independent deployment unit and can be brought into production independently of the other services in the case of Microservice-based approaches. Depending on the concrete technical infrastructure the services can be even smaller than the ones depicted in Fig. 17.
- Finally, the handling of UI is different: For Microservices the UI is part of the Microservice, while SOA typically offers only services, which then can be used by a portal.
- The division into UI and service in SOA has far reaching consequences: To implement a new functionality including the UI in SOA, at least the service has to be changed and the UI adjusted. This means that at least two teams have to be coordinated. When other services in other applications are used, even more teams are involved requiring consequently an even greater coordination effort. In addition there are also orchestration changes, which

are implemented likewise by a separate team. Microservices on the other hand attempt that an individual team can bring a new functionality into production with as little need for coordination with other teams as possible. Due to the Microservice-based architecture, interfaces between layers, which normally are between teams, are now within a team. This facilitates the implementation of changes. The changes can be processed in one team. If another team were involved, the changes had to be prioritized in relation to other requirements.

- Each Microservice can be developed and operated by one individual team. This team is responsible for a specific domain and can implement new requirements or changes to the domain completely independently of other teams.
- Moreover, the approach is different between SOA and Microservices: SOA introduces only one new layer above the existing services in order to combine applications in new ways. It aims at a flexible integration of the existing applications. Microservices serve to change the structure of the applications themselves – in pursuit of the goal to make changes to applications easier.

The communication relationships in case of Microservices are depicted in Fig. 18: The user interacts with the UI, which is implemented by the different Microservices. In addition, the Microservices communicate with each other. There is no central UI or orchestration.

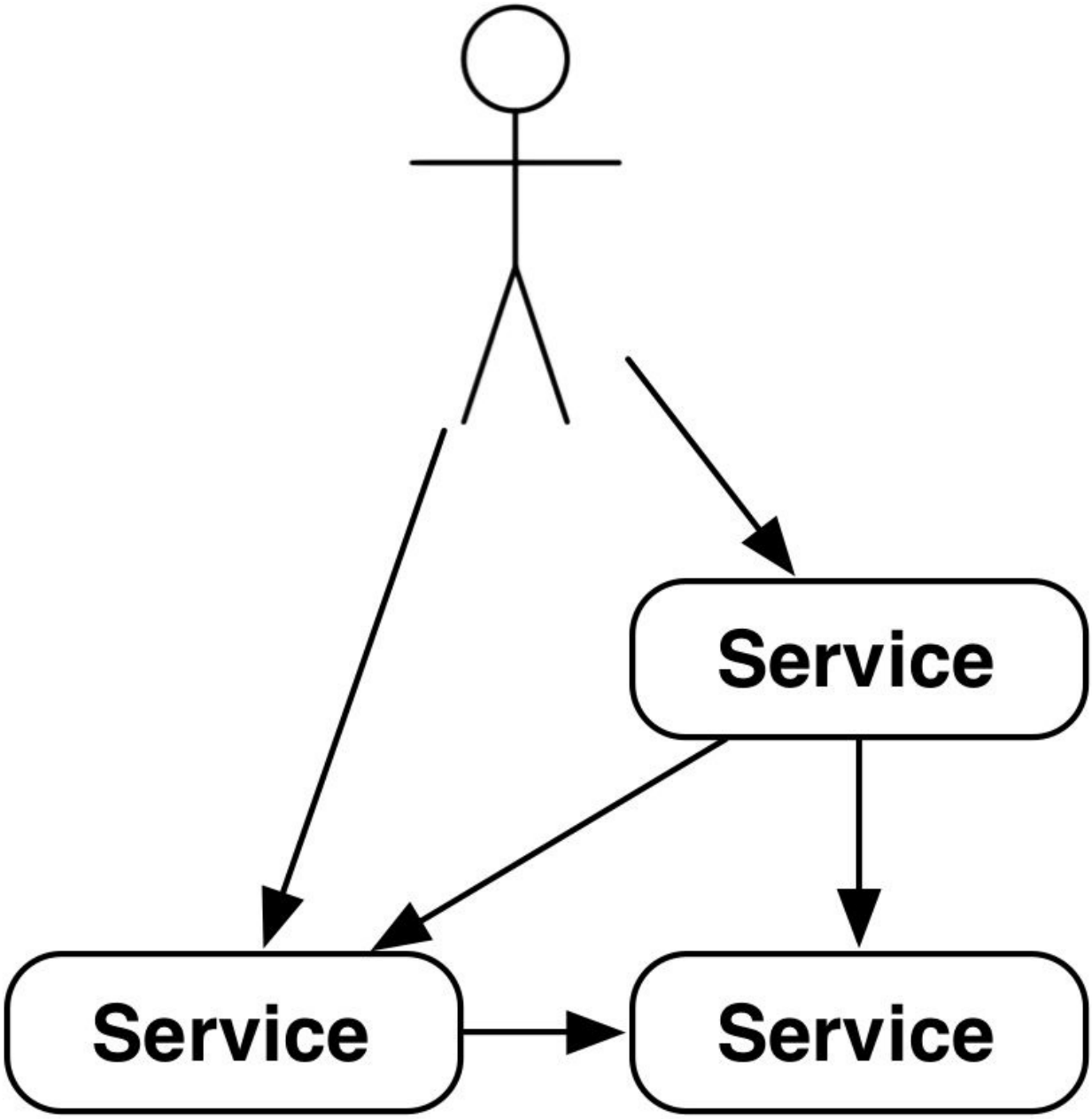


Fig. 18: Communication in the case of Microservices

Synergies

There are definitely areas where Microservices and SOA have synergies. In the end both approaches pursue the goal to resolve applications into services. Such a step can be helpful when migrating an application to Microservices: When the application is split into SOA services, individual services can be replaced or supplemented by Microservices. Certain calls can be processed by a Microservice while other calls are still processed by the application. This allows

to migrate applications in a stepwise manner and to implement the Microservices step by step.

Fig. 19 shows an example: The upper most service of CRM is supplemented by a Microservice. This Microservice now takes all calls and can, if necessary, call the CRM. The second CRM service is completely replaced by a Microservice. Thereby the CRM can be complemented by new functionalities. At the same time, it is not necessary to newly implement the entire CRM, instead Microservices can complement it at selected places. Section 8.5 presents additional approaches how legacy applications can be replaced by Microservices.

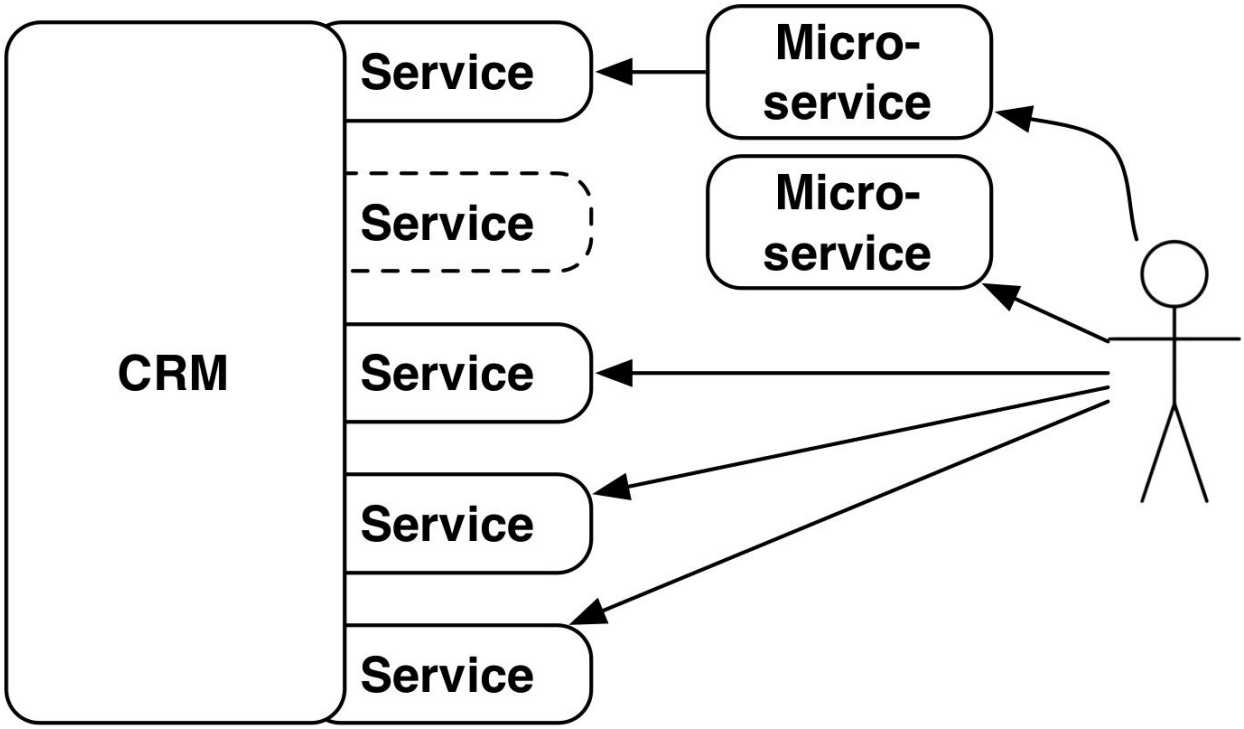


Fig. 19: SOA for migrating to Microservices

7.3 Conclusion

Tab. 2: Differences between SOA and Microservices

	SOA	Microservices
Scope	Enterprise-wide architecture	Architecture for one project
Flexibility	Flexibility by orchestration	Flexibility by fast deployment and rapid, independent development of Microservices
Organization	Services are implemented by different organizational	Services are implemented by teams in the same

	units	project
Deployment	Monolithic deployment of several services	Each Microservice can be deployed individually
UI	Portal as universal UI for all services	Service contains UI

At the organizational level the approaches are very different: SOAs place emphasis on the structure of the entire enterprise IT, Microservices can be utilized in an individual project. SOAs focus on an organization where some teams develop backend services, while a different team implements the UI. In a Microservice-based approach one team should implement everything to facilitate communication and thereby speed up the implementation of features. That is not a goal of SOA. In SOA a new feature can entail changes to numerous services and thus require communication between a large number of teams. Microservices try to avoid this.

At the technical level there are commonalities: Both concepts are based on services. The service granularity can even be similar. Because of these technical similarities it does not seem to be so easy to distinguish SOA from Microservices. However, from conceptual, architectural and organizational view points both approaches have very different effects.

Essential Points

- SOA and Microservices split applications into services, which are available in the network. Similar technologies can be employed to this end.
- SOA aims at flexibility at the level of the enterprise IT by orchestrating the services. This is a complex undertaking and only works when the services do not need to be modified.
- Microservices focus on individual projects and aim at facilitating deployment and enabling parallel work on different services.

Try and Experiment

A new functionality is supposed to be incorporated into the SOA landscape depicted in Fig. 15. The CRM does not have support for email campaigns. Therefore, a system for email campaigns has to be implemented. It is supposed to contain a service for the creation and execution of campaigns and a service for evaluating the results of a campaign.

An architect has to answer the following questions:

- Is the SOA infrastructure needed for integrating the two services? The service for campaign evaluation needs a large amount of data.
  - Would it be better to use data replication, UI-level integration or service calls for accessing the large amount of data?
  - Which of these integration options is typically offered by SOA?
- Should the service integrate into the existing portal or rather have its own user interface? Which arguments favor the one or the other option?
- Should the new functionality be implemented by the CRM team?

Part III: Implementing Microservices

This part of the book demonstrates how Microservices can be implemented. After studying this part the reader cannot only design Microservice-based architectures, but also implement them and evaluate the organizational effects.

Chapter 8: Architecture of Microservice-based Systems

Chapter 8 describes the architecture of Microservice-based systems. It focuses on the interplay between individual Microservices.

The domain architecture deals with Domain-Driven Design as basis of Microservice-based architectures and shows metrics which allow to measure the quality of the architecture. Architecture management is a challenge: It can be difficult to keep the overview of the numerous Microservices. However, often it is sufficient to understand how a certain use case is implemented and which Microservices interact in a specific scenario.

Practically all IT systems are subject to more or less profound change. Therefore the architecture of a Microservice system has to evolve and the system has to undergo continued development. To achieve this several challenges have to be solved, which do not arise in this form in the case of Deployment Monoliths – for instance the overall distribution into Microservices is difficult to change. However, changes to individual Microservices are simple.

In addition, Microservice systems need to integrate legacy systems. This is quite simple as Microservices can treat legacy systems as blackbox. A replacement of a Deployment Monolith by Microservices can progressively transfer more functionalities into Microservices without having to adjust the inner structure of the legacy system or having to understand the code in detail.

The technical architecture comprises typical challenges for the implementation of Microservices. In most cases there is a central configuration and coordination for all Microservices. Furthermore, a load balancer distributes the load between the individual instances of the Microservices. The security architecture has to leave each Microservice the freedom to implement its own authorizations in the system, but also ensure that a user needs to log in only once. Finally, Microservices

should return information concerning themselves as documentation and as metadata.

Chapter 9: Integration and Communication

Chapter 9 shows the different possibilities for the integration and communication between Microservices. There are three possible levels for integration:

- Microservices can integrate at the web level. In that case each Microservice delivers a part of the web UI.
- At the logic level Microservices can communicate via REST or messaging.
- Data replication is also possible.

Via these technologies the Microservices have internal interfaces for other Microservices. The complete system can have one interface to the outside. Changes to the different interfaces create different challenges. Accordingly, this chapter also deals with versioning of interfaces and the effects thereof.

Chapter 10: Architecture of Individual Microservices

Chapter 10 describes possibilities for the architecture of an individual Microservice. There are different approaches for an individual Microservice:

- CQRS divides read and write access into two separate services. This allows for smaller services and an independent scaling of both parts.
- Event Sourcing administrates the state of a Microservice via a stream of events from which the current state can be deduced.
- In a hexagonal architecture the Microservice possesses a core, which can be accessed via different adaptors and which communicates also via such adaptors with other Microservices or the infrastructure.

Each Microservice can follow an independent architecture.

In the end all Microservices have to handle technical challenges like resilience and stability – these issues have to be solved by their technical architecture.

Chapter 11: Testing Microservices and Microservice-based Systems

Testing is the focus of chapter 11. Also tests have to take the special challenges associated with Microservices into consideration.

The chapter starts off with explaining why tests are necessary at all and how a system can be tested in principle.

Microservices are small deployment units. This decreases the risk associated with deployments. Accordingly, besides tests also optimization of deployment can help to decrease the risk.

Testing the entire system represents a special problem in case of Microservices since only one Microservice at a time can pass through this phase. If the tests last one hour, only eight deployments will be feasible per working day. In the case of 50 Microservices that is by far too few. Therefore, it is necessary to limit these tests as much as possible.

Often Microservices replace legacy systems. The Microservices and the legacy system both have to be tested – and also their interplay. Tests for the individual Microservices differ in some respects greatly from tests for other software systems.

Consumer-driven contract tests are an essential component of Microservice tests: They test the expectations of a Microservice in regards to an interface. Thereby the correct interplay of Microservices can be ensured without having to test the Microservices together in an integration test. Instead a Microservice defines its requirements for the interface in a test, which the used Microservice can execute.

Microservices have to adhere to certain standards in regards to monitoring or logging. The adherence to these standards can also be checked by tests.

Chapter 12: Operation and Continuous Delivery of Microservices

Operation and Continuous Delivery are the focus of chapter 12. Especially the infrastructure is an essential challenge when introducing Microservices. Logging and monitoring have to be uniformly implemented across all Microservices, otherwise the associated expenditure gets too large. In addition, there should be a uniform deployment. Finally, starting and stopping of Microservices should be possible in a uniform manner – i.e. via a simple control. For these areas the chapter introduces concrete technologies and approaches. Additionally, the chapter presents infrastructures which especially facilitate the operation of a Microservices environment.

Chapter 13: Organizational Effects of a Microservice-based Architecture

Finally chapter 13 discusses how Microservices influence the organization. Microservices allow for a simpler distribution of tasks to independent teams and thus for parallel work on different features. To that end the tasks have to be distributed to the teams, which subsequently introduce the appropriate changes into their Microservices. However, new features can also comprise several Microservices. In that case one team has to put requirements to another team – this requires a lot of coordination and delays the implementation of new features. Therefore, it can be better that teams also change Microservices of other teams.

Microservices divide the architecture into micro and macro architecture: In regards to micro architecture the teams can make their own decisions while the macro architecture has to be defined for and coordinated across all Microservices. In areas like operation, architecture and testing individual aspects can be assigned to micro or macro architecture.

DevOps as organizational form fits well to Microservices since close cooperation between operation and development is very useful, especially for the infrastructure intensive Microservices.

The independent teams each need their own independent requirements, which in the end have to be derived from the domain. Consequently, Microservices have also effects in these areas.

Code recycling is likewise an organizational problem: How do the teams coordinate the different requirements for shared components? A model which is inspired by open source projects can help.

However, there is of course the question whether Microservices are possible at all without organizational changes – after all, the independent teams constitute one of the essential reasons for introducing Microservices.

8 Architecture of Microservice-based Systems

This chapter discusses how Microservices should behave from the outside and how the entire Microservice system can be developed. Chapter 9 will show possible communication technologies, which are another important technology component. Chapter 10 focuses on the architecture of individual Microservices.

Section 8.1 describes what the domain architecture of a Microservice system should look like. Section 8.2 presents appropriate tools to visualize and manage the architecture. Section 8.3 shows how the architecture can be adapted in a stepwise manner. Only the constant adaptation of the software architecture ensures that the system remains maintainable in the long run and can be developed further. Section 8.4 depicts which goals and which approaches are important to enable further development.

Subsequently, a number of approaches for the architecture of a Microservice-based system are explained. Section 8.6 introduces Event-driven Architecture. This approach allows for architectures that are very loosely coupled. Section 8.5 discusses the special challenges which arise when a legacy application is supposed to be supplemented or replaced by Microservices.

Finally 8.7 deals with the topic which technical aspects are relevant for the architecture of a Microservice-based system. Some of these aspects are presented in depth in the following sections: mechanisms for coordination and configuration (section 8.8), for Service Discovery (section 8.9), Load Balancing (section 8.10), scalability (section 8.11), security (section 8.12) and finally documentation and metadata (section 8.13).

8.1 Domain Architecture

The domain architecture of a Microservice-based system determines which Microservices within the system should implement which domain. It defines how the entire domain is split into different areas, which are each implemented by one Microservice and thus one team. To devise such an architecture presents a central challenge when introducing Microservices. After all, it is an important motivation for the use of Microservices that changes to the domain can ideally be

implemented by just one team by changing only one Microservice – so that little coordination and communication across teams is required. In this way, Microservices support the scaling of the software development since even large teams need little communication and thus can still work productively.

To really achieve this, a central point is the design of a domain architecture for the Microservices, in which changes are really limited to single Microservices and thus individual teams. When the distribution into Microservices does not support this, changes will require additional coordination and communication. In such a case the Microservice-based approach cannot bring its advantages fully to bear.

Strategic Design and Domain-Driven Design

Section 4.3 discussed already the distribution of Microservices based on Strategic Designs, which are taken from Domain-driven Design. A central element is here that the Microservices are distributed into contexts – i.e. areas which present each a separate functionality.

Often architects develop a Microservice architecture based on entities from a domain model. A certain Microservice implements the logic for a certain type of entity. In such a case there is for instance one Microservice for customers, one for items and one for deliveries. This approach conflicts with the idea of *Bounded Context*, according to which a uniform modeling of data is impossible. Moreover, this approach isolates changes very badly. When a process is supposed to be modified and for this reason entities have likewise to be adapted, the change is distributed across different Microservices. Thus, changing the order process will concern also the entity modeling for customers, items and deliveries. When that is the case, the three Microservices for the different entities have to be changed in addition to the Microservice for the order process. To avoid this, it can be sensible to keep certain parts of the data for customers, items and deliveries in the Microservice for the order process. This limits changes to the order process even in that case to only one Microservice when the data modeling has to be modified.

However, there can still be services dedicated to the administration of certain entities. For instance, it can be necessary to administrate at least the most fundamental data of a certain business entity in a service. Thus, a service can definitely administrate the client data, but leave specific client data such as a bonus program number to other Microservices – for example to the Microservice for the order process, which likely has to know this number.

Example Otto Shop

An example – i.e. the architecture of the Otto shop – illustrates this concept. There are on the one hand services like user, order or product, which are rather oriented towards data, and on the other hand areas like tracking, search & navigation and personalization, which are not geared to data, but to functionalities. Exactly such a domain design should be aimed at in a Microservice-based system.

A domain architecture requires a precise understanding of the domain. The domain architecture comprises not only the division of the system into Microservices, but also the dependencies. A dependency arises when a dependent Microservice uses another one – for instance by calling the Microservice, by using elements from the UI of the Microservice or by replicating its data. Such a dependency means that changes to a Microservice can influence also the Microservice that is dependent on it. If the Microservice modifies for instance its interface, the dependent Microservice has to be adapted to these changes. Also new requirements concerning the dependent Microservice might necessitate that the other Microservice modifies its interface. If the dependent Microservice needs more data to implement the requirements, the other Microservice has to offer these data and to adjust its interface accordingly.

For Microservices such dependencies cause disadvantages beyond software architecture: After all, Microservices can be implemented by different teams. In that case a change to an interface necessitates also collaboration between teams – this, however, is laborious and time-consuming.

Managing Dependencies

Managing dependencies between Microservices is central for the architecture of the system. Having too many dependencies will preclude that Microservices can be changed in isolation – which disagrees with the aim to develop Microservices independently of each other. Here, the two fundamental rules for good architecture apply:

- There should be a **loose coupling** between components such as Microservices. This means that they should have only few dependencies on other Microservices. This makes it easier to modify them since changes will only affect an individual Microservice.
- Within a component such as a Microservice the constituent parts should work closely together. This is referred to as having **high cohesion**. This ensures that all constituent parts within a Microservice really belong together.

When these two prerequisites are not given, it will be hardly possible to change an individual Microservice in an isolated manner, and changes will have to be coordinated across multiple teams and Microservices – this is just what Microservice-based architectures are supposed to avoid. However, this is actually rather a symptom: The fundamental problem is how the domain-based split of the functionalities between the Microservices was done – obviously functionalities, which would have belonged together in one Microservice, have been distributed across different Microservices. An order process, for instance, has also to generate a bill. These two functionalities are so different that they have to be distributed into at least two Microservices. However, when each modification of the order process affects also the Microservice, which creates the bills, the domain-based modeling is not optimal and should be adjusted. The functionalities have to be distributed differently to the Microservices, as we will see.

Unintended Domain-Based Dependencies

Not only a high number of dependencies poses a problem. Certain domain-based dependencies can simply be nonsensical. It is for instance surprising when in an E-commerce system the team responsible for product search suddenly has an interface with the Microservice for billing - because that should not be the case from a domain-based point of view. However, especially concerning the domains there are continuously surprises for laypersons. When a dependency is not meaningful from a domain-based point of view, something regarding the functionality of the Microservices has to be wrong. Maybe the Microservice implements features which belong into other Microservices from a domain-based perspective. Perhaps in the context of product search a scoring of the customer is required, which is implemented as part of billing. In that case one should consider whether this functionality is really implemented in the right Microservice. To keep the system maintainable in the long run, such dependencies have to be questioned and, if necessary, removed from the system. For instance, the scoring can be moved into an new independent Microservice or transferred into another existing Microservice.

Cyclic Dependencies

Cyclic dependencies can present additional problems for a comprehensive architecture. Let us assume that the Microservice for the order process calls the Microservice for billing (see Fig. 20). The Microservice for billing fetches data from the order process Microservice. When the Microservice for the order process is changed, modifications to the Microservice for billing might be

necessary since this Microservice fetches data from the Microservice for the order process. Conversely, changes to the billing Microservice entail changes to the order Microservice as this Microservice calls the billing Microservice. For this reason, cyclic dependencies are problematic: The components cannot be changed anymore in isolation, contrary to the underlying aim for a split into separate components. In case of Microservices especially much emphasis is laid on the independence, which is violated in this case. In addition to the necessary coordination of changes it can happen that the deployment has to be coordinated. When a new version of the one Microservice is rolled out, a new version of the other Microservice might have to be rolled out as well, if they have a cyclic dependency.

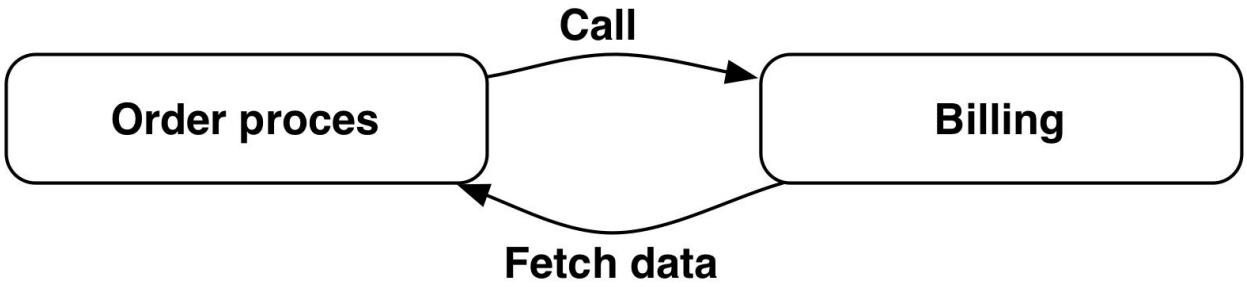


Fig. 20: Cyclic dependency

The remainder of the chapter shows approaches which allow to build Microservice-based architectures in such a way that they have a sound structure from a domain-based perspective. Metrics like cohesion and loose coupling can confirm that the architecture is really fitting. In the context of approaches like Event-driven Architecture (section 8.6) Microservices have hardly any direct technical dependencies since they send only messages. Who is sending the messages and who is processing them, can hardly be determined from the code so that the metrics look very good. However, from a domain-based perspective the system can still be much too complicated, since the domain-based dependencies are not examined by the metrics. Domain-based dependencies arise when two Microservices exchange messages. However, this is hardly ascertainable by code analysis so that the metrics will always look quite good. Thus metrics can only hint at problems. By just optimizing the metrics, the symptoms are optimized, but the underlying problems remain unsolved. Even worse: Even systems with good metrics can have architectural weaknesses. Therefore the metric looses it value to determine the quality of a software system.

A special problem in the case of Microservices is that dependencies between Microservices can also influence the independent deployment. If a Microservice requires a new version of another Microservice because it uses, for instance, a new version of an interface, the deployment will also be dependent: The Microservice has to be deployed before the dependent Microservice can be deployed. In extreme cases this can result in a large number of Microservices that have to be coordinately deployed – this is just what is supposed to be avoided. Microservices should be deployed independently of each other. Therefore, dependencies between Microservices can present an even greater problem than would be the case for modules within a Deployment Monolith.

8.2 Architecture Management

For a domain architecture it is critical which Microservices there are and what the communication relationships between the Microservices look like. Also in other systems the relationships between the components are very important. When domain-based components are mapped on modules, classes, Java packages, JAR files or DLLs, specific tools can determine the relationships between the components and control the adherence to certain rules. This is achieved by a static code analysis.

Tools for Architecture Management

If no such architecture management happens, unintended dependencies will rapidly creep in. The architecture will get more and more complex and hard to understand. Only with the help of architecture management tools developers and architects will be able to keep track of the system. Within a development environment developers view only individual classes. The dependencies between classes can only be found in the source code and are not readily discernible.

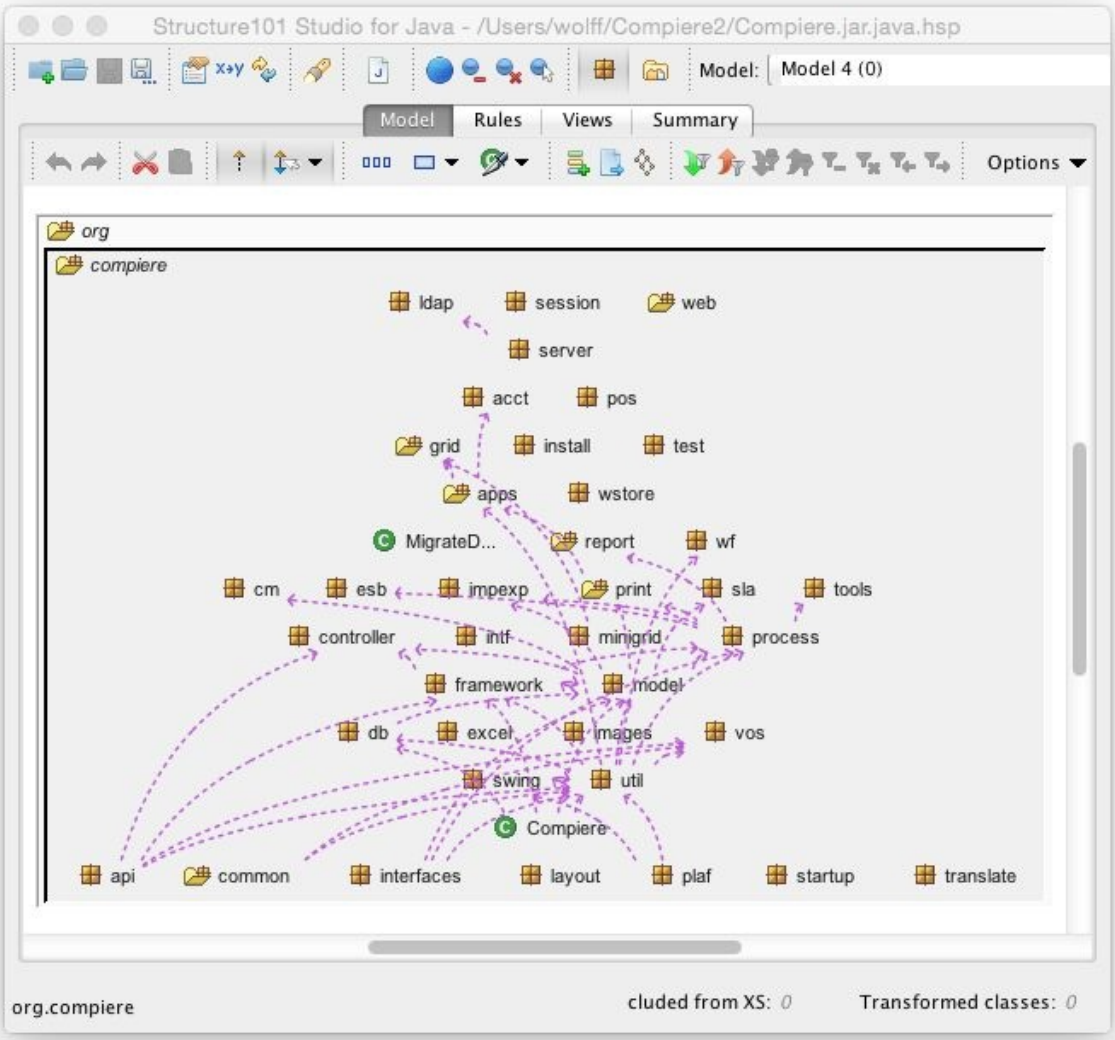


Fig. 21: Screenshot of the Architecture Management Tool Structure 101

Fig. 21 depicts the analysis of a Java project by the architecture management tool Structure 101. The image shows classes and Java packages, which contain classes. A Levelized Structure Map (LSM) presents an overview of them. Classes and packages which are further at the top of the LSM use classes and packages which are depicted further to the bottom of the LSM. To simplify the diagram, these relationships are not indicated there.

Cycle-Free Software

Architectures should be free of cycles. Cyclic dependencies mean that two artifacts are using each other reciprocally. In the screenshot such cycles are presented by dashed lines. They always run from bottom to top. The reciprocal

relationship in the cycle would be running from top to bottom and is thus not depicted.

Apart from cycles also packages which are located at a wrong position are relevant. There is, for instance, a package *util* that according to its name is supposed to contain helper classes. However, it is not located at the very bottom of the diagram. Thus, it has to have dependencies to packages or classes which are further down – which should not be the case. Helper classes should be independent from other system components and thus be depicted at the very bottom of an LSM.

Architecture management tools like Structure 101 cannot only analyze architectures, but with this tool architects can also define prohibited relationships between packages and classes. If a developer violates these rules, he/she will obtain an error message and can modify the code.

With the help of tools like Structure 101 the architecture of a system can easily be visualized. The compiled code has only to be loaded into the tool for analysis. In that way the visualization of the architecture is easily ensured.

Microservices and Architecture Management

For Microservices the problem is much larger: Relationships between Microservices are not as easy to determine as the relationships between code components. After all, the Microservices can even be implemented in different technologies. They communicate only via the network. Their relationships elude any management at code level since they appear only indirectly in the code. However, if the relationships between Microservices are unknown, architecture management becomes impossible.

There are different possibilities to visualize and manage the architecture:

- Each Microservice can bring a documentation along (compare section 8.13), which lists all used Microservices. This documentation has to adhere to a predetermined format, which enables visualization.
- The communication infrastructure can deliver the necessary data. If a Service Discovery (section 8.9) is used, it will be aware of all Microservices and will know which Microservices have access to which other Microservices. These data can then be used for the visualization of the relationships between the Microservices.

- If the access between Microservices is safeguarded by a firewall, the rules of the firewall will at least tell which Microservice can communicate with which other Microservice. This can also be used as basis for a visualization of relationships.
- Traffic within the network also reveals which Microservices communicate with which other Microservices. Tools like Packetbeat (compare section 12.3) can be very helpful here. They visualize the relationships between Microservices based on the recorded network traffic.
- The distribution into Microservices should correspond to the distribution into teams. If two teams can hardly work independently of each other any more, this is likely due to a problem in the architecture: The Microservices of the two teams depend so strongly on each other that they can now only be modified together. The involved teams probably know already due to the increased communication requirement which Microservices are problematic. To verify the problem, an architecture management tool or a visualization can be used. However, manually collected information might even be sufficient.

Tools

Different tools are useful to evaluate data about dependencies:

- There are versions of Structure 101 which can use custom data structures as input. One still has to write an appropriate importer. Structure 101 will then recognize cyclic dependencies and can depict the dependencies graphically.
- Gephi can generate complex graphs, which are helpful for visualizing the dependencies between Microservices. Also here a custom importer has to be written for importing the dependencies between the Microservices from an appropriate source into Gephi.
- jQAssistant is based on the graph database neo4j. It can be extended by a custom importer. Then the data model can be checked according to rules.

For all these tools custom development is necessary. It is not possible to analyze a Microservice-based architecture just like that, there is always some extra effort required. Since communication between Microservices cannot be standardized, it will likely also in the future not be possible to avoid custom developments.

Is Architecture Management Important?

The architecture management of Microservices is important as it is the only way to prevent chaos in the relationships between the Microservices. Microservices are a special challenge in this respect: With modern tools a Deployment Monolith

can be quite easily and rapidly analyzed. For Microservice-based architectures there are not even tools which can analyze the entire structure in a simple manner. The teams first have to create the necessary prerequisites for an analysis. Changing the relationships between Microservices is difficult – as the next section will show. Therefore, it is even more important to constantly review the architecture of the Microservices in order to be able to correct arising problems as early as possible. It is in favor of Microservice-based architectures that the architecture is also reflected in the organization. Problems with communication thus will point out architectural problems. Even without a formal architecture management architectural problems often become obvious.

On the other hand, experiences with complex Microservice-based systems teach that in such systems nobody understands the entire architecture. However, this is also not necessary since most changes are limited to individual Microservices. If a certain use case is supposed to be changed, which involves multiple Microservices, it is sufficient to understand this interaction and the involved Microservices. A global understanding is not absolutely necessary. This is a consequence of the independence of the individual Microservices.

Context Map

Context Maps present a possibility to get an overview of the architecture of a Microservice-based system[1]. They illustrate which domain models are used by which Microservices and visualize thus the different Bounded Contexts (compare section 4.3). The *Bounded Contexts* do not only influence the internal data presentation in the Microservices. Also in the case of calls between Microservices data are exchanged. They have to be in line with some type of model. However, the data models underlying communication can be distinct from the internal representations. If a Microservice for instance is supposed to identify recommendations for customers of an E-commerce shop, complex models can be employed internally for this, which contain a lot of information about customers, products and orders and correlate them in complex ways. To the outside these models are presumably much simpler.

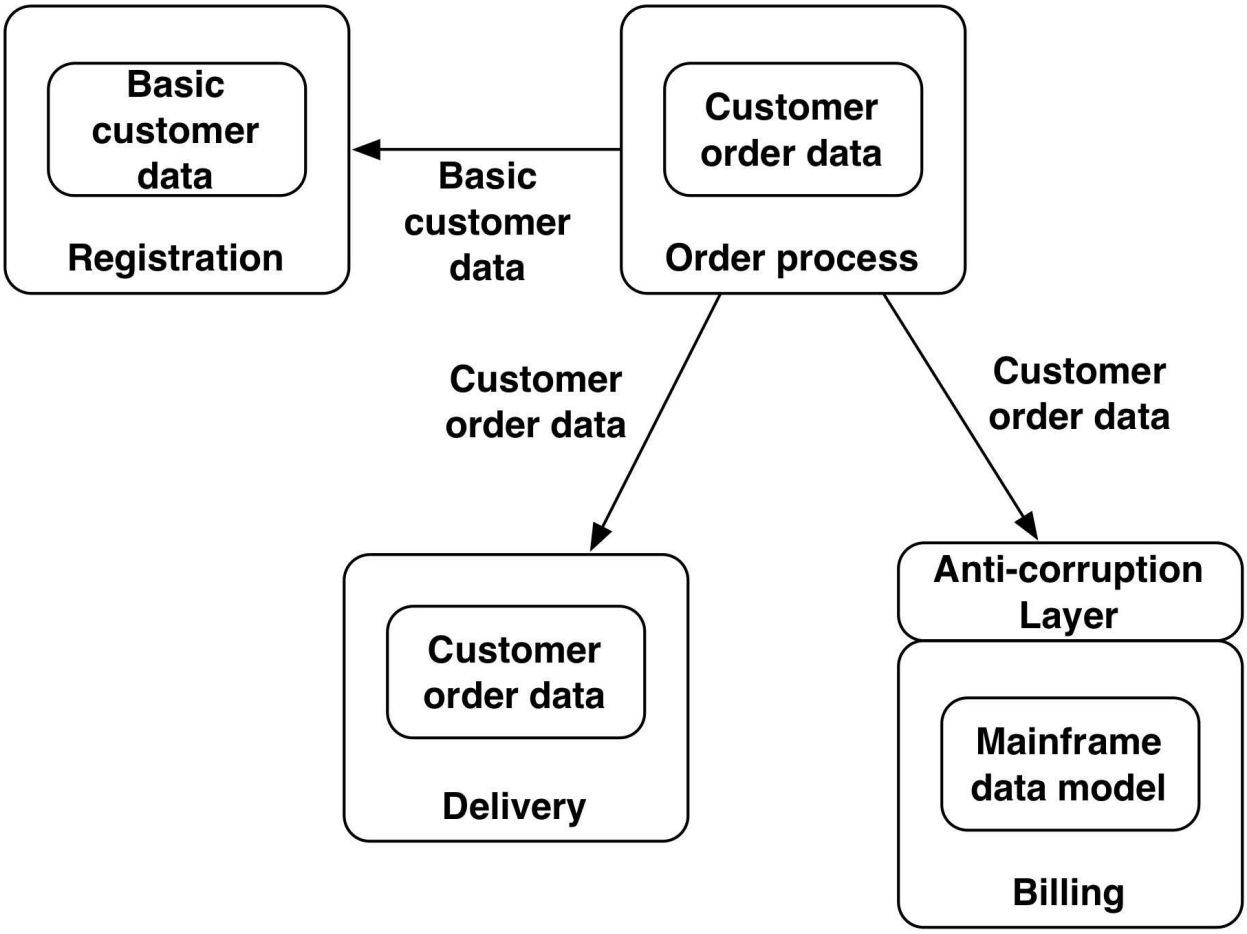


Fig. 22: An example for a Context Map

Fig. 22 shows an example for a *Context Map*:

- The registration registers the basic data of each customer. The order process also uses this data format to communicate with registration.
- In the order process the customer's basic data is supplemented by data such as billing and delivery address to obtain the customer order data. This corresponds to a *Shared Kernel* (compare section 4.3). The order process shares the kernel of the customer data with the registration process.
- The delivery and the billing Microservice use customer order data for communication, the delivery Microservice uses it even for the internal representation of the customer. Thus this model is a kind of standard model for the communication of customer data.
- Billing uses an old mainframe data model. Therefore, customer order data for the outside communication are decoupled from internal representation by an *Anti-corruption Layer*. The data model represents namely a very bad abstraction, which should by no means affect other Microservices.

In this model it stands out that the internal data representation in registration propagates to the order process. There it serves as basis for the customer order data. This model is used in delivery as internal data model as well as in the communication with billing and delivery. Accordingly, the model is hard to change since it is used by so many services. If this model was to be changed, all these services would have to be modified.

However, there are also advantages associated with this. If all these services had to implement the same change to the data model, only a single change would be necessary to update all Microservices at once. Nevertheless, this disagrees with the idea that changes should always concern only a single Microservice. If the change remains limited to the model, the shared model is advantageous since all automatically use the current modeling. However, when the change entails changes in the Microservices, now multiple Microservices have to be modified – and coordinately brought into production. This conflicts with an independent deployment of Microservices.

Try and Experiment

Download a tool for the analysis of architectures. Candidates are Structure 101, Gephi or jQAssistant. Use this tool to get an overview of an existing code basis. Which possibilities are there to insert your own dependency graphs into the tool? This would allow to also analyze the dependencies within a Microservice-based architecture with this tool.

spigo is a simulation for the communication between Microservices. It can be used to get an impression of more complex Microservice-based architectures.

8.3 Techniques to Adjust the Architecture

Microservices are first of all interesting for software which is subject to many changes. Due to the distribution into Microservices the system disaggregates into deployment units, which can be further developed independently of each other. This way each Microservice can implement its own stream of stories or requirements. Consequently, multiple changes can be worked on in parallel without much need for coordination.

Experience teaches that the architecture of a system is subject to changes. A certain distribution into domain-based components might seem sensible at first. However, once the architect gets to know the domain better, he/she might come to the conclusion that another distribution would be better. New requirements are hard to implement with the old architecture since it was devised based on different premises. This is especially frequent for agile processes, which entail less planning and more flexibility.

Where Does Bad Architecture Come from?

A system with a bad architecture does normally not come into being because the wrong architecture has been chosen at the outset. Based on the information available at the start of the project the architecture is often good and consistent. The problem is frequently that the architecture is not modified when there are new insights, which suggest changes to the architecture. The symptom was already mentioned in the last section: New requirements cannot be rapidly and easily implemented anymore. To that end the architecture would have to be changed. When this pressure to introduce changes is ignored for too long, the architecture will not fit at all anymore at some point. The permanent adjustment and modification of the architecture are essential prerequisites for keeping the architecture in a really sustainable state.

This section shows which techniques allow to change the interplay between Microservices in order to adapt the architecture to the entire system.

Changes in Microservices

Within a Microservice adjustments are easy. The Microservices are small and manageable. It is no big deal to adjust structures. And if the architecture of an individual Microservice is completely insufficient, it can be rewritten since it is not very large. Within a Microservice it is also easy to move components or to restructure the code in another manner. The term Refactoring [2] denotes techniques which serve to improve the code structure. Many of them even automate development tools. This allows for an easy adjustment of the code of an individual Microservice.

Changes to the Overall Architecture

However, when the split of the functionalities between the Microservices is not in line any more with the requirements, changing just one Microservice will not be sufficient. To achieve the necessary adjustment of the complete architecture,

functionalities have to be moved between Microservices. There can be different reasons for this:

- The Microservice is too large and has to be divided. Indications for this can be that the Microservice is hardly intelligible anymore or even that large that a single team is not sufficient to develop it further. Another indication can be that the Microservice comprises more than one *Bounded Context*.
- A functionality belongs really into another Microservice. An indication for that can be that certain parts of a Microservice communicate a lot with another Microservice. In that case the Microservices do not have a loose coupling anymore. Such intense communication can imply that the component belongs into another Microservice. Likewise, a low cohesion in a Microservice can suggest that the Microservice should be divided. In that case there are areas in a Microservice which depend little on each other. Consequently, they do not really have to be in one Microservice.
- Functionalities should be used by multiple Microservices. This can for instance become necessary when a Microservice has to use logic from another Microservice due to a new functionality.

There are three main challenges: Microservices have to be split, code has to be moved from one Microservice into another, and multiple Microservices are supposed to use the same code.

Shared Libraries

If two Microservices are supposed to use code together, the code can be transferred into a shared library (compare Fig. 23). The code is removed from the Microservice and packaged in a way that allows it to be used by the other Microservices. A prerequisite for this is that the Microservices are written in technologies that enable the use of a shared library. This is the case when they are written in the same language or at least use the same platform – e.g. JVM (Java Virtual Machine) or .NET Common Language Runtime (CLR).

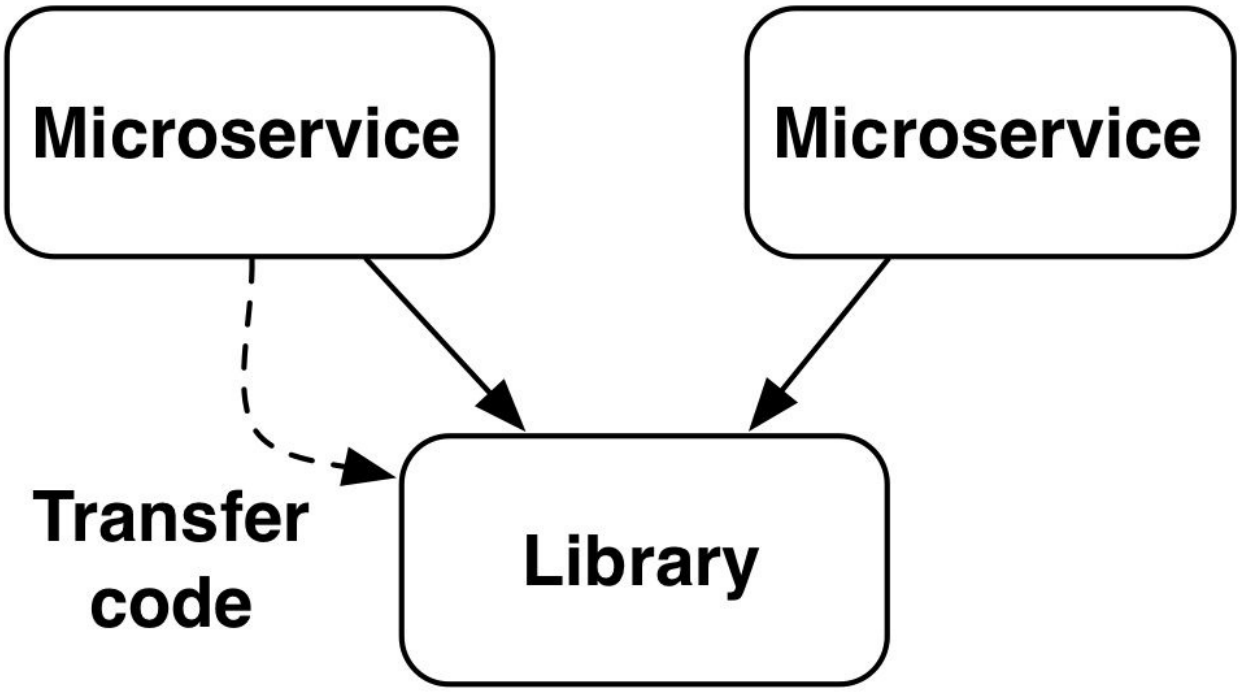


Fig. 23: Shared library

A shared library means that the Microservices become dependent on each other. Work on the library has to be coordinated. Features for both Microservices have to be implemented in the library. Via the backdoor each Microservice notices changes which are really meant for the other Microservice. This can result in errors. Therefore, the teams have to coordinate the development of the library and the changes to the library. Under certain conditions changes to a library can necessitate that a Microservice has to be newly deployed – for instance because a security gap has been closed in the library.

Moreover, via the library the Microservices might obtain additional code dependencies to 3rd party libraries. In a Java JVM, 3rd party libraries can only be present in one version. When the shared library requires a certain version of a 3rd party library, also the Microservice has to use this specific version and cannot use a different one. Besides, libraries often have a certain programming model. In that way libraries can provide code, which can be called, or a framework, in which custom code can be integrated, which is then called by the framework. The library might pursue an asynchronous model or a synchronous model. Such approaches can fit more or less well to a respective Microservice.

Microservices do not focus on the reuse of code since this leads to new dependencies between the Microservices. An important aim of Microservices is independence so that code reuse often causes more disadvantages than advantages.

This is a renunciation of the ideal of code recycling. Developers in the nineties still pinned their hopes on code reuse in order to increase productivity. Moving code into a library also has advantages. Errors and security gaps have to be corrected only once. The Microservices use always the current library version and thus automatically get the solutions for the errors.

Another problem associated with code reuse is that it requires a detailed understanding of the code – especially in the case of frameworks, into which the custom code has to embed itself. This kind of reuse is known as whitebox reuse: The internal code structures have to be known – not only the interface. This type of reuse requires a detailed understanding of the code which is to be reused which sets a high hurdle for the reuse.

An example can be a library which facilitates the generation of metrics for the system monitoring. It will be used in the billing Microservice. Other teams also want to use the code. Therefore, the code is extracted into a library. Since it is technical code, it is not modified in case of domain-based changes. Therefore, the library does not influence the independent deployment and the independent development of domain-based features. The library was supposed to be turned into an internal open source project (compare section 13.7).

However, to transfer domain code into a shared library is problematic, as it might introduce deployment dependencies into Microservices. When, for instance, the modeling of a customer is implemented in a library, then each change to the data structure has to be passed on to all Microservices, and they all have to be newly deployed. Besides, a uniform modeling of a data structure like customer is anyhow hardly possible due to *Bounded Context*.

Transfer Code

Another option for changing the architecture is to transfer code from one Microservice to another. This is sensible when thereby a loose coupling and a high cohesion of the entire system can be ensured. When two Microservices communicate a lot, the loose coupling is not ensured. When the part of the Microservice is transferred which communicates a lot with the other Microservice, this problem can be solved.

This approach is similar to the removal into a shared library. However, the code is no common dependency, which solves the problem of coupling between the Microservices. However, it is possible that the Microservices have to have a

common interface in order to still be able to use the functionalities after the code transfer. This is a blackbox dependency: Only the interface has to be known, but not the internal code structures.

In addition, it is possible to transfer the code into another Microservice, while keeping it in the original Microservice. This causes redundancies. Errors will then have to be corrected in both versions. And the two versions can develop into different directions. However, on the other hand the Microservices are independent, especially in regards to deployment.

The technological limitations are still the same as for a shared library – the two Microservices have to use similar technologies because otherwise the code cannot be transferred. However, in a pinch the code can also be rewritten in a new programming language or with a different programming model. Microservices are not very large. The code which has to be rewritten is only a part of a Microservice. Consequently, the required effort is manageable.

However, there is the problem that the size of that Microservice into which the code is transferred increases. Thus the danger increases that the Microservice turns into a monolith over time.

One example: The Microservice for the order process frequently calls the billing Microservice in order to calculate the price for the delivery. Both services are written in the same programming language. The code is transferred from the one Microservice into the other. From a domain perspective it turns out that the calculation of delivery costs rather belongs into the order process Microservice. The code transfer is only possible when both services use the same platform and programming language. Moreover, the communication across Microservices has to be replaced by local communication.

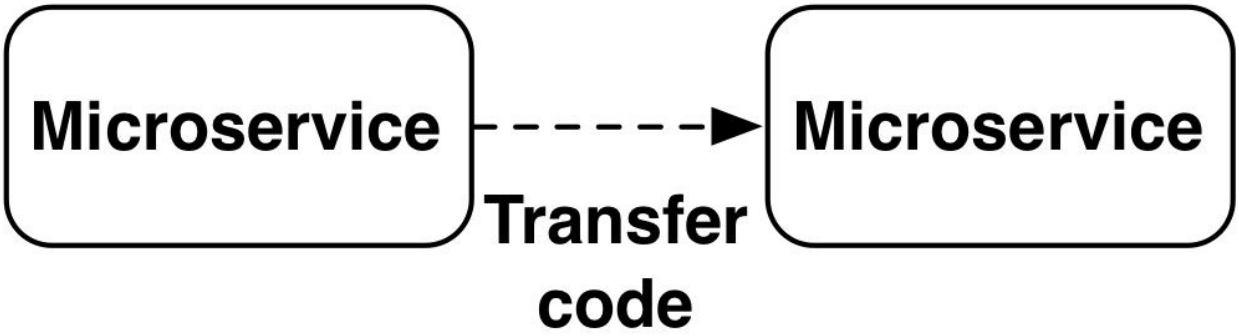


Fig. 24: Transfer Code

Reuse or Redundancy?

Instead of attributing shared code to one or the other Microservice, the code can also be maintained in both Microservices. At first this sounds dangerous – after all, the code will then be redundant in two places, and bug fixes have accordingly to be performed in both places. Most of the time developers try to avoid such situations. An established best practice is “Don’t Repeat Yourself” (DRY). Each decision and consequently all code should only be stored at exactly one place in the system. In Microservice-based architectures redundancy has a decisive advantage: The two Microservices stay independent of each other and can be independently deployed and independently developed further. In this way the central characteristic of Microservices is preserved.

Moreover, it is questionable whether a system can be built without any redundancies at all. Especially in the beginning of object-orientation many projects invested a lot of effort to transfer shared code into shared frameworks and libraries. This was meant to decrease the expenditure associated with the creation of the individual projects. In reality the code to be reused was often difficult to understand and thus hard to use. A redundant implementation in the different projects might have been the better alternative. It can be less laborious to implement code several times than to design it in a reusable manner and to actually reuse it.

There are of course successful reuses of code: Hardly any project can get along nowadays without open source libraries. At this level code reuse is taking place all the time. This approach can be a good template for the reuse of code between Microservices. However, this has effects on the organization. Section 13.7 discusses organization and thereby also code reuse according to an open source model.

Shared Service

Instead of transferring the code into a library, it can also be moved into a new Microservice (compare Fig. 25). Thereby the typical advantages of a Microservice-based architecture ensue: The technology of the new Microservice does not matter. As long as it uses the universally defined communication technologies and can be operated like the other Microservices, its internal structure can be arbitrary – to the point of programming language.

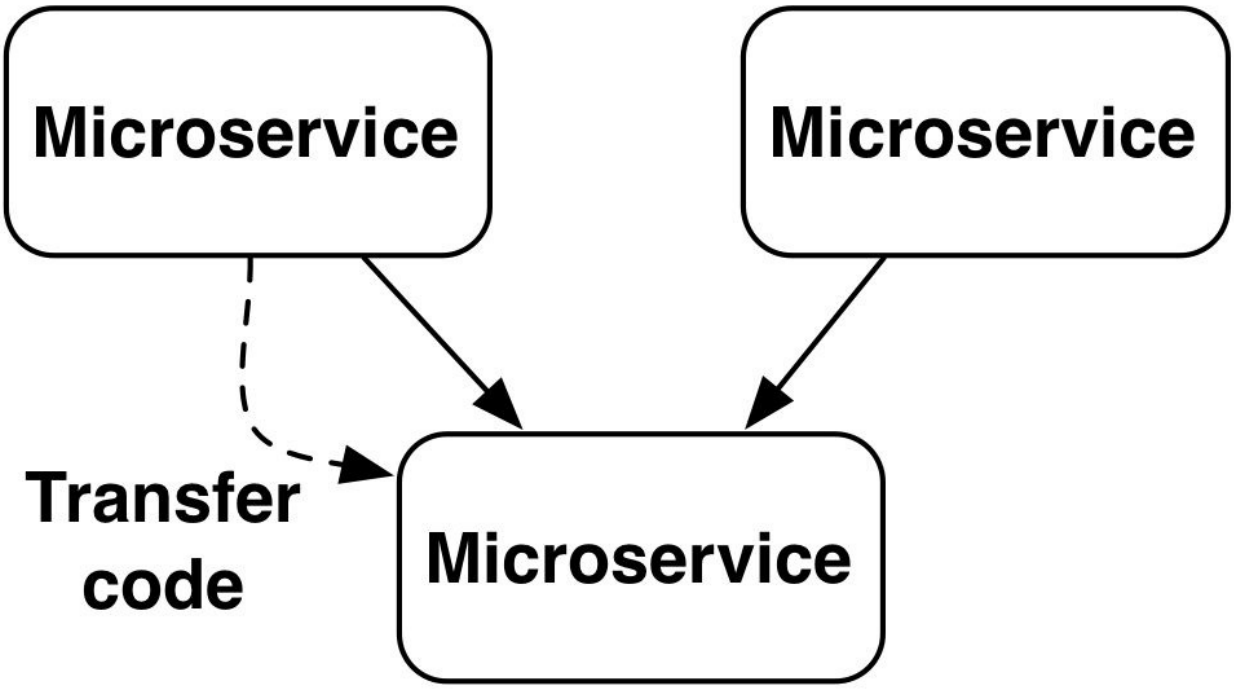


Fig. 25: Shared Microservice

The use of a Microservice is simpler than the use of a library. Only the interface of the Microservice has to be known – the internal structure does not matter. Moving code into a new service decreases the average size of a Microservice – and therefore the intelligibility and replaceability of the Microservices. However, the transfer replaces local calls with calls via the network. Changes for new features might not be limited to one Microservice anymore.

In software development big modules are often a problem. So transferring code into new Microservices can be a good option for keeping the modules small. Besides, the new Microservice can be further developed by the team which was already responsible for the original Microservice. This will facilitate the close coordination of new and old Microservices since the required communication happens within only one team.

The split into two Microservices has also the consequence that a call to the Microservice-based system is not processed by just one single Microservice, but by several Microservices. These Microservices call each other. Some of those Microservices will not have a UI, but are pure backend services.

To illustrate this, let us turn again to the order process, which frequently calls the billing Microservice for calculating the delivery costs. The calculation of delivery costs can also be separated into an Microservice by itself. This is even

possible when the billing service and the order process Microservice use different platforms and technologies. However, a new interface will have to be established, which enables the new delivery cost Microservice to communicate with the remainder of the billing service.

Spawn a New Microservice

In addition, it is also possible to use part of the code of a certain Microservice to generate a new Microservice (compare Fig. 26). The advantages and disadvantages are identical to the scenario in which code is transferred into a shared Microservice. However, the motivation is different in this case: The size of the Microservices is meant to be reduced to increase their maintainability or maybe to transfer the responsibility for a certain functionality to another team. Here, the new Microservice is not supposed to be shared by multiple other Microservices.

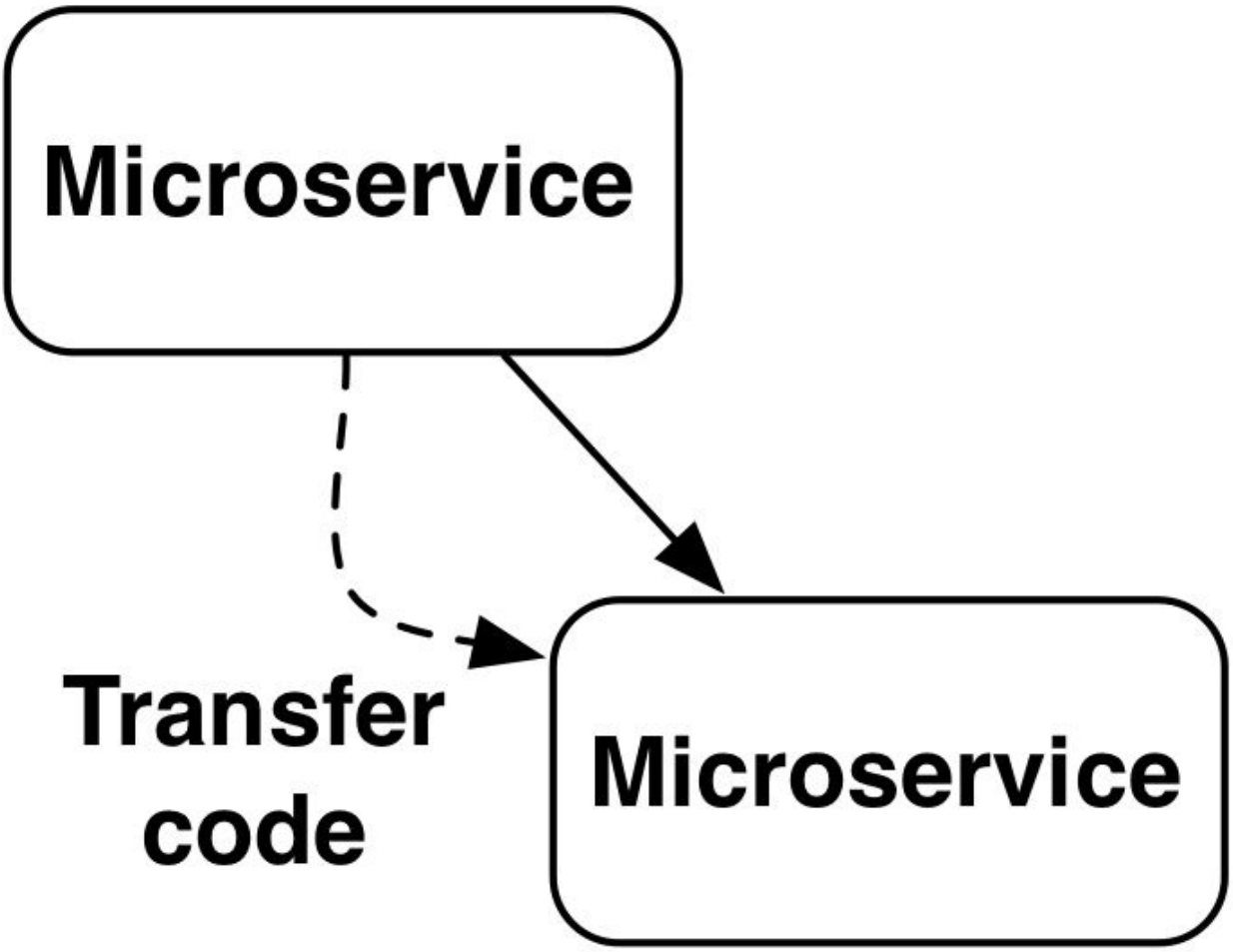


Fig. 26: Spawning a new Microservice

For instance, the service for the registration might have become too complex in the meantime. Therefore, it is distributed into multiple services, which each handle certain user groups. A technical distribution would also be possible – for instance according to CQRS (compare section 10.2), Event Sourcing (section 10.3) or Hexagonal Architecture (section 10.4).

Rewriting

Finally, an additional possibility to handle Microservices, whose structure does not fit anymore, is to rewrite them. Due to the small size of Microservices and because of their use via defined interfaces this possibility is much more feasible with Microservices than in the case of other architectural approaches. In the end, not the entire system has to be rewritten, but just a part. It is also possible to implement the new Microservice in a different programming language, which is maybe better suited for this purpose. Rewriting Microservices can also be advantageous since new insights about the domain can leave their mark on the new implementation in this manner.

A Growing Number of Microservices

The experience with Microservice-based systems teaches that during the time a project is running new Microservices will permanently be generated. This entails a higher effort for the infrastructure and the operation of the system. The number of deployed services will increase all the time. For classical projects such a development is unusual and appears therefore problematic. However, as this section demonstrated, the generation of new Microservices is the best alternative for the shared use of logic and for the ongoing development of a system. Besides the growing number of Microservices ensures that the average size of individual Microservices stays constant. Consequently, the positive characteristics of Microservices are preserved.

Generating new Microservices should be as easy as possible as this allows to preserve the properties of the Microservice system. Potential for optimization is mainly present when it comes to establishing Continuous Delivery pipelines, a build infrastructure and the required server for the new Microservice. Once these things are automated, new Microservices can be generated comparably easily.

Microservice-based Systems Are Hard to Modify

This section has shown that it is difficult to adjust the overall architecture of a Microservice-based system. New Microservices have to be generated. This

entails changes to the infrastructure and the need for additional Continuous Delivery pipelines. Shared code in libraries is rarely a sensible option.

In a Deployment Monolith such changes would be easy to introduce: Often the integrated development environments even automatize the transfer of code or other structural changes. Due to automation the changes are less laborious and less prone to errors. Besides, there are no effects whatsoever on the infrastructure or Continuous Delivery pipelines in the case of Deployment Monoliths.

Thus, changes are difficult at the level of the entire system – because it is hard to transfer functionalities between different Microservices. In the end, this is exactly the effect, which was termed “strong modularization” and listed as advantage in section 1.2: To cross the boundaries between Microservices is difficult so that the architecture at the level between the Microservices will also remain intact in the long-run. However, this entails as well that the architecture is hard to adjust at this level.

Try and Experiment

A developer has written a helper class, which facilitates the interaction with a logging framework, which is also used by other teams. It is not very large and complex.

- Should it be used by other teams?
- Should the helper class be turned into a library or an independent Microservice or should the code simply be copied?

8.4 Growing Microservice-based Systems

Microservices primarily have advantages in very dynamical environments. Due to the independent deployment of individual Microservices, teams can work in parallel on different features without much need for coordination. This is especially advantageous when it is unclear which features are really meaningful and experiments on the market are necessary to identify the promising approaches.

Planning Architecture?

Especially in such an environment it is hardly possible to plan a good split of the domain logic into Microservices right from the start. The architecture has to adjust to the facts.

- The split according to domain aspects is even more important for Microservices than in the context of a classical architecture approach. This is due to the fact that the domain-based distribution influences also the distribution into teams and therefore the independent working of the teams – the central advantage of Microservices (section 8.1).
- Section 8.2 demonstrated that tools for architecture management cannot readily be used in Microservice-based architectures.
- As section 8.3 discussed, it is difficult to modify the architecture of Microservices – especially in comparison to Deployment Monoliths.
- Microservices are especially advantageous in dynamic environments – where it is even more difficult to determine a meaningful architecture right from the start.

The architecture has to be changeable, however, this is difficult due to the technical facts. This section shows how the architecture of a Microservice-based system can nevertheless be modified and developed further in a stepwise manner.

Start Big

One possibility to handle this inherent problem is to start out with several big systems, which are subsequently step by step fragmented into Microservices. Section 4.1 defined as upper limit for the size of a Microservice the amount of code which an individual team can still handle. At least at the outset of a project it is hard to violate this upper limit. The same is true for the other upper limits: modularization and replaceability.

When the entire project consists only of one or few Microservices, functionalities are still easy to move since the transfer will mostly occur within one service rather than between services. Step by step more people can be moved into the project so that additional teams can be assembled. In parallel the system can be distributed into progressively more Microservices to allow the teams to work independently of each other. Such a ramp-up is also for organizational reasons a good approach since the teams can be assembled in a stepwise manner.

Of course, it would also be possible to start off with a Deployment Monolith. However, starting with a monolith has a decisive disadvantage: There is the danger that dependencies and problems creep into the architecture, which preclude a later distribution into Microservices. Besides, in that case there will be only one Continuous Delivery pipeline. When the monolith gets distributed into Microservices, the teams will have to generate new Continuous Delivery

pipelines. This can be very laborious, especially when the Continuous Delivery pipeline for the Deployment Monolith had been generated manually. In that case all the additional Continuous Delivery pipelines would likewise have to be manually generated in a laborious manner.

When the projects start from the beginning with multiple Microservices, this problem is avoided. There is no monolith which later would have to be distributed, and there anyhow has to be an approach for the generation of new Continuous Delivery pipelines. Thus the teams can from the start work independently on their own Microservices. Over the course of the project the initial Microservices are distributed into additional smaller Microservices.

Start Big corresponds to the observation that the number of Microservices will increase over the course of the project. In line with this it is sensible to start with few big Microservices and to spawn new Microservices in a stepwise manner. Thereby the most current insights can always be integrated into the distribution into Microservices. It is just not possible to define the perfect architecture right from the start. Instead the teams should adapt the architecture step by step to the new circumstances and insights and have the courage to implement the necessary changes.

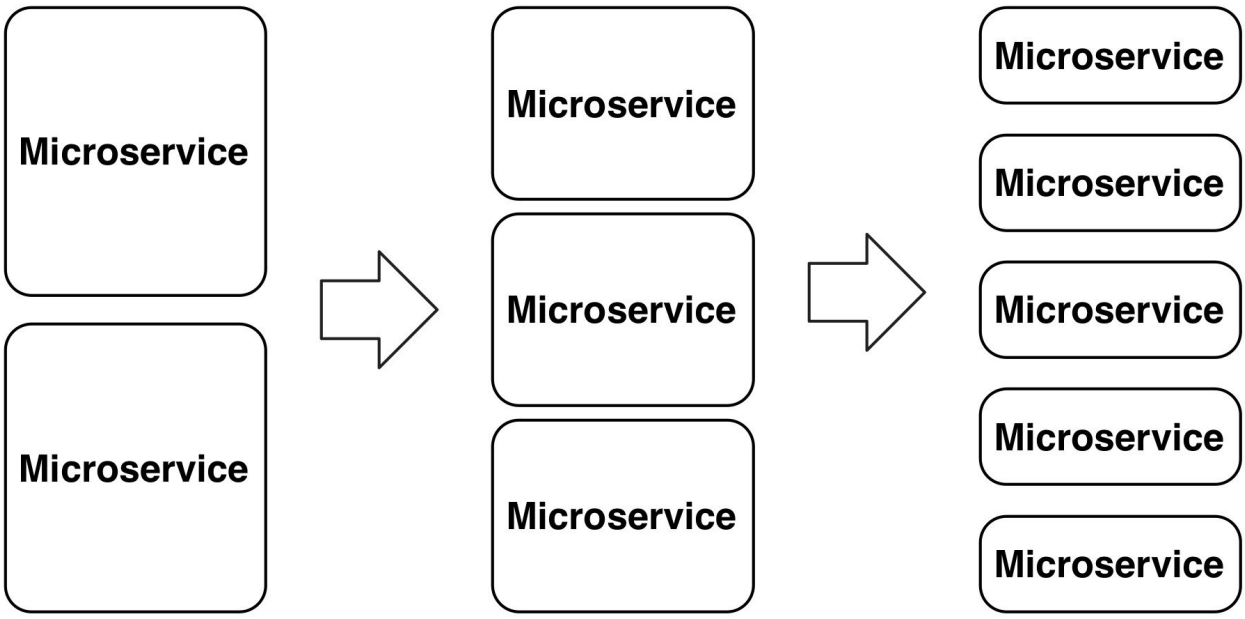


Fig. 27: Start Big: From few Microservices originate progressively more Microservices.

This approach results in a uniform technology stack – this will facilitate operation and deployment. For developers it is also easier to work on other Microservices.

Start Small?

It is also imaginable to start with a distribution into a large number of Microservices and to use this distribution as basis for further development. However, the distribution of the services is very difficult. “Building Microservices” [3] provides an example where a team was supposed to develop a tool for the support of Continuous Delivery as a Microservice-based system. The team was very familiar with the domain, had already created products in this area and thus chose an architecture, which distributed the system early on into numerous Microservices. However, as the new product was supposed to be offered in the cloud, the architecture was, for subtle reasons, not suitable in some respects. To implement changes got difficult because modifications for features had to be introduced in multiple Microservices. To solve this problem and make it easier to change the software, the Microservices were united again into a monolith. One year later the team distributed the monolith again into Microservices and thereby decided the final architecture. This example demonstrates that a too early distribution into Microservices can be problematic – even if a team knows the domain very well.

Limits of Technology

However, this is in the end a limitation of the technology. If it were easier to move functionalities between Microservices (compare section 8.4), the split into Microservices could be corrected. In that case it would be much less risky to start off with a split into small Microservices. When all Microservices use the same technology, it is easier to transfer functionalities between them. Chapter 15 discusses technologies for Nanoservices, which are based on a number of compromises, but in exchange allow for smaller services and an easier transfer of functionalities.

Replaceability as a Quality Criterion

An advantage of the Microservice approach is the replaceability of the Microservices. This is only possible when the Microservices do not grow beyond a certain size and internal complexity. One aim during the continued development of Microservices is to maintain the replaceability of Microservices. Then a Microservice can be replaced by a different implementation – for instance in the case that its further development is not feasible anymore due to its bad structure. In addition, replaceability is a meaningful aim to preserve the intelligibility and maintainability of the Microservice. If the Microservice is not replaceable

anymore, it is probably also not intelligible anymore and therefore hard to develop any further.

The Gravity of Monoliths

One problem is that large Microservices attract modifications and new features. They cover already several features; therefore, it seems a good idea to implement new features also in this service. This is true in the case of too large Microservices, but even more so for Deployment Monoliths. A Microservices-based architecture can be aimed at replacing a monolith. However, in that case the monolith contains so many functionalities, that care is needed not to introduce too many changes into the monolith. For this purpose, Microservices can be created, even if they contain hardly any functionalities at the beginning. To introduce changes and extensions to the monolith is exactly the course of action that has rendered the maintenance of the Deployment Monolith impossible and led to its replacement by Microservices.

Keep Splitting

As mentioned, most architectures do not have the problem that they were originally planned in a way that did not fit the task. In most cases the problem is rather that the architecture did not keep up with the changes in the environment. A Microservice-based architecture also has to be constantly adjusted, otherwise it will at some point not be able anymore to support the requirements. To these adjustments belong a management of the domain-based split as well as of the size of the individual Microservices. This is the only way to ensure that the advantages of the Microservice-based architecture are maintained over time. Since the code amount of a system usually increases, the number of Microservices will grow as well in order to keep the average size constant. Thus an elevation of the number of Microservices is not a problem, but rather a good sign.

Global Architecture?

However, not only the size of Microservices can be a problem. The dependencies of the Microservices can also cause problems (compare section 8.1). Such problems can be solved most of the time by adjusting a number of Microservices – i.e. those which have problematic dependencies. This requires only contributions from the teams, which work on these Microservices. These teams are also the ones to spot the problems, because they will be affected by the bad architecture and the greater need for coordination. By modifying the architecture, they are able to solve these issues. In that case there is no need for a global management of dependencies. Metrics like a high number of dependencies or

cyclic dependencies can only be an indication for a problem. Whether such metrics indeed indicate a problem can only be solved by evaluating them together with the involved teams. If the problematic components are, for instance, not going to be developed any further in the future, it does not matter whether the metrics indicate a problem. Maybe there have for other reasons never been problems during development. Even if there is a global architecture management, it can only work effectively in close cooperation with the different teams.

Don’t Miss the Exit Point or How to Avoid the Erosion of a Microservice (Lars Gentsch)

by Lars Gentsch, E-Post Development GmbH

Practically, it is not too difficult to develop a Microservice. But how can you ensure that the Microservice remains a Microservice and does not secretly become a monolith? An example shall illustrate at which point a service starts to develop into the wrong direction and which measures are necessary to ensure that the Microservice remains a Microservice.

Let’s envision a small web application for customer registration. This scenario can be found in nearly every web application. A customer wants to buy a product in an Internet shop (Amazon, Otto etc.) or to register for a video-on-demand portal (Amazon Prime, Netflix etc.). As a first step the customer is led through a small registration workflow. He/she is asked for his/her username, a password, the email address and the street address. This is a small self-contained functionality, which is very well suited for a Microservice.

Technologically this service has probably a very simple structure. It consists of two or three HTML pages or an AngularJS-Single Page App, a bit of CSS, some Spring Boot and a MySQL database. Maven is used to build the application.

When data are entered, they are concomitantly validated, transferred into the domain model and put into the database for persistence. How can the Microservice grow step-by-step into a monolith?

Incorporation of New Functionality

Via the shop or the video-on-demand portal items and content are supposed to be delivered, which are only allowed to be accessed by people who are of age. For this purpose the age of the customer has to be verified. One possibility to do this

is to store the birth date of the client together with other data and to incorporate an external service for the age verification.

Thus, the data model of our service has to be extended by the birth date. More interesting is the incorporation of the external service. To achieve this, a client for an external API has to be written, which should also be able to handle error situations like the non-availability of the provider.

It is highly probable that the initiation of the age verification is an asynchronous process so that our service might be forced to implement a callback interface. So the Microservice must store data about the state of the process. When was the age verification process initiated? Is it necessary to remind the customer via email? Was the verification process successfully completed?

What is Happening to the Microservice here?

1. The customer data are extended by the birthdate. That is not problematic.
2. In addition to customer data there are now process data. Attention: Here process data are mixed with domain data.
3. In addition to the original CRUD functionality of the service, some kind of workflow is now required. Synchronous processing is mixed with asynchronous processing.
4. An external system is incorporated. The testing effort for the registration Microservice increases. An additional system and its behavior have to be simulated during test.
5. The asynchronous communication with the external system has other demands in regards to scaling. While the registration Microservice requires estimated ten instances due to load and failover, the incorporation of the age verification can be operated in a fail-safe and stable manner with just two instances. Thus, different run time requirements are mixed here.

As the example demonstrates, a per se small requirement like the incorporation of an age verification can have tremendous consequences for the size of the Microservice.

Criteria Arguing for a new Microservice Instead of Extending an Existing One:

1. Introduction of different data models and data (domain vs. process data)
2. Intermixture of synchronous and asynchronous data processing
3. Incorporation of additional services

4. Different load scenarios for different aspects within one service

The example of the registration service could be further extended: Also the verification of the customer's street address could be performed by an external provider. This is common in order to ensure the existence of the denoted address. Another scenario is the manual clearance of a customer in case of double registration. The incorporation of a solvency check or customer scoring upon registration is likewise a frequent scenario.

All these domain-based aspects belong in principle to the customer registration and tempt developers and architects to integrate the corresponding requirements into the existing Microservice. Thereby the Microservice grows into more than just one Microservice.

How to Recognize Whether the Initiation of a new Microservice Should Have Occurred Already?

1. The service can only be sensibly developed further as Maven multi modul project or Gradle multi module project.
2. Tests have to be divided into test groups and have to be parallelized for execution since the run time of the tests surpasses five minutes (violation of the "fast feedback" principle).
3. The configuration of the service is grouped by domain within the configuration file or the file is divided into single configuration files to improve the overview.
4. A complete build of the service takes long enough to make a coffee break. Fast feedback cycles are not possible anymore (violation of the "fast feedback" principle).

Conclusion

As the example of the registration Microservice illustrates, it is a big challenge to let a Microservice remain a Microservice and not give in to the temptation to integrate new functionalities into an existing Microservice due to time pressure. This holds even true when the functionalities clearly belong, like in the example, to the same domain.

What can prophylactically be done to prevent the erosion of a Microservice? In principle, it has to be as simple as possible to create new services including their own data storage. Frameworks like Spring Boot, Grails and Play make a relevant contribution to this. The allocation of project templates like Maven archetypes and

the use of container deployments with Docker are additional measures to simplify the generation and configuration of new Microservices as well as their way into the production environment as much as possible. By reducing the “expenditure” for the setting up of a new service the inhibition threshold for the introduction of a new Microservice decreases clearly and thus the temptation to implement new functionalities into existing services.

8.5 Microservices and Legacy Applications

The transformation of a legacy application into a Microservice-based architecture is a scenario which is frequently met with in practice. Completely new developments are rather rare, and Microservices first of all promise advantages for long term maintenance. This is especially interesting for applications which are already on the brink of not being maintainable anymore. Besides the distribution into Microservices allows for an easier handling of Continuous Delivery: Instead of deploying and testing a monolith in an automated fashion small Microservices can be deployed and tested. The expenditure for this is by far lower. A Continuous Delivery pipeline for a Microservice is not very complex – however, for a Deployment Monolith the expenditure can be very large. This advantage is sufficient for many companies to justify the effort of migrating to Microservices.

In comparison to building up completely new systems there are some important differences when migrating from a Deployment Monolith to Microservices:

- For a legacy system the functionality is clear from the domain perspective. This can be a good basis for generating a clean domain architecture for the Microservices. Especially such a clean domain-based division is very important for Microservices.
- However, there is already a large amount of code in existence. The code is often of bad quality. There are few tests, and deployment times are often much too long. Microservices should remove these problems. Accordingly, the challenges in this area are often significant.
- Likewise it is well possible that the module boundaries in the legacy application do not answer to the *Bounded Context* idea (compare section 4.3). In that case migrating to a Microservice-based architecture is a challenge because the domain-based design of the application has to be changed.

Breaking up Code?

In a simple approach the code of the legacy application can be split into several Microservices. This can be problematic when the legacy application does not have a good domain architecture, which is often the case. The code can be especially easily split into Microservices when the Microservices are geared to the existing modules of the legacy application. However, when those have a bad domain-based split, this bad division will be passed on to the Microservice-based architecture. And the consequences of a bad domain-based design are even more profound in a Microservice-based architecture: The design influences also the communication between teams. Besides, the initial design is hard to change later on in a Microservice-based architecture.

Supplementing Legacy Applications

However, it is also possible to get by without a division of the legacy application. An essential advantage of Microservices is that the modules are distributed systems. Due to that the module boundaries are at the same time the boundaries of processes which communicate via the network. This has advantages for the distribution of a legacy application: It is not at all necessary to know the internal structures of the legacy application or, based on that, to perform a split into Microservices. Instead Microservices can supplement or modify the legacy application at the interface. For this it is very helpful when the system to be replaced is already built in a SOA (section 7.2). If there are individual services, they can be supplemented by Microservices.

Enterprise Integration Patterns

Enterprise Integration Patterns [4] offer an inspiration for possible integrations of legacy applications and Microservices:

Designing, Building, and Deploying Messaging Solutions, Addison-Wesley Longman, 2003, ISBN 978-0-32120-068-6

- *Message Router* describes that certain messages go to another service. A Microservice can select some messages which are processed then by the Microservice instead of by the legacy application. Thereby the Microservice-based architecture does not have to newly implement the entire logic at once, but can at first select some parts.
- A special router is the *Content Based Router*. It determines based on the content of a message where the message is supposed to be sent. This allows to send specific messages to a specific Microservice – even if the message differs only in one field.

- The *Message Filter* avoids that a Microservice receives uninteresting messages. For that it just filters all messages out the Microservice is not supposed to get.
- A *Message Translator* translates a message into another format. Thereby the Microservices architecture can use other data formats and does not necessarily have to employ the formats used by the legacy application.
- The *Content Enricher* can supplement data in the messages. If a Microservice requires supplementary information in addition to the data of the legacy application, the *Content Enricher* can add this information without the legacy application or the Microservice noticing anything.
- The *Content Filter* achieves the opposite: Certain data are removed from the messages so that the Microservice obtains only the information which is relevant for it.

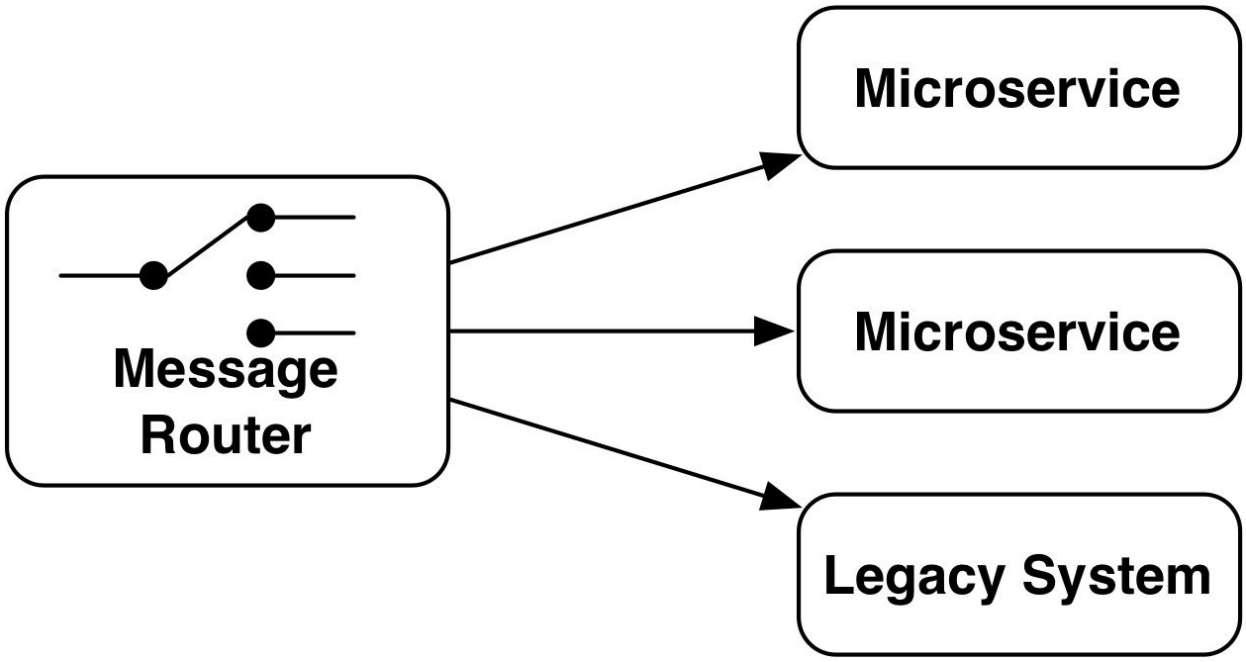


Fig. 28: Supplementing legacy applications by a *Message Router*

Fig. 28 shows a simple example: A Message Router takes calls and sends them to a Microservice or the legacy system. This allows to implement certain functionalities in Microservices. These functionalities are also still present in the legacy system – but are not used there anymore. In this way the Microservices are largely independent of the structures within the legacy system. For instance, Microservices can start off with processing orders for certain customers or certain items. Thereby they do not have to implement all special cases.

The patterns can serve as inspiration how a legacy application can be supplemented by Microservices. There are numerous additional patterns – the list provides only a glimpse of the entire catalog. Like in other cases the patterns can be implemented in different ways: Actually, they focus on messaging systems. But it is possible to implement them with synchronous communication mechanisms – even though less elegant. For instance, a REST service can take a POST message, supplement it with additional data and finally send it to another Microservice. That would then be a *Content Enricher*.

To implement such patterns, the sender has to be uncoupled from the recipient. This enables the integration of additional steps into the processing of requests without the sender noticing anything. In case of a messaging approach this is easily possible as the sender knows only one queue in which he/she places the messages. The sender does not know who fetches the messages. However, in the case of synchronous communication via REST or SOAP the message is sent directly to the recipient. Only by Service Discovery (compare section 8.9) the sender gets uncoupled from the recipient. Then one service can be replaced by another service without need to change the senders. This allows for an easier implementation of the patterns. When the legacy application is supplemented by a *Content Enricher*, this *Content Enricher* instead of the legacy application is registered in the Service Discovery, but no sender has to be modified. To introduce Service Discovery can therefore be a first step towards a Microservices architecture, since it allows to supplement or replace individual services of the legacy application without having to modify the users of the legacy application.

Limiting Integration

Especially for legacy applications it is important that the Microservices are not too dependent on the legacy application. Often it is especially the bad structure of the old application which is the reason why the application is supposed to be replaced in the first place. Therefore, certain dependencies should not be allowed at all. When Microservices directly access the database of the legacy application, the Microservices are dependent on the internal data representation of the legacy application. Besides neither the legacy application nor the Microservices can still change the schema since such changes have to be implemented in Microservices and legacy application. The shared use of a database in legacy application and Microservices has to be avoided on all accounts. However, to replicate the data of the legacy application into an separate database schema is of course still an option.

Advantages

It is an essential advantage of such an approach that the Microservices are largely independent of the architecture of the legacy application. And the replacement of a legacy application is mostly initiated because its architecture is not sustainable any more. Besides, this allows to supplement systems by Microservices, which are actually not at all meant to be extended. Though, for instance, standard solutions in the area of CRM, E-commerce or ERP are internally extensible, their extension by external interfaces can be a welcome alternative since such a supplement is often easier. Moreover, such systems often attract functionalities, which do not really belong there. A distribution into a different deployment unit via a Microservice ensures a permanent and clear delimitation.

Integration via UI and Data Replication

However, this approach only tackles the problem on the level of logic integration. Chapter 9 describes another level of integration, namely data replication. This allows a Microservice to access also comprehensive datasets of a legacy application with good performance. It is important that the replication does not happen based on the data model of the legacy application. In that case the data model of the legacy application would practically not be changeable anymore since it is also used by the Microservice. An integration based on the use of the same database would be even worse. Also at the level of UI integrations are possible. Especially links in web applications are attractive since they cause only few changes in the legacy application.

Content Management Systems

In this manner Content Management Systems (CMS), for instance, which often contain many functionalities, can be supplemented by Microservices. CMS contain the data of a website and administrate the content so that editors can modify it. The Microservices take over the handling of certain URLs. Similar to a *Message Router* an HTTP request can be sent to a Microservice instead of to the CMS. Or the Microservice changes elements of the CMS like in the case of a *Content Enricher* or modifies the request like in the case of a *Message Translator*. Lastly, the Microservices could store data in the CMS and thereby use it as a kind of database. Besides JavaScript representing the UI of a Microservice can be delivered into the CMS. In that case the CMS turns into a tool for the delivery of code in a browser.

Some examples could be:

- A Microservice can import content from certain sources. Each source can have its own Microservice.
- The functionality which allows a visitor of the web page e.g. to follow an author can be implemented in a separate Microservice. The Microservice can either have its own URL and be integrated via links or it modifies the pages, which the CMS delivers.
- While an author is still known in the CMS, there is other logic which is completely separate from the CMS. This could be vouchers or E-commerce functionalities. Also in this case a Microservice can appropriately supplement the system.

Especially in the case of CMS systems, which create static HTML, Microservices-based approaches can be useful for dynamic content. The CMS moves into the background and is only necessary for certain content. There is a monolithic deployment of the CMS content while the Microservices can be deployed much more rapidly and in an independent manner. In this context the CMS is like a legacy application.

Conclusion

The integrations all have the advantage that the Microservices are not bound to the architecture or the technology decisions of the legacy application. This provides the Microservices with a decisive advantage compared to a modifications of the legacy application. However, the migration away from the legacy application using this approach poses a challenge at the level of architecture: In effect, Microservice-based systems have to have a well structured domain-based design to enable the implementation of features within one Microservice and by an individual team. In case of a migration, which follows the outlined approach, this cannot always be put into effect since the migration is influenced by the interfaces of the legacy application. Therefore, the design cannot always be as clear-cut as desirable. Besides, domain-based features will still be also implemented in the legacy application until a large part of the migration has been completed. During this time the legacy application cannot be finally removed. When the Microservices confine themselves to transforming the messages, the migration can take a very long time.

No Big Bang

The outlined approaches suggest that the existing legacy application is supplemented in a stepwise manner by Microservices or that individual parts of the legacy application are replaced by Microservices. This type of approach has

the advantage that the risk is minimized. Replacing the entire legacy application in one single step entails a high risk due to the size of the legacy application. In the end, all functionalities have to be represented in the Microservices. In this process numerous mistakes can creep in. In addition, the deployment of Microservices is complex as they all have to be brought into production in a concerted manner in order to replace the legacy application in one step. A stepwise replacement nearly imposes itself in the case of Microservices since they can be deployed independently and supplement the legacy application. Thereby the legacy application can be replaced by Microservices in a stepwise manner.

Legacy = Infrastructure

Part of a legacy application can also simply be continued to be used as infrastructure for the Microservices. For example, the database of the legacy application can also be used for the Microservices. It is important that the schemas of the Microservices are separate from each other and also from the legacy application. After all, the Microservices should not be closely coupled.

The use of the database of the legacy application does not have to be mandatory for the Microservices. Microservices can definitely also use other solutions. However, the existing database is established in regards to operation or backup. Using this database can also for the Microservices present an advantage. The same is true for other infrastructure components. A CMS for instance can likewise serve as common infrastructure, to which functionalities are added from the different Microservices and into which the Microservices can also deliver content.

Other Qualities

The so far introduced migration approaches focus on enabling the domain-based division into Microservices in order to facilitate the long-term maintenance and continued development of the system. However, Microservices have many additional advantages. When migrating it is important to understand which advantage motivates the migration to Microservices because depending on this motivation an entirely different strategy might be adopted. Microservices offer for instance also increased robustness and resilience since the communication with other services is taken care of accordingly (compare section 10.5). If the legacy application currently has a deficit in this area or a distributed architecture already exists, which has to be optimized in respect to these points, appropriate

technology and architecture approaches can be defined without necessarily requiring that the application has to divided into Microservices.

Try and Experiment

Do research on the remaining Patterns of Enterprise Integration:

- Can they be meaningfully employed when dealing with Microservices? In which context?
- Can they really only be implemented with messaging systems?

Hidden Dependencies (Oliver Wehrens)

by Oliver Wehrens, E-Post Development GmbH

In the beginning there is the monolith. Often it is sensible and happens naturally that software is created as a monolith. The code is clearly arranged, and the business domain is just coming into being. In that case it is better when everything has a common base. There is a UI, business logic and a database. Refactoring is simple, deployment is easy, and everybody can still understand the entire code.

Over time the amount of code grows, and it gets hard to see through. Not everybody knows all parts of the code anymore. The compiling takes longer, and the unit and integration tests invite developers to take a coffee break. In case of a relatively stable business domain and a very large code basis many projects will consider at this point the option to distribute the functionality into multiple Microservices.

Depending on the status of the business and the understanding of the business/product owners the necessary tasks will be completed. Source code is distributed, Continuous Delivery pipelines are created and server provisioned. During this step no new features are developed. The not negligible effort is justified just by the hope that in future features will be faster and more independently created by other teams. Developers are going to be very assured of this, other stakeholders often have to be convinced first.

In principle everything has been done to reach a better architecture. There are different teams which have independent source code. They can bring their software at any time into production and independent of other teams.

Almost.

The Database

Every developer has a more or less pronounced affinity to the database. In my experience many developers view the database as necessary evil, which is somewhat cumbersome to refactor. Often tools are being used which generate the database structure for the developers (e.g. Liquibase or Flyway in the JVM area). Tools and libraries (Object Relation Mapper) render it very easy to persist objects. A few annotations later and the domain is saved in the database.

All these tools remove the database from the typical developers, who “only” want to write their code. This has sometimes the consequence that there is not much attention given to the database during the development process. For instance, indices which were not created will slow down searches on the database. This will not show up in a typical test, which does not work with large data amounts, and thus go like that into production.

Let’s take the fictional case of an online shoe shop. The company requires a service which allows users to log in. A user service is created containing the typical fields like ID, first name, family name, address and password. To now offer fitting shoes to the users, only a selection of shoes in their actual size is supposed to be displayed. The size is registered in the welcome mask. What could be more sensible than to store this data in the already existing user service? Everybody is sure: These are user-associated data, and this is the right location.

Now the shoe shop expands and starts to sell additional types of clothing. Dress size, collar size and all other related data are now also stored in the user service.

Several teams are employed in the company. The code gets progressively more complex. It is the point in time, where the monolith is split into domain-based services. The refactoring in the source code works well, and a soon the monolith is split apart into many Microservices.

Unfortunately, it turns out that it is still not easy to introduce changes. The team in charge of shoes wants to accept different currencies because of international expansion and has to modify the structure of the billing data including the address format. During the upgrade the database is blocked. Meanwhile no dress size or favorite color can be changed. Moreover, the address data are used in different

standard forms of other services and thus cannot be changed without coordination and effort. Therefore the feature cannot be implemented promptly.

Even though the code is well separated, the teams are indirectly coupled via the database. To rename columns in the user service database is nearly impossible because nobody knows anymore in detail who is using which columns. Consequently, the teams do workarounds. Either fields with the name ‘Userattribute1’ are created, which then are mapped onto the right description in the code, or separations are introduced into the data like ‘#Color:Blue#Size:10’. Nobody except the involved team knows what is meant by ‘Userattribute1’, and it is difficult to generate an index on ‘#Color:#Size. Database structure and code are progressively harder to read and to maintain.

It has to be essential for every software developer to think about how to persist the data. This means: not only about the database structures, but also about where which data is stored. Is the table respectively database the place where these data should be located? From a business domain perspective do these data have connections to other data? In order to remain flexible in the long term, it is worthwhile to carefully consider these questions every time. Typically, databases and tables are not created very often. However, they are a component which is very hard to modify later. Besides, databases and tables are often the origin of a hidden interdependence between services. In general, it has to apply that data can only be used by exactly one service via direct database access. All other services, which want to use the data, may only access it via the public interfaces of the service.

8.6 Event-driven Architecture

Microservices can call each other in order to implement shared logic. For example, at the end of the order process the Microservice for billing as well as the Microservice for the order execution can be called to create the bill and make sure that the ordered items are indeed delivered.

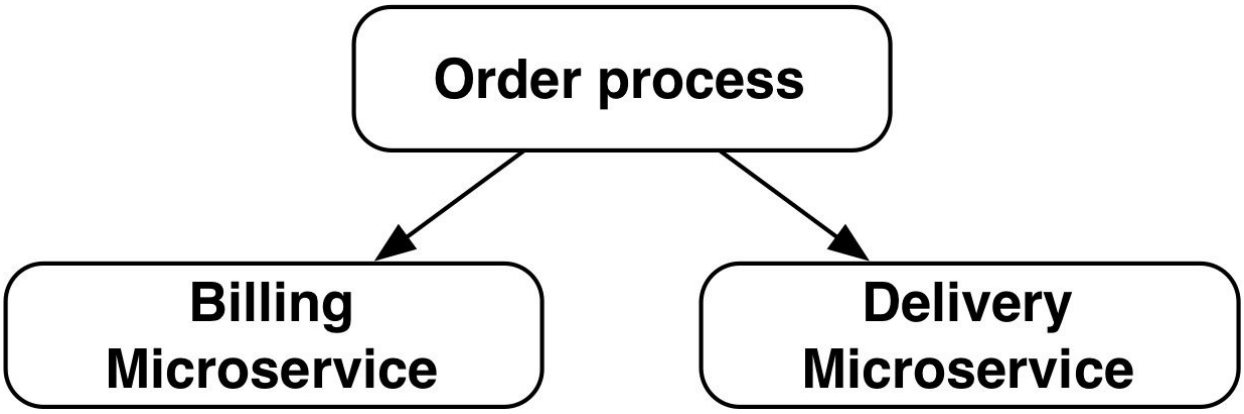


Fig. 29: Calls between Microservices

This requires that the order process knows the service for the billing and for the delivery. If a completed orders necessitates additional steps, the order service also has to call the services responsible for these steps.

Event-driven Architecture (EDA) enables a different modeling: When the order processing has been successfully finished, the order process will send an event. It is an event emitter. This event signals to all interested Microservices (event consumers) that there is a new successful order. Thus, one Microservice can now print a bill, and another Microservice can initiate a delivery.

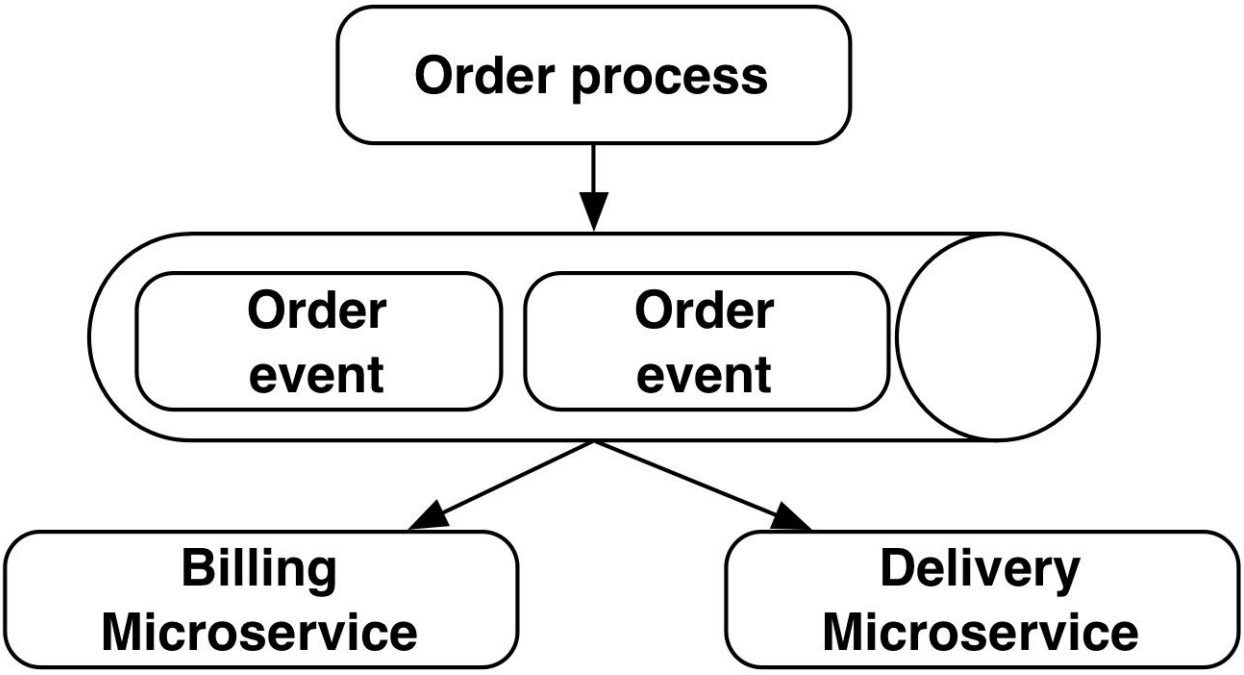


Fig. 30: Event-driven Architecture

This procedure has a number of advantages:

- When other Microservices are also interested in orders, they can easily register. Modifying the order process is not necessary anymore.
- Likewise, it is imaginable that also other Microservices trigger identical events – again without changes to the order process.
- The processing of events is temporally unlinked. It can happen later on.

At the architectural level Event-driven Architectures have the advantage that they allow for a very loose coupling and thus facilitate changes. The Microservices need to know only very little about each other. However, the coupling requires that logic is integrated and therefore implemented in different Microservices. Thereby a split into Microservice with UI and Microservices with logic can arise. That is not desirable. Changes to the business logic entail often changes to logic and UI. These are then separate Microservices. The change cannot readily take place in only one Microservice anymore and thus gets more complex.

Technically, such architectures can be implemented without a lot of effort via messaging (compare section 9.4). Microservices within such an architecture can very easily implement CQRS (section 10.2) or Event Sourcing (section 10.3).

8.7 Technical Architecture

To define a technology stack, with which the system can be built, is one of the main parts of an architecture. For individual Microservices this is likewise a very important task. However, the focus of this chapter is the Microservice-based system in its entirety. Of course, a certain technology can be bindingly defined for all Microservices. This has advantages: In that case the teams can exchange knowledge about the technology. Refactorings are simpler because members of one team can easily help out in other teams.

However, defining standard technologies is not mandatory: If they are not defined, there will be a plethora of different technologies and frameworks. However, since typically only one team is in contact with each technology, such an approach can be acceptable. Generally, Microservice-based architectures aim for the largest possible independence. In respect to the technology stack this independence translates into the ability to use different technology stacks and to independently make technology decisions. However, this freedom can also be restricted.

Technical Decisions for the Entire System

Nevertheless, at the level of the entire system there are some technical decisions to make. However, other aspects are more important for the technical architecture of the Microservice-based system than the technology stack for the implementation:

- As discussed in the last section, there might be technologies which can be used by all Microservices - for instances databases for data storage. Using these technologies does not necessarily have to be mandatory. However, especially in the case of persistence technologies, like for example databases, backups and disaster recovery concepts have to exist so that at least these technical solutions have to be obligatory. The same is true for other basic systems such as CMS for instance, which likewise have to be used by all Microservices.
- The Microservices have to adhere to certain standards in respect to monitoring, logging and deployment. Thereby, it can be ensured that the plethora of Microservices can still be operated in a uniform manner. Without such standards this is hardly possible anymore in case of a larger number of Microservices.
- Additional aspects relate to configuration (section 8.8), Service Discovery (section 8.9) and security (section 8.12).
- Resilience (section 10.5) and Load Balancing (section 8.10) are concepts which have to be implemented in a Microservice. Still the overall architecture can demand that each Microservice takes precautions in this area.
- An additional aspect is the communication of the Microservices with each other (compare chapter 9). For the system in its entirety a communication infrastructure has to be defined to which also the Microservices adhere.

The overall architecture does not necessarily restrict the choice of technologies. For logging, monitoring and deployment an interface could be defined. So there can be a standard according to which all Microservices log messages in the same manner and hand them over to a common log infrastructure. However, the Microservices do not necessarily have to use the same technologies for this. Similarly, it can be defined how data can be handed to the monitoring system and which data are relevant for the monitoring. A Microservice has to hand over the data to the monitoring, but a technology does not necessarily have to be prescribed. For deployment a completely automated Continuous Delivery pipeline can be demanded, which deploys software or deposits it into a repository in a certain manner. Which specific technology is used, is again a question for the

developers of the respective Microservice to decide. Practically, there are advantages when all Microservices employ the same technology. This reduces complexity, and there will also be more experience how to deal with the employed technology. However, in case of specific requirements, it is still possible to use a different technical solution when for this special case the advantages of such a solution predominate. This is an essential advantage of the technology freedom of Microservice-based architectures.

Sidecar

Even if certain technologies for implementing the demands on Microservices are rigidly defined, it will still be possible to integrate other technologies. Therefore, the concept of a Sidecar can be very useful. This is a process which integrates into the Microservices-based architecture via standard technologies and offers an interface which enables another process to use these features. This process can be implemented in an entirely different technology so that the technology freedom is preserved. FIg. 31 illustrates this concept: The Sidecar uses standard technologies and renders them accessible for another Microservice in an optional technology. The Sidecar is an independent process, and therefore can be called for instance via REST so that Microservices in arbitrary technologies can use the Sidecar. Section 14.12 shows a concrete example for a Sidecar.

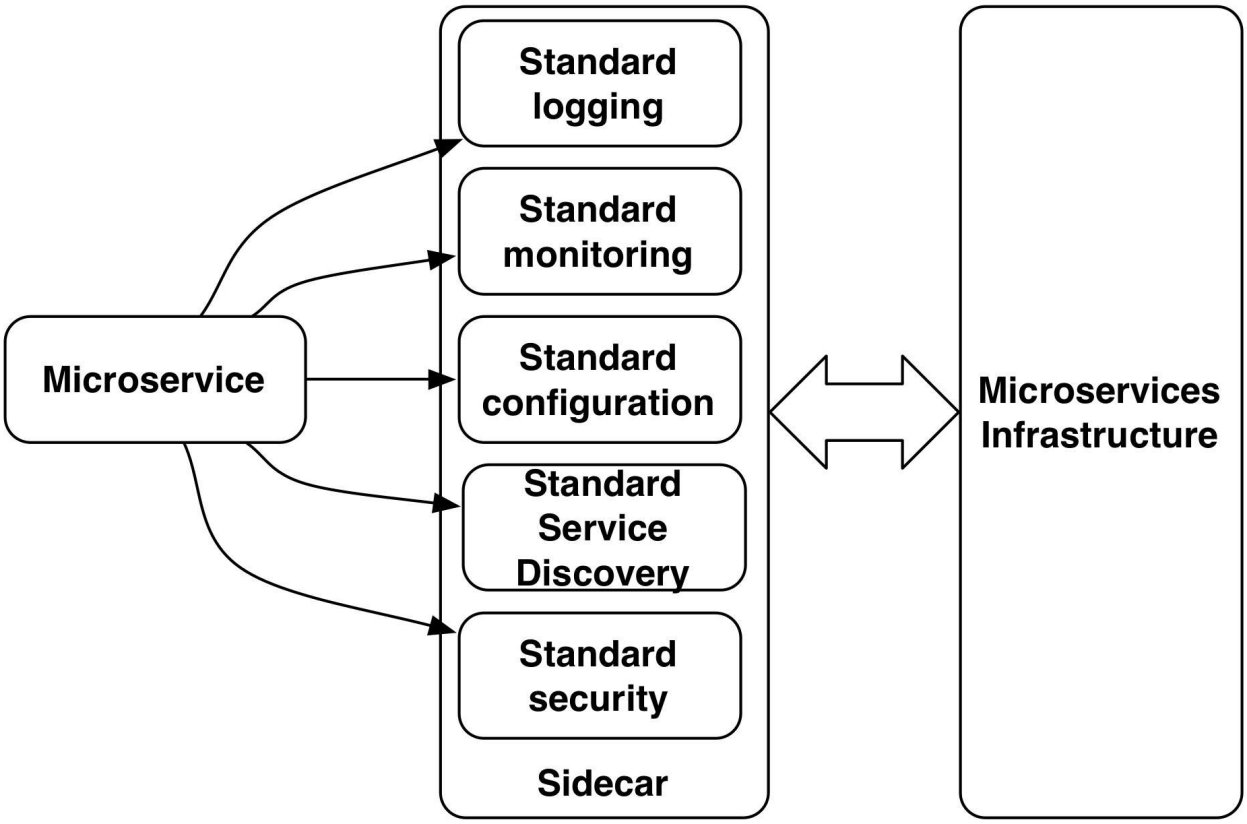


Fig. 31: A Sidecar renders all standard technologies accessible via a simple interface.

With this approach also such Microservices can be integrated into the architecture whose technological approach otherwise would exclude the use of the general technical basis for configuration, Service Discovery and security as the client component is not available for the entire technology.

In some regards the definition of the technology stack also affects other fields. The definition of technologies across all Microservices also affects the organization or can be the product of a certain organization (compare chapter 13).

Try and Experiment

A Microservices-based architecture is supposed to be defined.

- Which technical aspects could it comprise?
- Which aspects would you prescribe to the teams? Why?
- Which aspects should the teams decide on their own? Why?

In the end, the question is how much freedom one allows the teams to have. There are numerous possibilities – ranging from complete freedom up to the prescription of practically all aspects. However, some areas can only be centrally defined – the communication protocols for example. Section 13.3 discusses in more detail who should make which decisions in a Microservice-based project.

8.8 Configuration and Coordination

Configuring Microservice-based systems is laborious. They comprise a plethora of Microservices, which all have to be provided with the appropriate configuration parameters.

Some tools can store the configuration values and make them available to all Microservices. Ultimately, these are solutions in key/value stores, which save a certain value under a certain key:

- *Zookeeper* is a simple hierarchical system, which can be replicated onto multiple servers in a cluster. Updates arrive in an orderly fashion at the clients. This can also be used in a distributed environment, for instance for synchronization. Zookeeper has a consistent data model: All nodes have always the same data. The project is implemented in Java and is under Apache license.
- *etcd* originates from the Docker/CoreOS environment. It offers an HTTP interface with JSON as data format. etcd is implemented in Go and also under Apache license. Similar to Zookeeper, etcd also has a consistent data model and can be used for distributed coordination. For instance, etcd allows to implement a locking in a distributed system.
- *Spring Cloud Config* likewise has a REST-API. The configuration data can be provided by a Git backend. Thereby Spring Cloud Config directly supports data versioning. The data can also be encrypted to protect passwords. The system is well integrated into the Java framework Spring and can be used without additional effort in Spring systems since Spring itself provides already configuration mechanisms. Spring Cloud Config is

written in Java and is under Apache license. Spring Cloud Config does not offer support for synchronizing different distributed components.

Consistency as Problem

Some of the configuration solutions offer consistent data. This means that all nodes return the same data in case of a call. This is in a sense an advantage. However, according to the CAP theorem a node can only return an inconsistent response in case of a network failure – or none at all. In the end, without a network connection the node cannot know whether other nodes have already received other values. If the system allows only consistent responses, there can be no response at all in this situation. For certain scenarios this is highly sensible.

For instance, only one client should execute a certain code at a given time – for example in order to initiate a payment exactly once. The therefore necessary locking can be done by the configuration system: Within the configuration system there is a variable, which upon entering this code has to be set. Only in that case the code may be executed. In the end, it is better when the configuration system does not return a response so that not by chance two clients execute the code in parallel.

However, for configurations such strict requirements regarding consistency are often not necessary. Maybe it is better when a system gets an old value rather than that is does not get any value at all. However, in the case of CAP different compromises are possible. etcd for instance returns under certain conditions rather an incorrect response than no response at all.

Immutable Server

Another problem associated with the centralized storage of configuration data is that the Microservices do not only depend on the state of their own file system and the contained files, but also on the state of the configuration server. Therefore, a Microservice now cannot be exactly replicated anymore – for this also the state of the configuration server is relevant. This makes the reproduction of errors and the search for errors in general more difficult.

In addition, the configuration server is in opposition to the concept of Immutable Server. In this approach every software change leads to a new installation of the software. Ultimately, the old server is terminated upon an update, and a new server with an entirely new installation of the software is started. However, in case of an external configuration server a part of the configuration will not be

present on the server, and therefore the server is after all changeable in the end by adjusting the configuration. However, exactly this is not supposed to happen. To prevent it, a configuration can be made in the server itself instead of the configuration server. In that case configuration changes can only be implemented by rolling out a new server.

Alternative: Installation tools

The installation tools (discussed in section 12.4) represent a completely different approach for the configuration of individual Microservices. These tools support not only the installation of software, but also the configuration. For the configuration configuration files can for instance be generated, which can subsequently be read by Microservices. The Microservice itself does not notice the central configuration since it reads only a configuration file. Still, these approaches support all scenarios, which typically occur in a Microservices-based architecture. Thus, this approach allows a central configuration and is not in opposition to Immutable Server as the configuration is completely transferred to the server.

8.9 Service Discovery

Service Discovery ensures that Microservices can find each other. This is in a sense a very simple task: For instance, a configuration file detailing the IP address and the port of the Microservice can be delivered on all computers. Typical configuration management systems enable the rollout of such files. However, this approach is not sufficient:

- Microservices can come and go. This does not only happen due to server failures, but also because of new deployments or the scaling of the environment by the start of new servers. Service Discovery has to be dynamic. A fixed configuration is not sufficient.
- Due to Service Discovery the calling Microservices are not so closely coupled anymore to the called Microservice. This has positive effects for scaling: A client is not bound to a concrete server instance anymore, but can contact different instances – depending on the current load of the different servers.
- When all Microservices have a common approach for Service Discovery, a central registry of all Microservices arises. This can be helpful for an architecture overview (compare section 8.2). Or monitoring information can be retrieved by all systems.

In systems, which employ messaging, Service Discovery can be dispensable. Messaging systems already decouple sender and recipient. Both know only the shared channel via which they communicate. However, they do not know the identity of their communication partner. The flexibility, which Service Discovery offers, is then provided by the decoupling via the channels.

Service Discovery = Configuration?

In principle it is conceivable to implement Service Discovery by configuration solutions (compare section 8.8). In the end, only the information which service is reachable at which location is supposed to be transferred. However, configuration mechanisms are in effect the wrong tools for this. For a Service Discovery a high availability is more important than for a configuration server. In the worst case a failure of Service Discovery can have the consequence that communication between Microservices gets impossible. Consequently, the trade-off between consistency and availability is different compared to configuration systems. Therefore, configuration systems should only be used for Service Discovery when they offer an appropriate availability. This can have consequences for the necessary architecture of the Service Discovery system.

Technologies

There are many different technologies for Service Discovery:

- One example is *DNS* (Domain Name System). This protocol ensures that a host name like *www.ewolff.com* can be resolved to an IP address. DNS is an essential component of the Internet and has clearly proven its scalability and availability. DNS is hierarchically organized: There is a DNS server which administrates the *.com* domain. This DNS-Server knows which DNS server administrates the subdomain *ewolff.com*, and the DNS server of this subdomain finally knows the IP address of *www.ewolff.com*. In this way a namespace can be hierarchically organized, and different organizations can administrate different parts of the namespace. If a server named *server.ewolff.com* is supposed to be created, this can be easily done by a change in the DNS server of the domain *ewolff.com*. This independence fits well to the concept of Microservices, which especially focus on independence in regards to their architecture. To ensure reliability there are always several servers, which administrate a domain. In order to reach scalability DNS supports caching so that calls do not have to implement the entire resolution of a name via multiple DNS servers, but can be served by a cache. This does not only promote performance, but also reliability.

For Service Discovery it is not sufficient to resolve the name of a server into an IP address. In addition, there has to be a network port for each service. Therefore, the DNS has SRV records. These contain the information on which computer and port the service is reachable. In addition, a priority and a weight can be set for a certain server. These values can be used to select one of the servers and thereby to prefer powerful servers. Via this approach, DNS offers reliability and Load Balancing onto multiple servers. Advantages of DNS are apart from scalability also the availability of many different implementations and the broad support in different programming languages.

- A frequently used implementation for a DNS server is *BIND*. BIND runs on different operating systems (Linux, BSD, Windows, Mac OS X), is written in the programming language C and is under an open source license.
- *Eureka* is part of the Netflix stack. It is written in Java and is under Apache license. The example application in this book uses Eureka for Service Discovery (compare section 14.8). For every service Eureka stores under the service name a host and a port, under which the service is available. Eureka can replicate the information about the services onto multiple Eureka servers in order to increase the availability. Eureka is a REST service. A Java library for the clients belongs to Eureka. Via the Sidecar concept (section 8.7) this library can also be used by systems, which are not written in Java. The Sidecar takes over the communication with the Eureka server, which then offers Service Discovery to the Microservice. On the clients the information from the server can be held in a cache so that calls are possible without communication with the server. The server regularly contacts the registered services to determine which services failed. Eureka can be used as basis for Load Balancing since several instances can be registered for one service. The load can then be distributed onto these instances. Eureka was originally designed for the Amazon Cloud.
- *Consul* is a key/value store and fits therefore also into the area of configuration servers (section 8.8). Apart from consistency it can also optimize in regards to availability. Clients can register with the server and react to certain events. In addition to a DNS interface it also has a HTTP/JSON interface. It can check whether services are still available by executing health checks. Consul is written in Go and is under the Mozilla open source license. Besides, Consul can create configuration files from templates. Thereby a system expecting services in a configuration file can likewise be configured by Consul.

Every Microservice-based architecture should use a Service Discovery system. It forms the basis for the administration of a large number of Microservices and for additional features like Load Balancing. If there is only a small number of Microservices, it is still imaginable to get along without Service Discovery. However, for a large system Service Discovery is indispensable. Since the number of Microservices increases over time, Service Discovery should be integrated into the architecture right from the start. Besides, practically each system uses at least the name resolution of hosts, which is already a simple Service Discovery.

8.10 Load Balancing

It is one of the advantages of Microservices that each individual service can be independently scaled. To distribute the load between the instances, multiple instances, which share the load, can simply be registered in a messaging solution (compare 9.4). The actual distribution of the individual messages is then performed by the messaging solution. Messages can either be distributed to one of the receivers (Point-to-Point) or to all receivers (Publish/Subscribe).

REST/HTTP

In case of REST and HTTP a load balancer has to be used. The load balancer has the function to behave to the outside like a single instance, but to distribute requests to multiple instances. Besides, a load balancer can be useful during deployment: Instances of the new version of the Microservice can initially start without getting load. Afterwards the load balancer can be reconfigured in a way that the new Microservices are put into operation. In doing so the load can also be increased in a stepwise manner. This decreases the risk of a system failure.

Fig. 32 illustrates the principle of a proxy-based load balancer: The client sends its requests to a load balancer running on another server. This load balancer is responsible for sending each request to one of the known instances. There the request is processed.

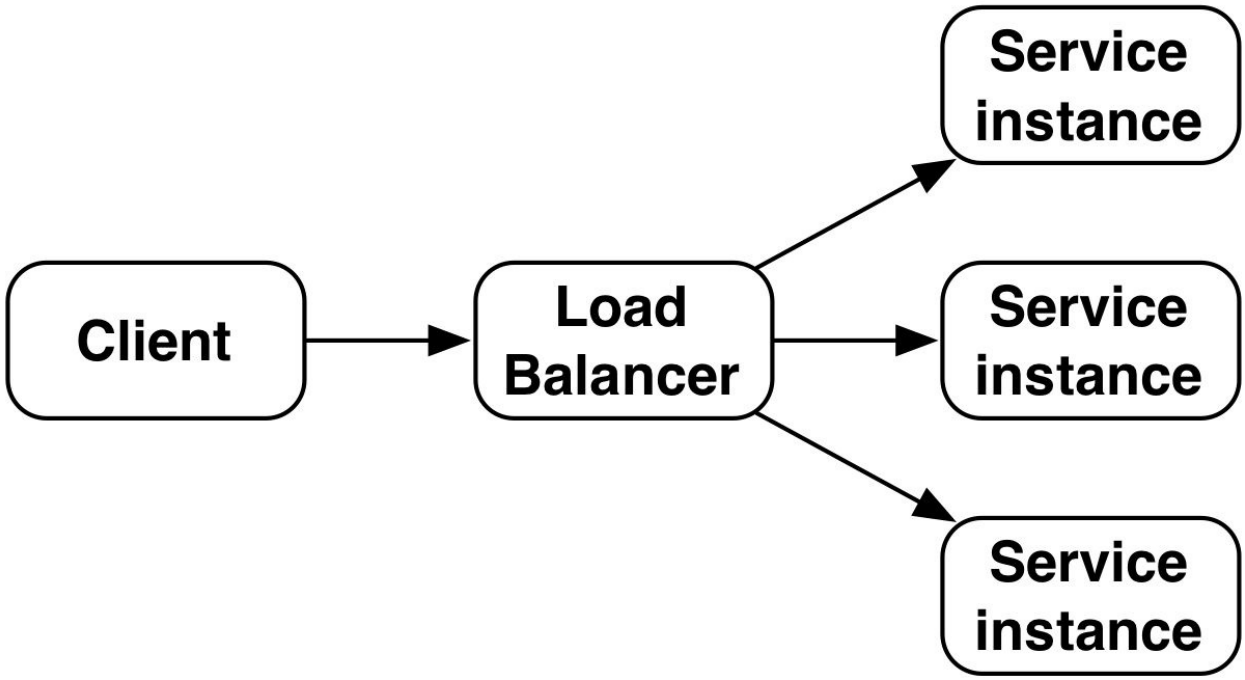


Fig. 32: Proxy-based Load Balancer

This approach is common for websites and relatively easy to implement. The load balancer retrieves information from the service instances to determine the load of the different instances. In addition, the load balancer can remove a server from the Load Balancing when the node does not react to requests anymore.

On the other hand, this approach has the disadvantage that the entire traffic for one kind of service has to be directed via a load balancer. Thereby the load balancer can turn into a bottleneck. Besides, a failure of the load balancer results in the failure of a Microservice.

Central Load Balancer

A central load balancer for all Microservices is not only not to be recommended for these reasons but also because of the configuration. The configuration of the load balancer gets very complex when only one load balancer is responsible for many Microservices. Besides, the configuration has to be coordinated between all Microservices. Especially when deploying a new version of a Microservice a modification of the load balancer can be sensible in order to put the new Microservice only after a comprehensive test under load. The need for coordination between Microservices should especially be avoided in regards to deployment to ensure the independent deployment of Microservices. In case of such a reconfiguration one has to make sure that the load balancer supports a dynamic reconfiguration and for instance does not lose information regarding

sessions if the Microservice uses sessions. Also for this reason it cannot be recommended to implement stateful Microservices.

A Load Balancer pro Microservice

There should be one load balancer per Microservice, which distributes the load between the instances of the Microservice. This allows the individual Microservices to independently distribute load, and different configurations per Microservice are possible. Likewise, it is simple to appropriately reconfigure the load balancer upon the deployment of a new version. However, in case of a failure of the load balancers the Microservice will not be available anymore.

Technologies

For Load Balancing there are different approaches:

- The Apache httpd web server supports Load Balancing with the extension mod_proxy_balancer.
- The web server nginx can likewise be configured in a way that it supports Load Balancing. To use a web server as load balancer has the advantage that it can also deliver static websites, CSS and images. Besides, the number of technologies will be reduced.
- HAProxy is a solution for Load Balancing and high availability. It does not support HTTP, but all TCP-based protocols.
- Cloud providers frequently also offer load balancer. Amazon for instance offers Elastic Load Balancing. This can be combined with Auto Scaling so that higher loads automatically trigger the start of new instances, and thereby the application automatically scales with load.

Service Discovery

Another possibility for Load Balancing is Service Discovery (Fig. 33) (compare section 8.9). When the Service Discovery returns different nodes for a service, the load can be distributed across several nodes. However, this approach allows redirecting to another node only in the case that a new Service Discovery is performed. This makes it difficult to achieve a fine granular Load Balancing. For a new node it will therefore take some time until it gets a sufficient share of load. Finally, the failure of a node is hard to correct because a new Service Discovery would be necessary for that. It is useful that in case of DNS it can be stated for a set of data how long the data is valid (time-to-live). Afterwards the Service Discovery has to be run again. This allows a simple Load Balancing via DNS

solutions and also with Consul. However, unfortunately this time-to-live is often not completely correctly implemented.

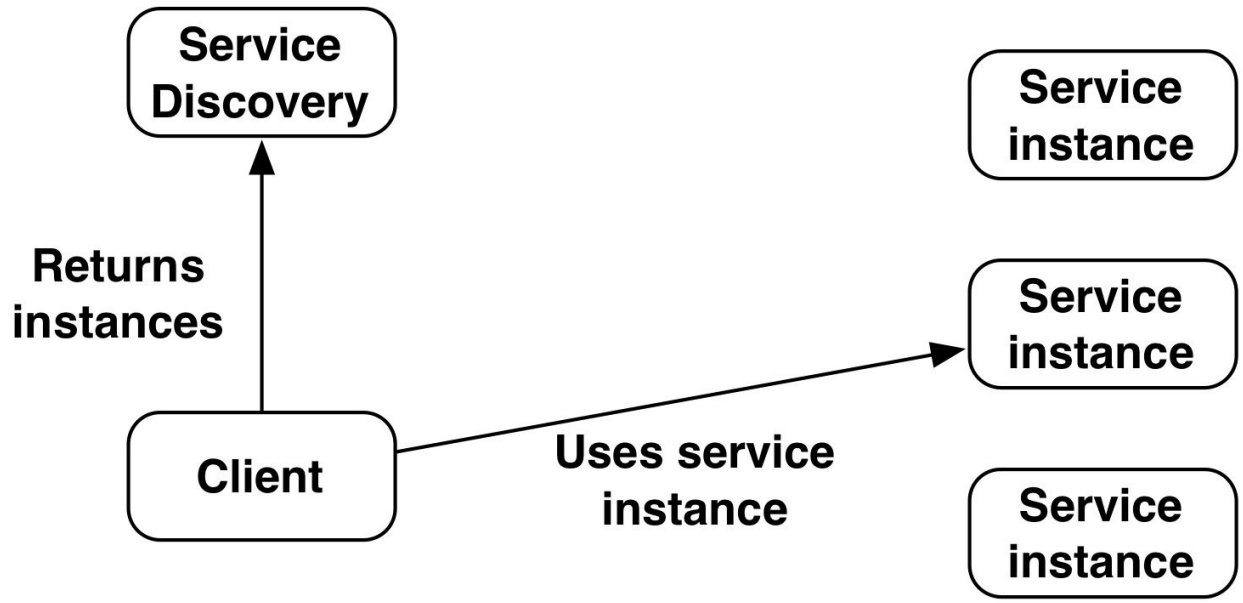


Fig. 33: Load Balancing with Service Discovery

Load Balancing with Service Discovery is simple because Service Discovery anyhow has to be present in a Microservice-based system. Therefore, the Load Balancing does not introduce additional software components. Besides avoiding a central load balancer has the positive effect that there is no bottle neck and no central component whose failure would have tremendous consequences.

Client-based Load Balancing

The client itself can also use a load balancer. The load balancer can be implemented as a part of the code of the Microservice or it can come as a proxy-based load balancer such as nginx or Apache httpd, which runs on the same computer as the Microservice. In that case there is no bottle neck because each client has its own load balancer, and the failure of an individual load balancer has hardly consequences. However, configuration changes have to be passed on to all load balancers, which can cause quite a lot of network traffic and load.

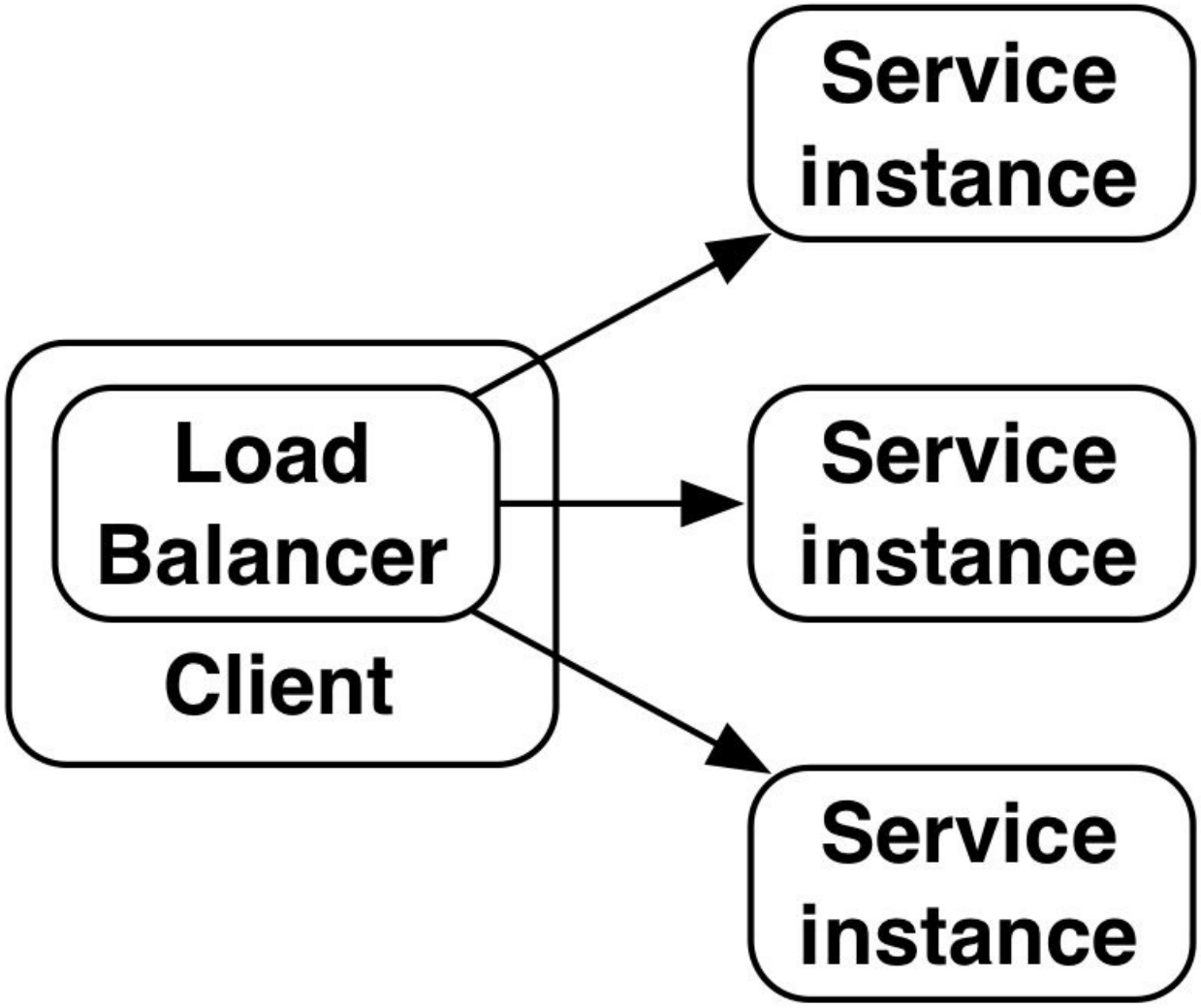


Fig. 34: Client-based Load Balancing

Ribbon is an implementation of client-based Load Balancing. It is a library which is written in Java and can use Eureka to find service instances. Alternatively, a list of servers can be handed over to Ribbon. Ribbon implements different algorithms for Load Balancing. Especially when using it in combination with Eureka, the individual load balancer does not need to be configured anymore. Because of the Sidecar concept Ribbon can also be used by Microservices which are not implemented in Java. The example system uses Ribbon (compare section 14.11).

Consul offers the possibility to define a template for configuration files of load balancers. This allows to feed the load balancer configuration with data from Service Discovery. A client-based Load Balancing can be implemented by defining a template for each client, into which Consul writes all service instances. This process can be regularly repeated. In this manner a central system

configuration is again possible and a client-based Load Balancing relatively simple to implement.

Load Balancing and Architecture

It is hardly sensible to use more than one kind of Load Balancing within a single Microservice-based system. Therefore, this decision should be made once for the entire system. Load Balancing and Service Discovery have a number of contact points. Service Discovery knows all service instances; Load Balancing distributes the loads between the instances. Both technologies have to work together. Thus the technology decisions in this area will influence each other.

8.11 Scalability

To be able to cope with high loads, Microservices have to scale. Scalability means that a system can process more load when it gets more resources.

There are two different kinds of scalability:

Horizontal scalability
: means that more resources are used, which each process part of the load, i.e. the number of resources increases.

Vertical scalability
: means that more powerful resources are employed to handle a higher load. Here, an individual resource will process more load, while the number of resources stays constant.

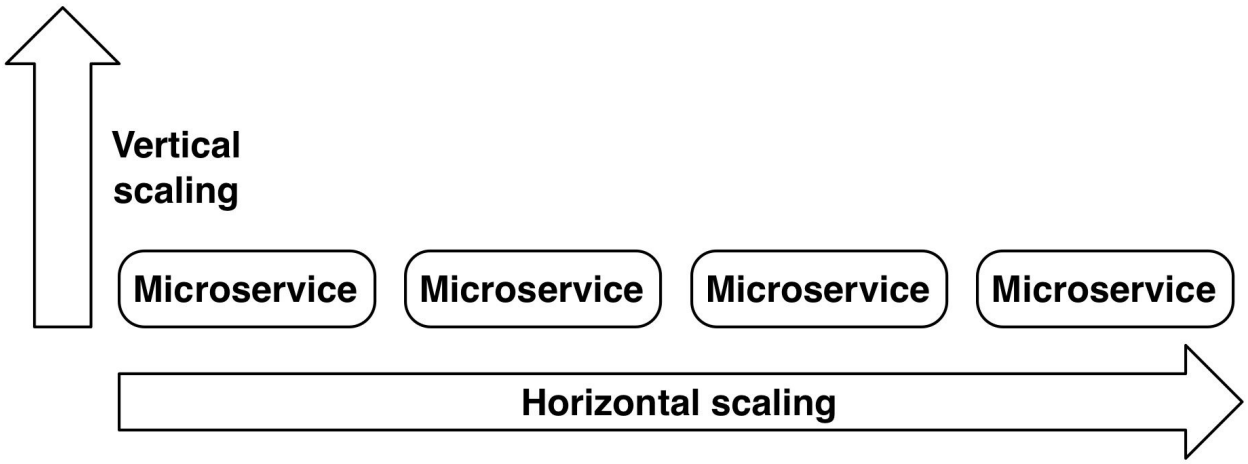


Fig. 35: Horizontal and vertical scaling

Horizontal scalability is often the better choice since the limit for the possible number of resources and therefore the limit for the scalability is very high. Besides, it is cheaper to buy more resources than more powerful ones. One fast computer is often more expensive than many slow ones.

Scaling, Microservices and Load Balancing

Microservices employ mostly horizontal scaling where the load is distributed across several Microservice instances via Load Balancing. The Microservices themselves have to be stateless for this. More precisely: They should not have any state, which is specific for an individual user, because then the load can only be distributed to nodes, which have the respective state. The state for a user can be stored in a database or alternatively be put into an external storage (e.g. In-Memory-Store), which can be accessed by all Microservices.

Dynamic Scaling

Scalability means only that the load can be distributed to multiple nodes. How the system really reacts to the load, is not defined. In the end it is more important that the system really adapts to an increasing load. For that it is necessary that, depending on the load, a Microservice starts new instances, onto which the load can be distributed. This allows the Microservice to also cope with high loads. This process has to be automated as manual processes would be too laborious.

There are different places in the Continuous Deployment pipeline (chapter 12) where it is necessary to start a Microservice to test the services. For that a suitable deployment system such as Chef or Puppet can be used. Alternatively, a new virtual machine or a new Docker container with the Microservice is simply started. This mechanism can also be used for dynamic scaling. It only has additionally to register the new instances with the Load Balancing. However, the instance should be able to handle the production load right from the start: Therefore, the caches should for instance already be filled with data.

Dynamic scaling is especially simple with Service Discovery: The Microservice has to register with the Service Discovery. The Service Discovery can configure the load balancer in a way that it distributes load to the new instance.

The dynamic scaling has to be performed based on a metric. When the response time of a Microservice is too long or the number of requests is very high, new instances have to be started. The dynamic scaling can be part of a monitoring (compare section 12.3) since the monitoring should enable the reaction to

extraordinary metric values. Most monitoring infrastructures offer the possibility to react to metric values by calling a script. The script can start additional instances of the Microservice. This is fairly easy to do with most cloud and virtualization environments. Environments like the Amazon Cloud offer suitable solutions for automatic scaling, which work in a similar manner. However, a home-grown solution is not very complicated since the scripts run anyhow only every few minutes so that failures are tolerable, at least for a limited time. Since the scripts are part of the monitoring, they will have a similar availability like the monitoring and should therefore be sufficiently available.

Especially in the case of cloud infrastructures it is important to shut the instances down again in case of low load because every running instance costs money in a cloud. Also here scripts can serve as reaction to certain metric values.

Microservices: Advantages for Scaling

In regards to scaling, Microservices have first of all the advantage that they can be scaled independently of each other. In case of a Deployment Monolith only the entire monolith can be started as more instances. The fine granular scaling does not appear to be an especially striking advantage at first glance, however, to run an entire E-commerce shop in many instances just to speed up the search, causes high expenditures: A lot of hardware is needed, a complex infrastructure has to be built up, and system parts are held available, which are not used at all. These system parts render the deployment and monitoring more complex. The possibilities for dynamic scaling depend critically on the size of the services and on the speed with which new instances can be started. In this area Microservices possess clear advantages.

In most cases Microservices have already an automated deployment, which is also very easy to implement. In addition, there is already a monitoring. Without automated deployment and monitoring a Microservice-based system can hardly be operated. If there is in addition load balancing, then it is only a script which is still missing for automated scaling. Therefore Microservices represent an excellent basis for dynamic scaling.

Sharding

Sharding means that the administrated data amount is divided and that each instance gets the responsibility for part of the data. For example, an instance can be responsible for the customers A-E or for all customers whose customer number ends with the number 9. Sharding is a variation of horizontal scaling: More

servers are used. However, not all servers are equal, but every server is responsible for a different subset of the dataset. In case of Microservices this type of scaling is easy to implement since the domain is anyhow distributed across multiple Microservices. Every Microservice can then shard its data and via this sharding scale horizontally. A Deployment Monolith is hardly scalable in this manner because it handles all the data. When the Deployment Monolith administrates customers and items, it can hardly be sharded for both types of data. In order to really implement sharding the Load Balancer has of course to distribute the load appropriately to the shards.

Scalability, Throughput and Response Times

Scalability means that more load can be processed by more resources. The throughput increases – i.e. the number of processed requests per unit of time. However, the response time stays constant in the best case – depending on circumstances it might rise, but not to such an extent that the system causes errors or gets too slow for the user.

When faster response times are required, horizontal scaling does not help. However, there are some approaches to optimize the response time of Microservices:

- The Microservices can be deployed on faster computers. This is vertical scaling. Then the Microservices can process the individual requests more rapidly. Because of the automated deployment vertical scaling is relatively simple to implement. The service has only to be deployed on faster hardware.
- Calls via the network have a long latency. Therefore, a possible optimization can be to forego such calls. Instead caches can be used, or the data can be replicated. Caches can often very easily be integrated into the existing communication. For REST, for instance, a simple HTTP cache is sufficient.
- If the domain architecture of Microservices is well designed, a request should only be processed in one Microservice so that no communication via the network is necessary. In case of a good domain architecture the logic for processing a request is implemented in one Microservice so that changes to the logic only require changes to one Microservice. In that case Microservices do not have longer response times than Deployment Monoliths. In regards to an optimization of response times Microservices have the disadvantage that their communication via the network causes rather longer response times. However, there are means to counteract this effect.

8.12 Security

In a Microservice-based architecture each Microservice has to know which user triggered the current call and wants to use the system. Therefore, a uniform security architecture has to exist: After all, Microservices can work together for a request, and for each part of the processing of the request another Microservice might be responsible. Thus the security structure has to be defined at the level of the entire system. This is the only way to ensure that the access of a user is uniformly treated in the entire system in regards to security.

Security comprises two essential aspects: Authentication and authorization. Authentication is the process, which validates the identity of the user. Authorization denotes the decision whether a certain user is allowed to execute a certain action. Both processes are independent of each other: The validation of the user identity in the context of authentication is not directly related to authorization.

Security and Microservices

In a Microservice-based architecture the individual Microservices should not perform authentication. It does not make much sense for each Microservice to validate user name and password. For authentication a central server has to be used. For authorization an interplay is necessary: Often there are user groups or roles which have to be centrally administered. However, whether a certain user group or role is allowed to use certain features of a Microservice should be decided by the concerned Microservice. Thereby changes to the authorization of a certain Microservice can be limited to the implementation of this Microservice.

OAuth2

One possible solution for this challenge is OAuth2. This protocol is also widely used in the internet. Google, Microsoft, Twitter, XING or Yahoo all offer support for this protocol.

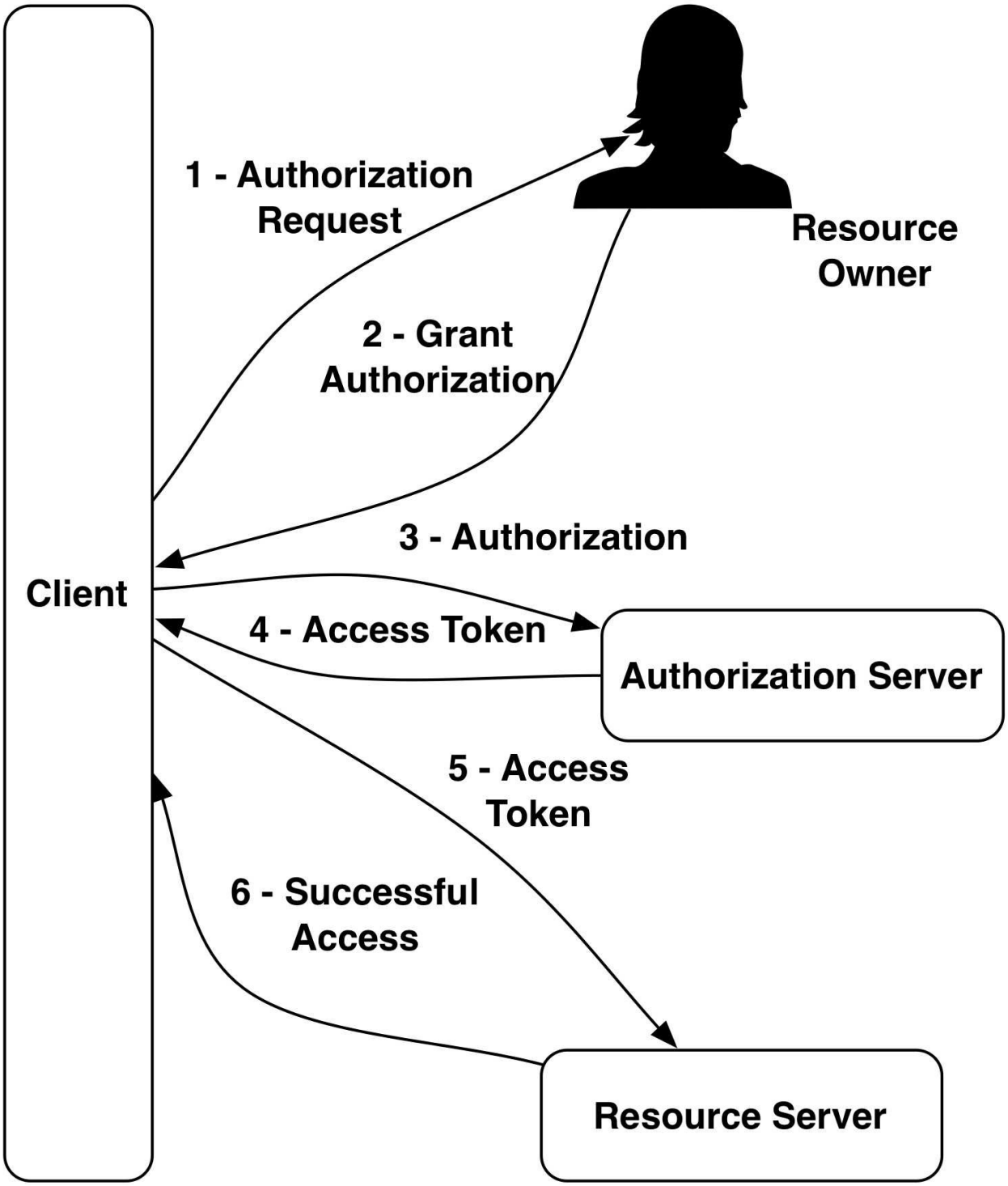


Fig. 36: The OAuth2 protocol

Fig. 36 shows the workflow of the OAuth2 protocol as defined by the standard:

1. The client inquires of the Resource Owner whether it might execute a certain action. For example, the application can request access to the profile or

certain data in a social network which the Resource Owner stored there. The Resource Owner is usually the user of the system.
2. If the Resource Owner grants the client access, the client receives a respective response from the Resource Owner.
3. The client uses the response of the Resource Owner to put a request to the authorization server. In the example the authorization server would be located in the social network.
4. The authorization server returns an access token.
5. With this access token the client can now call a Resource Server and there obtain the necessary information. For the call the token can for instance be put into an HTTP header.
6. The Resource Server answers the requests.

Possible Authorization Grants

The interaction with the authorization server can work in different ways:

- In case of the *Password Grant* the client shows an HTML form to the user in step 1. The Resource Owner can enter user name and password. In step 3 this information is used by the client to obtain the access token from the authorization server via an HTTP POST. This approach has the disadvantage that the client processes user name and password. The client can be insecurely implemented, and then these data are endangered.
- In case of the *Authorization Grant* the client directs the user in step 1 to a web page, which the authorization server displays. There the user can choose whether he/she permits the access. If that is the case, the client will obtain in step 2 an authorization code via an HTTP-URL. In this way the authorization server can be sure that the correct client obtains the code since the server chooses the URL. In step 3 the client can then generate the access token with this authorization code via an HTTP POST. The approach is mainly implemented by the authorization server and thus very easy to use by a client. In this scenario the client would be a web application on the server: It will obtain the code from the authorization server and is the only one able to turn it via the HTTP POST into an access token.
- In case of *Implicit* the procedure resembles the Authorization Grant. After the redirect to the authorization server in step 1 the client directly gets an access token via an HTTP redirect. This allows the browser or a mobile application to immediately readout the access token. Step 3 and 4 are omitted. However, here the access token is not as well protected against attacks since the authorization server does not directly send it to the client.

This approach is sensible when JavaScript code on the client or a mobile application is supposed to use the access token.
- In case of *Client Credentials* the client uses in step 1 a credential, which the client knows, to obtain the access token from the authorization server. Thereby the client can access the data without additional information from the Resource Owner. For example, a statistics software could readout and analyze customer data in this manner.

Via the access token the client can access resources. The access token has to be protected: When unauthorized people obtain access to the access token, they can thereby trigger all actions, which the Resource Owner can also trigger. Within the token itself some information can be encoded. For instance, in addition to the real name of the Resource Owner the token can also contain information, which assigns certain rights to the user or the membership to certain user groups.

JSON Web Token (JWT)

JSON Web Token (JWT) is a standard for the information, which is contained in an access token. JSON serves as data structure. For the validation of the access token a digital signature with JWS (JSON Web Signature) can be used. Likewise the access token can be encrypted with JSON Web Encryption (JWE). The access token can contain information about the issuer of the access token, the Resource Owner, the validity interval or the addressee of the access token. Individual data can also be contained in the access token. The access token is optimized for use as HTTP header by an encoding of the JSON with BASE64. These headers are normally subject to size restrictions.

OAuth2, JWT and Microservices

In a Microservice-based architecture the user can initially authenticate via one of the OAuth2 approaches. Afterwards the user can use the web page of a Microservice or call a Microservice via REST. With each further call every Microservice can hand over the access token to other Microservices. Based on the access token the Microservices can decide whether a certain access is granted or not. For that the validity of the token can first be checked. In case of JWT the token only has to be decrypted and the signature of the authorization server has to be checked. Subsequently, it can be decided based on the information of the token whether the user may use the Microservice as he/she intends. Information from the token can be used for that. For instance, it is possible to store the affiliation with certain user groups directly in the token.

It is important that it is not defined in the access token, which access to which Microservice is allowed. The access token is issued by the authorization server. If the information about the access was available in the authorization server, every modification of the access rights would have to occur in the authorization server – and not in the Microservices. This limits the changeability of the Microservices since modifications to the access rights would require changes of the authorization server as central component. The authorization server should only administer the assignment to user groups and the Microservices should then allow or prohibit access based on such information from the token.

Technologies

In principle, other technical approaches than OAuth2 could also be used as long as they employ a central server for authorization and use a token for regulating the access to individual Microservices. One example is Kerberos, which has a relatively long history. However, it is not as well tuned to REST like OAuth2. Other alternatives are SAML and SAML 2.0. They define a protocol, which uses XML and HTTP to perform authorization and authentication.

Finally, signed cookies can be created by a home-grown security service. Via a cryptographic signature it can be determined whether the cookie has really been issued by the system. The cookie can then contain the rights or groups of the user. Microservices can examine the cookie and restrict the access if necessary. There is the risk that the cookie is stolen. However, for that to occur the browser has to be compromised or the cookie has to be transferred via a non encrypted connection. This is often acceptable as risk.

With a token approach it is possible that Microservices do not have to handle the authorization of the caller, but still can restrict the access to certain user groups or roles.

There are good reasons for the use of OAuth2:

- There are numerous libraries for practically all established programming languages, which implement OAuth2 or an OAuth2 server. The decision for OAuth2 hardly restricts the technology choice for Microservices.
- Between the Microservices only the access token still has to be transferred. This can occur in a standardized manner via an HTTP header when REST is used. In case of different communication protocols similar mechanisms can be exploited. Also in this area OAuth2 hardly limits the technology choice.

- Via JWT information can be placed into the token, which the authorization server communicates to the Microservices in order for them to allow or prohibit access. Therefore, also in this area the interplay between the individual Microservice and the shared infrastructure is simple to implement – with standards, which are widely supported.

Spring Cloud Security offers especially for Java-based Microservices a good basis for implementing OAuth2 systems.

Additional Security Measures

OAuth2 solves first of all the problem of authentication and authorization – primarily for human users. There are additional measures for securing a Microservice-based system:

- The communication between the Microservices can be protected by SSL/TLS against wiretapping. All communication is then encrypted. Infrastructures like REST or messaging systems mostly support such protocols.
- Apart from authentication with OAuth2 certificates can be used to authenticate clients. A certificate authority creates the certificates. They can be used to verify digital signatures. This makes it possible to authenticate a client based on its digital signature. Since SSL/TLS supports certificates, at least at this level the use of certificates and authentication via certificates is possible.
- API keys represent a similar concept. They are given to external clients to enable them to use the system. Via the API key the external clients authenticate themselves and can obtain the appropriate rights. In case of OAuth2 this can be implemented with Client Credential.
- Firewalls can be used to protect the communication between Microservices. Normally firewalls secure a system against unauthorized access from outside. A firewall for the communication between the Microservices prevents that all Microservices are endangered if an individual Microservice has been successfully taken over. In this way the intrusion can be restricted to one Microservice.
- Finally, there should be an intrusion detection to detect unauthorized access to the system. This topic is closely related to monitoring. The monitoring system can also be used to trigger an appropriate alarm in case of an intrusion.
- Datensparsamkeit is also an interesting concept. It is derived from the data security field and states that only those data are to be saved, which are

absolutely necessary. Form a security perspective this results in the advantage that collecting lots of data is avoided. This makes the system less attractive for attacks, and in addition the consequences of a security breach will not be as bad.

Hashicorp Vault

Hashicorp Vault is a tool, which solves many problems in the area of Microservice security. It offers the following features:

- Secrets like passwords, API Keys, keys for encryption or certificates can be saved. This can be useful to allow users to administrate their secrets. In addition also Microservices can be equipped with certificates in such a manner as to protect their communication with each other or with external servers.
- Secrets are given via a lease to services. Besides, they can be equipped with an access control. This helps to limit the problem in case of a compromised service. Secrets can for instance also be declared invalid.
- Data can be immediately encrypted or decrypted with the keys without the Microservices themselves having to save these keys.
- Access is made traceable by an audit. This allows to trace who got which secret and at which time.
- In the background Vault can use HSMs, SQL databases or Amazon IAM to store secrets. In addition, it can for instance also generate new access keys for the Amazon Cloud by itself.

In this manner Vault takes care of handling keys and thereby relieves Microservices of this task. It is a big challenge to really handle keys securely. It is difficult to implement something like that in a really secure manner.

Additional Security Goals

In regards to a software architecture security comes in very different shapes. Approaches like OAuth2 only help to achieve confidentiality. They prevent that data is accessible for unauthorized users. However, even this confidentiality is not entirely safeguarded by OAuth2 on its own: The communication in the network likewise has to be protected against wiretapping – for instance via HTTPS or other kinds of encryption.

Additional security aspects are:

Integrity
: means that there are no unnoticed changes to the data. Every Microservice has to solve this problem. For instance, data can be signed to ensure that they have not been manipulated in some way. The concrete implementation has to be performed by each Microservice.

Confidentiality
: ensures that modifications cannot be denied. This can be achieved by signing the changes introduced by different users by keys that are specific for the individual user. Then it is clear that exactly one specific user has modified the data. The overall security architecture has to provide the keys; the signing is then the task of each individual service.

Data security
: is ensured as long as no data are lost. This issue can be handled by backup solutions and highly available storage solutions. This problem has to be addressed by the Microservices since it is within their responsibility as part of their data storage. However, the shared infrastructure can offer certain databases, which are equipped with appropriate backup and disaster recovery mechanisms.

Availability
: means that a system is available. Also here the Microservices have to contribute individually. However, since especially in the case of Microservice-based architectures one has to deal with the possibility of failures of individual Microservices, Microservice-based systems are often well prepared in this area. Resilience (section 10.5) is for instance useful for this.

These aspects are often not considered when devising security measures – however, the failure of a service has often even more dramatic consequences than the unauthorized access to data. One danger is Denial of Service attacks, which result in such an overloading of servers that they cannot perform any sensible work anymore. The technical hurdles for this are often shockingly low, and the defense against such attacks is frequently very difficult.

8.13 Documentation and Metadata

To keep the overview in a Microservice-based architecture certain information about each Microservice has to be available. Therefore, the Microservice-based architecture has to define how Microservices can provide such information. Only

when all Microservices provide this information in a uniform way, the information can be easily collected. Possible information of interest is for instance:

- Fundamental information like the name of the service and the responsible contact person.
- Information about the source code: Where the code can be found in the version control and which libraries have been used. The used libraries can be interesting in order to compare open source licenses of the libraries with the company policies or to identify in case of a security gap in a library the affected Microservices. For such purposes the information has to be available even if the decision about the use of a certain library rather concerns only one Microservice. The decision itself can be made largely independently by the responsible team.
- Another interesting information is with which other Microservices the Microservice works together. This information is central for the architecture management (compare section 8.2).
- In addition, information about configuration parameters or about feature toggles might be interesting. Feature toggles can switch features on or off. This is useful for activating new features only in production when their implementation is really finished, or for avoiding the failure of a service by deactivating certain features.

It is not sensible to document all components of the Microservices or to unify the entire documentation. A unification only makes sense for information, which is relevant outside of the team implementing the Microservice. Whenever it is necessary to manage the interplay of Microservices or to check licenses, the relevant information has to be available outside of the responsible team. These questions have to be solved across Microservices. Each team can create additional documentation about their own Microservices. However, this documentation is only relevant for this one team and therefore does not have to be standardized.

Outdated Documentation

A common problem concerning the documentation of any software is that the documentation gets easily outdated and then documents a state which is not up to date anymore. Therefore, the documentation should be versioned together with the code. Besides, the documentation should be created from information, which is anyhow present in the system. For instance, the list of all used libraries can be taken from the build system since exactly this information is needed during the

compilation of the system. Which other Microservices are used can be obtained from Service Discovery. This information can for instance be used to create firewall rules when a firewall is supposed to be used to protect the communication between the Microservices. In summary, the documentation does not have to be maintained separately, but results from the anyhow available information.

Access to Documentation

The documentation can be part of the artifacts which are created during the build. In addition, there can be a run-time interface which allows to read out metadata. Such an interface can correspond to the otherwise common interfaces for monitoring and for instance provide JSON documents via HTTP. In this way, the metadata are only an additional information Microservices provide at run-time.

In a service template it can exemplarily be shown how the documentation is created. The service template can then form the basis for the implementation of new Microservices. When the service template already contains this aspect, it facilitates the implementation of a standard-conform documentation. In addition, at least the formal characteristics of the documentation can be checked by a test.

8.14 Conclusion

The domain architecture of a Microservice-based system is essential because it influences not only the structure of the system, but also the organization (section 8.1). Unfortunately, especially for Microservices tools for dependency management are rare so that teams have to develop home-made solutions. However, often an understanding of the implementation of the individual business processes will be sufficient and an overview of the entire architecture is not really necessary (section 8.2).

For an architecture to be successful it has to be permanently adjusted to the changing requirements. For Deployment Monoliths there are numerous refactoring techniques to achieve this. Such possibilities do also exist for Microservices – however without the support of tools and with much higher hurdles (section 8.3). Still Microservice-based systems can be sensibly developed further – for instance by starting initially with few large Microservices and creating over time more and more Microservices (section 8.4). An early distribution into many Microservices entails the risk to end up with a wrong distribution.

A special case is the migration of a legacy application to a Microservice-based architecture (section 8.5). In this case, the code base of the legacy application can be divided into Microservices - however this can lead to a bad architecture due to the often bad structure of the legacy application. Alternatively, the legacy application can be supplemented by Microservices, which replace functionalities of the legacy application in a stepwise manner.

Event-driven Architecture (section 8.6) can serve to uncouple the logic in the Microservices. This allows an easy extensibility of the system.

Defining the technological basis is one of the tasks of an architecture (section 8.7). In case of Microservice-based systems this does not relate to the definition of a shared technology stack for implementation, but to the definition of shared communication protocols, interfaces, monitoring and logging. Additional technical functions of the entire system are coordination and configuration (section 8.8). In this area tools can be selected, which all Microservices have to employ. Alternatively, one can do without a central configuration and instead leave each Microservice to bring along its own configuration.

For Service Discovery (section 8.9) likewise a certain technology can be chosen. A solution for Service Discovery is in any case sensible for a Microservice-based system – except messaging is used for communication. Based on Service Discovery Load Balancing can be introduced (section 8.10) to distribute the load across the instances of the Microservices. Service Discovery knows all instances, the load balancing distributes the load to these instances. Load Balancing can be implemented via a central load balancer, via Service Discovery or via one load balancer per client. This provides the basis for scalability (section 8.11). This allows a Microservice to process more load by scaling up.

Microservices have a significantly higher technical complexity than Deployment Monoliths. Operating systems, networks, load balancer, Service Discovery and communication protocols all become part of the architecture. Developers and architects of Deployment Monoliths are largely spared from these aspects. Thus architects have to deal with entirely different technologies and have to carry out architecture at an entirely different level.

In the area of security a central component has to take over at least authentication and parts of authorization. The Microservices should then settle the details of access (section 8.12). In order to obtain certain information from a system, which

is composed of many Microservices, the Microservices have to possess a standardized documentation (section 8.13). This documentation can for instance provide information about the used libraries – to compare them with open source license regulations or to remove security issues when a library has a security gap.

The architecture of a Microservice-based system is different from classical applications. Many decisions are only made in the Microservices, while topics like monitoring, logging or Continuous Delivery are standardized for the entire system.

Essential Points

- Refactoring between Microservices is laborious. Therefore, it is hard to change the architecture at this level. Accordingly, the continued development of the architecture is a central point.
- An essential part of the architecture is the definition of overarching technologies for configuration and coordination, Service Discovery, Load Balancing, security, documentation and meta data.

1. Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley,2003, ISBN 978-0-32112-521-7↩
2. Martin Fowler: Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999, ISBN 978-0201485677↩
3. Sam Newman: Building Microservices: Designing Fine-Grained Systems, O'Reilley Media, 2015, ISBN978-1-4919-5035-7↩
4. Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns:↩

9 Integration and Communication

Microservices have to be integrated and need to communicate. This can be achieved at different levels (Fig. 37). Each approach has certain advantages and disadvantages. Besides, at each level different technical implementations of integration are possible.

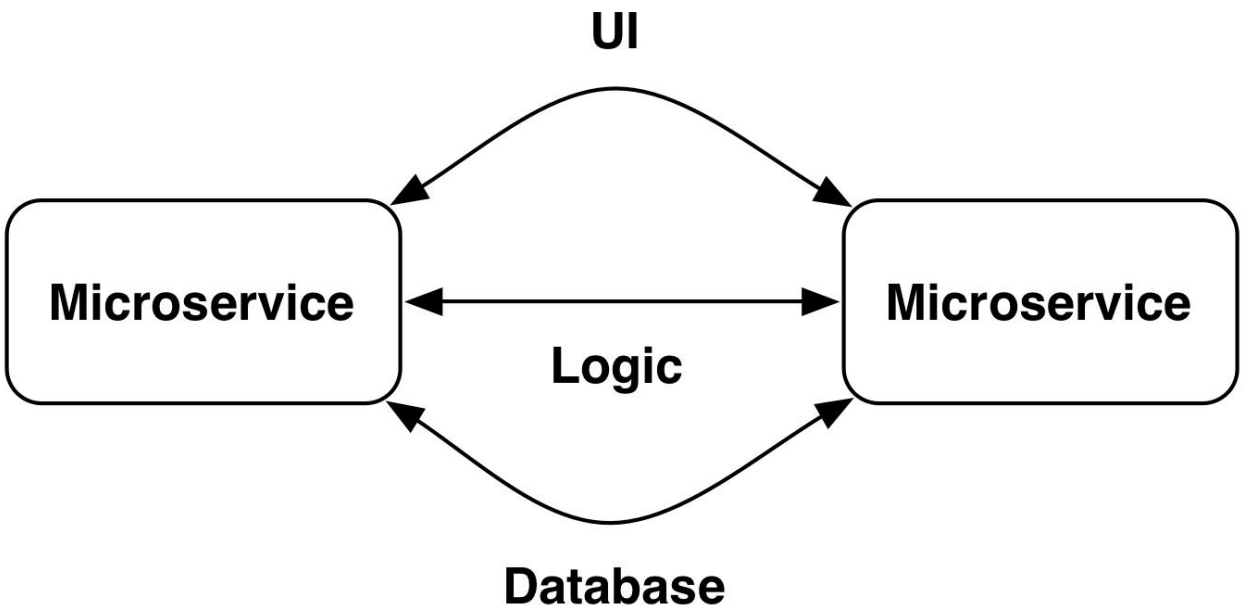


Fig. 37: Different levels of integration

- Microservices contain a graphical user interface. Therefore, Microservices can be integrated at the level of the UI. This type of integration is introduced in section 9.1.
- Also the logic can be integrated. Microservices can use REST (section 9.2), SOAP or RCP (section 9.3) or messaging (section 9.4) to achieve the integration of logic.
- Finally, the integration can be performed at the level of the database via data replication (section 9.5).

General rules for the design of interfaces are provided in section 9.6.

9.1 Web and UI

Microservices should bring their own UI along. This allows to implement functionalities even in those cases in only one Microservice, when the changes also affect the UI. At the level of the entire system it is necessary to jointly integrate the UIs of the Microservices. This can be achieved by different approaches, which are reviewed in the innoQ Blog.

Multiple Single-Page-Apps

Single-Page-App (SPA) implements the entire UI with just one HTML page. The logic is implemented in JavaScript, which dynamically changes parts of the page. The logic can manipulate the URL displayed in the browser so that bookmarks and other typical browser features can be used. However, SPAs are not in line with the original web thinking: SPAs marginalize HTML as central web technology. Most logic is implemented in JavaScript. Classical web architectures implement logic nearly exclusively on the server.

SPAs are especially advantageous when complex interactions or offline ability are required. Google's GMail is an example which also decisively shaped the term SPA. Mail clients are often native applications. GMail as SPA offers nearly the same comfort.

There are different technologies for the implementation of Single-Page-Apps:

- AngularJS is very popular. AngularJS has amongst other features a bidirectional UI data-binding: If the JavaScript code assigns a new value to an attribute of a bound model, the view components displaying the value are automatically changed. The binding works also from UI to the code: AngularJS can bind the input of a user to a JavaScript variable. Furthermore, AngularJS can render HTML templates in the browser. Thereby JavaScript code can also generate complex DOM structures. In that case the entire frontend logic is implemented in the JavaScript code running the browser. AngularJS was made by Google who put the framework under the very liberal MIT license.
- Ember.js works in line with the principle Convention over Configuration and represents in essence the same features like AngularJS. Via the supplementary module Ember Data it offers a model-driven approach for accessing REST resources. Ember.js is under the MIT license and is looked after by developers from the open source community.
- Ext JS offers apart from an MVC approach also components which developers can compose to a UI similar like for Rich Client applications. Ext

JS is available as Open Source under GPL v3.0. However, for commercial development a licence has to be bought from the manufacturer Sencha.

SPA per Microservice

In case of Microservices with Single Page Apps each Microservice can bring its own SPA along (Fig. 38). The SPA can call the Microservice for instance via JSON/REST. This is especially easy to implement with JavaScript. Between the SPAs a link can be used.

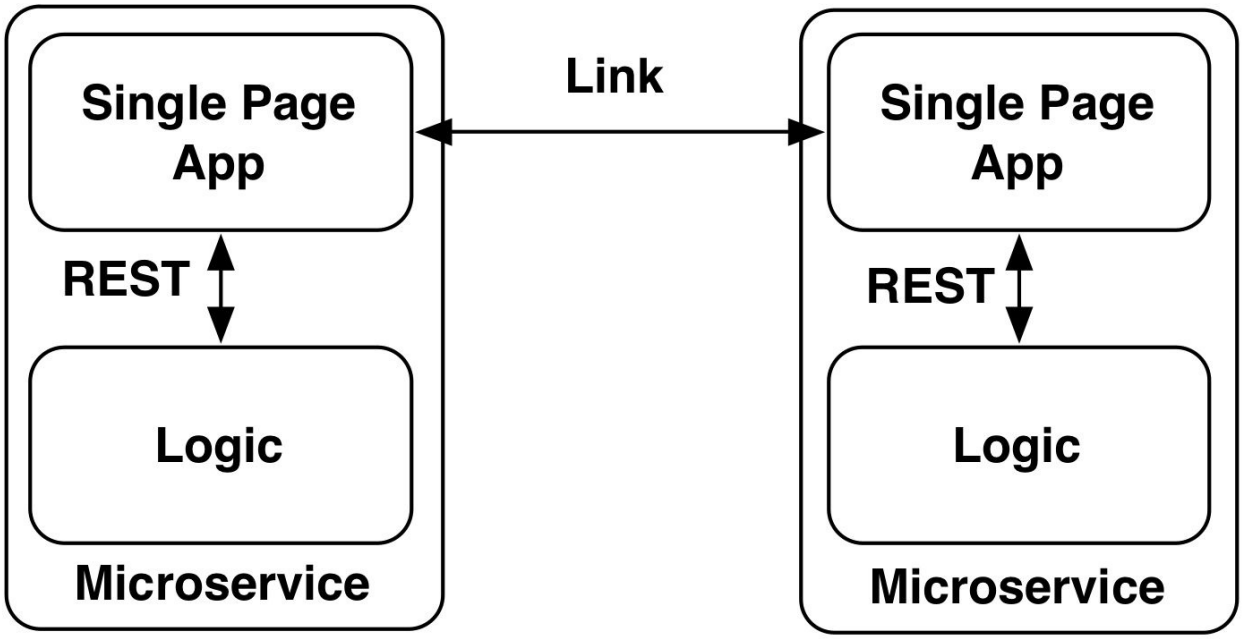


Fig. 38: Microservices with Single Page Apps

Thereby the SPAs are completely separate and independent. New versions of a SPA and of the associated Microservice can be rolled out without further ado. However, a tighter integration of SPAs is difficult. When the user switches from one SPA to another, the browser loads a new web page and starts a different JavaScript application. Even modern browsers need so much time for this that this approach is only sensible when switching between SPAs is an exception.

Asset Server for Uniformity

Besides SPAs can be heterogeneous. Each brings its own individually designed UI along. However, this issue can be solved by using an Asset Server. Such a server is used to provide JavaScript files and CSS files for the applications. When the SPAs of the Microservices are only allowed to use these kinds of resources via the Asset Server, a uniform user interface can be achieved. To accomplish this, a Proxy Server can distribute requests to the Asset Server and the Microservices.

Thereby it will look for the web browser as if all resources as well as the Microservices possess a shared URL. This approach avoids that security rules prohibit the use of the contents because they originate from different URLs. Caching can then reduce the time for loading the applications. When only JavaScript libraries, which are stored on the Asset Server, are allowed to be used, the choice of technologies for the Microservices can be reduced. Therefore, uniformity and free technology choice are competing aims.

Besides the shared assets will create code dependencies between the Asset Server and all Microservices. A new version of an asset entails the modification of all Microservices which use this asset. In the end, they have to modified in a way that they use the new version. Such code dependencies endanger the independent deployment and therefore should be avoided. Code dependencies in the backend are often a problem (compare section 8.3). In fact, such dependencies should also be reduced in the frontend. However, in such a case an Asset Server is rather a problem than a solution.

Apart from an Asset Server UI guidelines can be helpful, which describe the design of the application in more detail and thereby enable a uniform approach also at different levels. This allows for the implementation of a uniform UI even without a shared Asset Server and code dependencies.

In addition, it has to be ensured that the SPAs possess a uniform authentication and authorization so that the users do not have to log in multiple times. An OAuth2 or a shared signed cookie can be a solution for this (compare also section 8.12).

JavaScript can only access data which are available under the domain from where the JavaScript code originates. This Same Origin Policy avoids that JavaScript code can read data from other domains. When all Microservices are accessible to the outside under the same domain due to a Proxy, this is no limitation. Otherwise the policy has to be deactivated when the UI of a Microservice is supposed to access the data of another Microservice. This problem can be solved by CORS (Cross Origin Resource Sharing) with which the server delivering the data can also allow JavaScript from other domains. Another option is to offer all SPA and REST services to the outside only via one domain so that an access across domains is not necessary. In this way also the access to shared JavaScript code on an Asset Server can be implemented.

A Single Page App for all Microservices

The division into multiple SPAs results in a strict separation of the frontends of the Microservices. If for instance a SPA is responsible for registering orders and another one for a fundamentally different use case like reports, the load times needed when changing between SPAs are still acceptable. Maybe the user groups are even different so that changes between the applications do not occur.

There are cases when a tighter integration of the user interfaces of the Microservices is necessary. For example, in an order also details about the items can be displayed. Displaying the order is the responsibility of one Microservice, displaying the items is performed by another. In this case the SPA can be distributed into modules. Each module belongs to another Microservice and therefore to another team. The modules should be deployed separately. They can for instance be stored on the server in individual JavaScript files and possess separate Continuous Delivery pipelines. Besides there have to be suitable conventions for the interfaces. For example, only the sending of events might be allowed. Events uncouple the modules because the modules communicate only changes in the states, but not how other modules have to react to them.

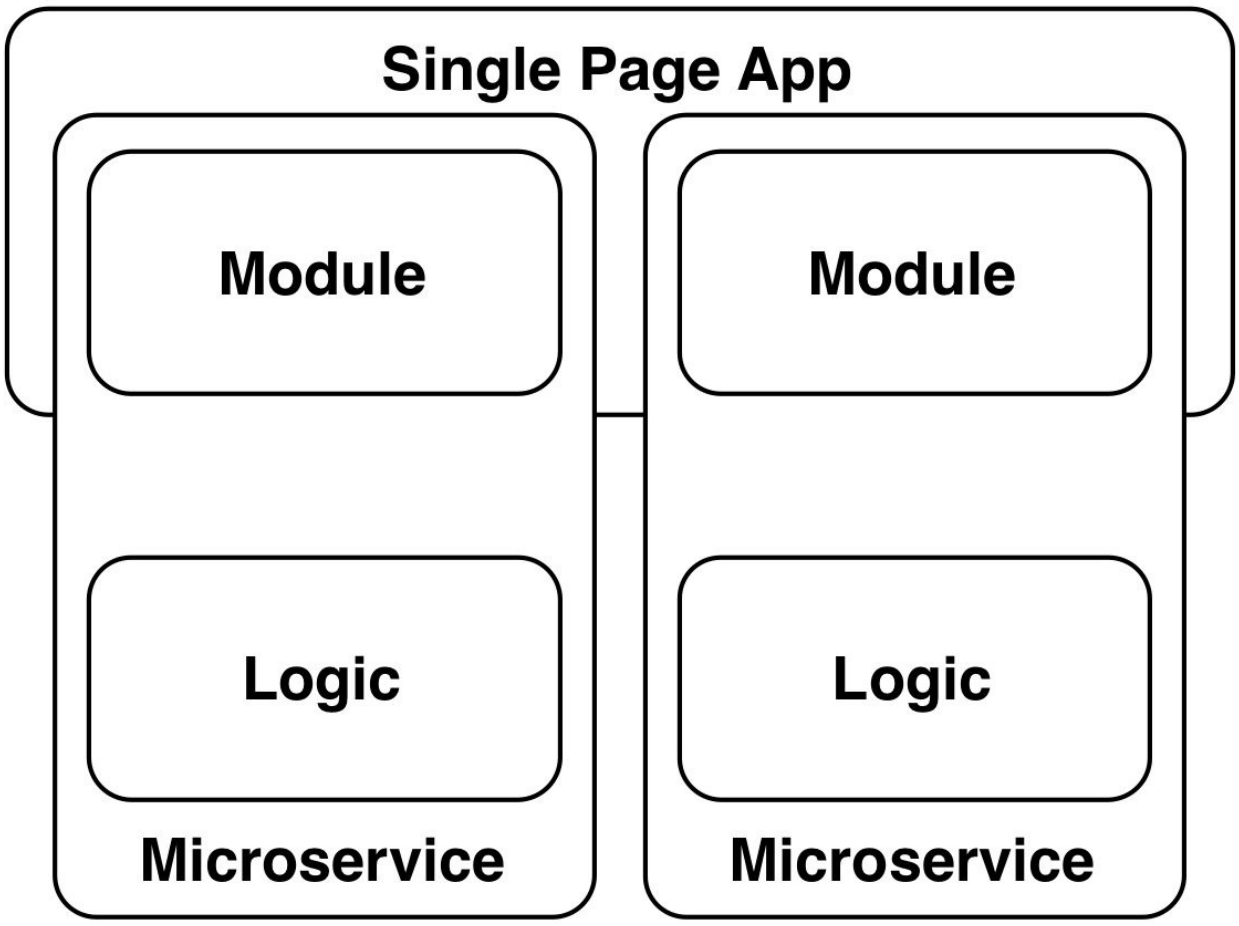


Fig. 39: Close integration of Microservices sharing one Single-Page-App

AngularJS for instance has a module concept which allows to implement individual parts of the SPA in separate units. A Microservice could provide an AngularJS module for displaying the user interface of the Microservice. The model can integrate, if necessary, AngularJS modules of other Microservices.

However, such an approach has disadvantages:

- Deploying the SPA is often only possible as complete application. When a module is modified, the entire SPA has to be rebuilt and deployed. This has to be coordinated between the Microservices, which provide the modules of the application. In addition, the deployment of the Microservices on the server has to be coordinated with the deployment of the modules since the modules call the Microservices. This necessity for coordination for the deployment of modules of an application should be avoided by Microservices.
- The modules can call each other. Depending on the way calls are implemented, changes to a module can entail that also other modules have to changed, for instance because an interface has been modified. When the modules belong to separate Microservices, this enforces again a coordination across Microservices, which should be avoided.

For SPA modules a much closer coordination is necessary than for links between applications. On the other hand the SPA modules offer the advantage that UI elements from different Microservices can be simultaneously displayed to the user. However, this approach closely couples the Microservices at the level of the UI. The SPA modules correspond to the module concepts which also exist in other programming languages and cause a simultaneous deployment. Thus, the Microservices, which really should be independent of each other, are combined at the UI level in one shared deployment artifact. Therefore, this approach undoes one of the most important advantages of a Microservice-based architecture – the independent deployment.

HTML Applications

Another option for implementing the user interface are HTML-based user interfaces. Every Microservice has one or more web pages which are generated on the server. The web page can also use JavaScript. Here, contrary to SPAs, only

a new HTML web page and not necessarily an application is loaded by the server when changing between web pages.

ROCA

ROCA (Resource Oriented Client Architecture) offers the possibility to arrange the handling of JavaScript and dynamical elements in HTML user interfaces. ROCA views itself as alternative to SPAs. In ROCA the role of JavaScript is limited to optimizing the usability of the web pages. JavaScript can facilitate their use or can add effects to the HTML web pages. However, the application has to remain useable without JavaScript. It is not the purpose of ROCA that users really use web pages without JavaScript. The applications are only supposed to use the architecture of the web, which is based on HTML and HTTP. Especially when a web application is supposed to be divided into Microservices, ROCA reduces the dependencies and simplifies the division. Between Microservices the coupling of the UI can be achieved by links. For HTML applications links are the usual tool for navigating between the web pages and represent a natural integration. They are no foreign body like in the case of SPAs.

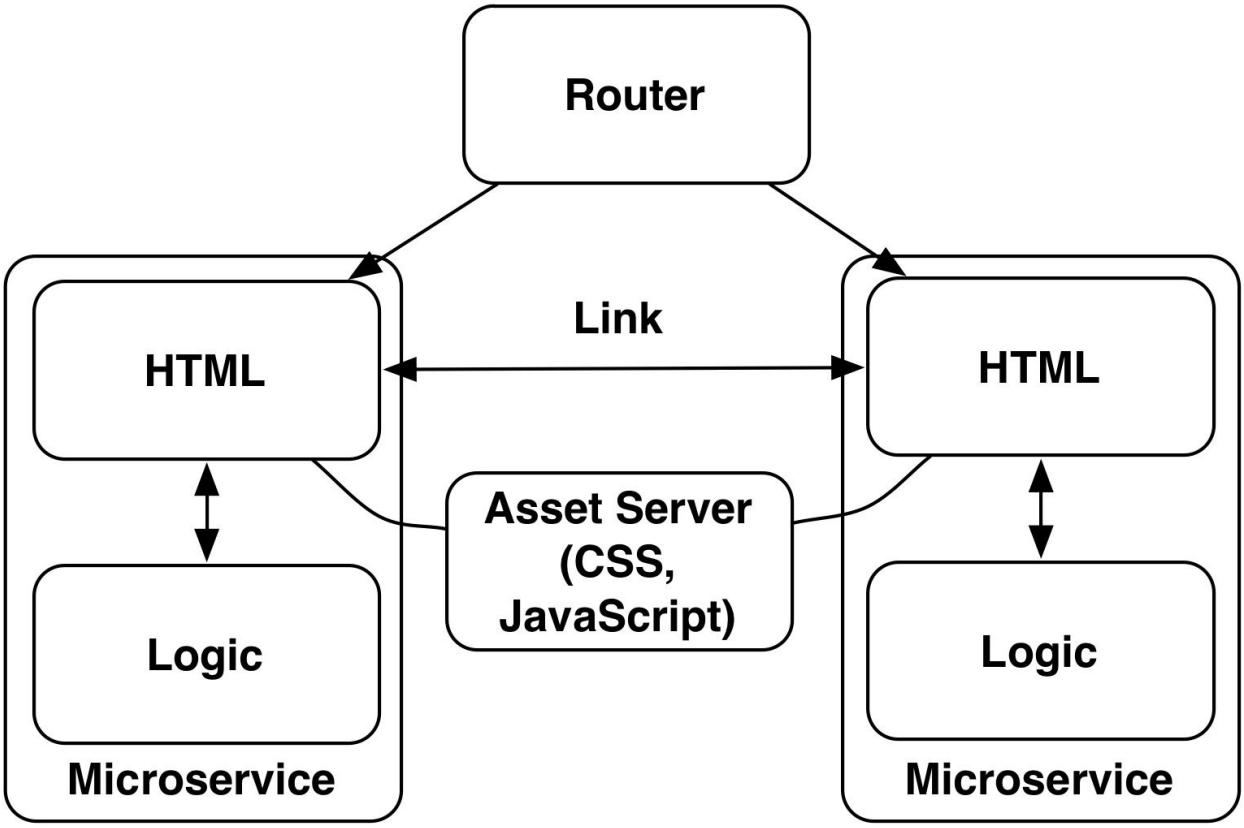


Fig. 40: HTML user interface with an asset server

To support the uniformity of the HTML user interfaces, the Microservices can use a shared Asset Server like in the case of SPAs (Fig. 40). It contains all CSS and JavaScript libraries. When the teams in addition define design guidelines for the HTML web pages and look after the assets on the Asset Server, the user interfaces of the different Microservices will be largely identical. However, as described before, this will lead to code dependencies between the UIs of the Microservices.

Easy Routing

To the outside the Microservices should appear like a single web application – ideally with one URL. This also facilitates the shared use of assets since the Same Origin Policy is not violated. However, from the outside user requests have to be directed to the right Microservice. This is the function of the router. It can receive HTTP requests and forward them to one of the Microservices. This can be done based on the URL. How individual URLs are mapped to Microservices can be decided by rules, which can also be complex. The example application uses Zuul for this task (compare section 14.9). Reverse Proxies are an alternative. These can for instance be web servers like Apache httpd or nginx, which can direct requests to other servers. In the process the requests can be modified, URLs can

for instance be rewritten. However, these mechanisms are not as flexible as Zuul, which is very easy to extend with home-grown code.

When the logic in the router is very complex, this can cause problems. If this logic has to be changed because a new version of a Microservice is brought into production, an isolated deployment is not easy anymore. This endangers the independent development and the independent deployment of the Microservices.

Arrange HTML with JavaScript

In some cases, a closer integration is necessary. It can happen that information originating from different Microservices is displayed on one HTML web page. For example a web page might display order data from one Microservice and data concerning the ordered items from another Microservice. In that case one router is not sufficient anymore. A router can only allow that a Microservice generates a complete HTML web page.

A simple solution which employs the architecture presented in Fig. 40 is based on links. AJAX allows to load the content of a link from another Microservice. Afterwards the link is replaced by the thereby received HTML. In the example a link to an item could be transformed into an HTML description of this item. This allows to implement the logic for the presentation of a product in one Microservice, while the design of the entire web page is implemented in another Microservice. The entire web page would be the responsibility of the order Microservice, while the presentation of the products would be the responsibility of the product Microservice. This enables the continued independent development of both Microservices and displaying presentations from both components. If the presentation of the items has to be changed or new products necessitate a revised presentation, these modifications can be implemented in the product Microservice. The entire logic of the order Microservice remains unchanged.

Another example for this approach is Facebook's BigPipe. It optimizes not only the load time, but allows also the composition of web pages from pagelets. A custom implementation can use JavaScript to replace certain elements of the web page by other HTML. This can be links or **div**-elements like the ones also otherwise used for structuring web pages. Such a **div**-element can be replaced by HTML code.

However, this approach causes relatively long load times. It is mainly advantageous when the web UI anyhow uses a lot of JavaScript and when there

are not many transitions between web pages.

Frontend Server

Fig. 41 shows an alternative for a tight integration. A frontend server composes the HTML web page from HTML snippets, which are each generated by a Microservice. Assets like CSS or JavaScript libraries can also be stored in the frontend server. Edge Side Includes (ESI) represents a possibility to implement this concept. ESI offers a relatively simple language for combining HTML from different sources. With ESI caches can supplement static content – for instance the skeleton of a web page– with dynamic content. In this way caches can help with the delivery of web pages, even if they contain dynamic content. Proxies and caches like Varnish or Squid implement ESI. Another alternative are Server Side Includes (SSI). They are very similar to ESIs, however, they are not implemented in caches, but in web servers. With SSIs web servers can integrate HTML snippets from other servers into HTML web pages. The Microservices can deliver components for the web page, which then will be assembled on the server. Apache httpd supports SSIs for instance with mod_include. nginx uses the ngx_http_ssi_module for the support of SSIs.

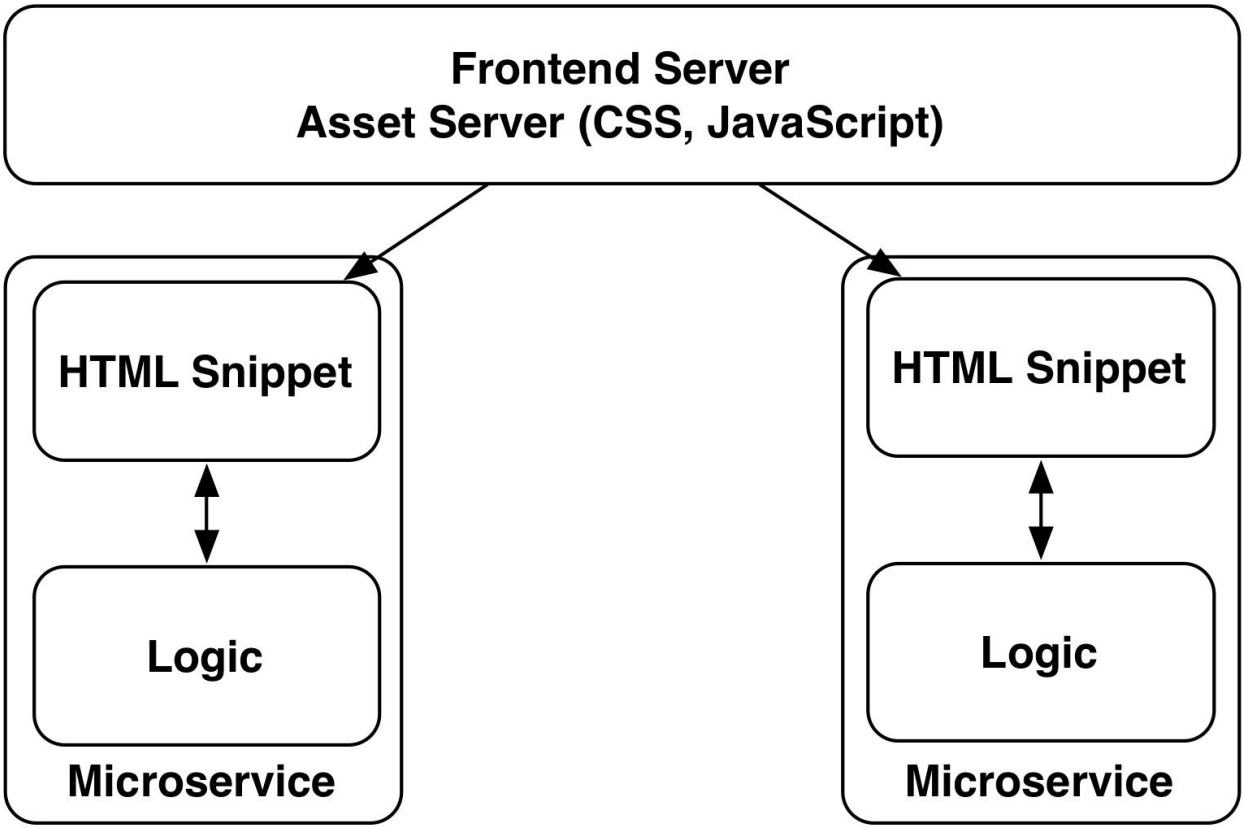


Fig. 41: Integration using a Frontend server

Portals also consolidate information from different sources on one web page. Most products use Java Portlets in line with the Java standard JSR 168 (Portlet 1.0) or JSR 286 (Portlet 2.0). Portlets can be brought into production independently of each other and therefore solve one of the central challenges surrounding Microservice-based architectures. In practice these technologies result frequently in complex solutions. Portlets behave technically very differently in comparison to normal Java web applications so that the use of many technologies from the Java environment is either difficult or impossible. Portlets allow the user to compose a web page from previously defined portlets. In this way the user can assemble for instance his/her most important information sources on one web page. However, this is not really necessary for creating a UI for Microservices. The additional features result in additional complexity. Therefore, portal servers which are based on portlets are not a really good solution for the web user interfaces of Microservices. In addition, they restrict the available web technologies to the Java field.

Mobile Clients and Rich Clients

Web user interfaces do not need any installation of software on the client. The web browser is the universal client for all web applications. On the server site the deployment of the web user interface can easily be coordinated with the deployment of the Microservice. The Microservice implements a part of the UI and can deliver the code of the web user interface via HTTP. This allows for a relatively easy coordinated deployment of client and server.

For mobile apps, Rich Clients, or desktop applications the situation is different: A software has to be installed on the client. This client application is a Deployment Monolith, which has to offer an interface for all Microservices. If the client application is supposed to comprise functionalities of different Microservices, it would technically have to be modularized, and the individual modules like the associated Microservices would have to be brought into production independently of each other. However, this is not possible since the client application is a Deployment Monolith. A SPA can also easily turn into a Deployment Monolith. Sometimes a SPA is used to separate the development of client and server. In a Microservices context such a use of SPAs is not desirable.

When a new feature is implemented in a Microservice, which also requires modifications of the client application, this change cannot solely be rolled out via a new version of the Microservice. In addition, a new version of the client application has to be delivered. However, it is unrealistic to deliver the client application over and over again for each small change of a feature. If the client applications is supposed to be available in the app store of a mobile operation system, an extensive review of each version is necessary. If multiple changes are supposed to be delivered together, the change has to be coordinated. And the new version of the client application has to be coordinated with the Microservices so that the new versions of the Microservices are ready in time. This results in deployment dependencies between the Microservices, which should really be avoided.

Organizational Level

At the organizational level there is often a designated team for developing the client application. In this manner the division into an individual module is also implemented at the organizational level. Especially when different platforms are supported, it is unrealistic that there is one developer in each Microservice team for each platform. The developers are going to form one team for each platform. This team has to communicate with all Microservice teams, which offer Microservices for mobile applications. This can necessitate a lot of

communication. However, Microservices have set out to avoid such excessive communication requirements. Therefore, the Deployment Monolith poses a challenge for client applications at the organizational level.

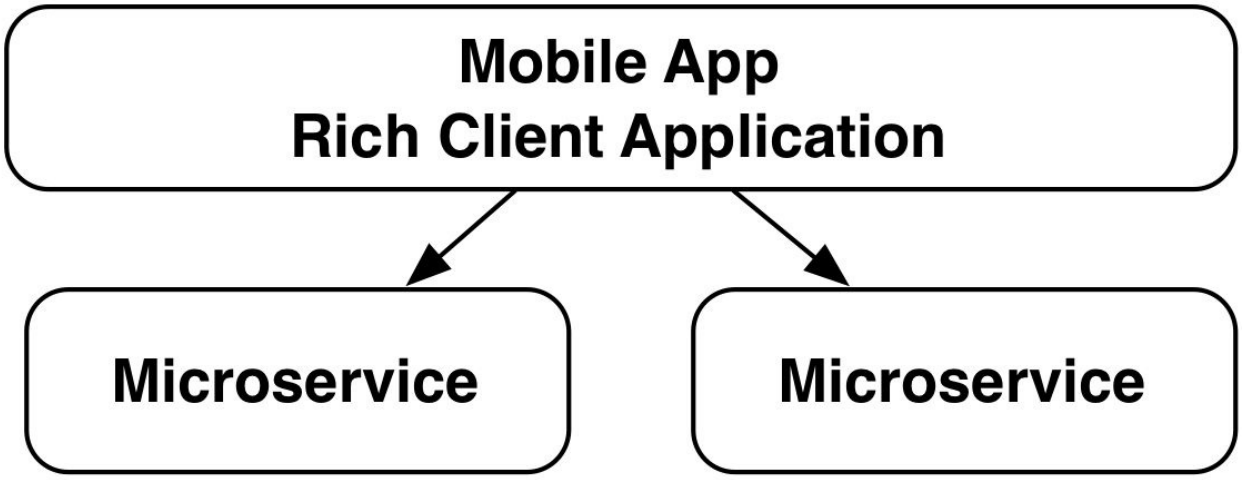


Fig. 42: Mobile Apps and Rich Client are Deployment Monoliths that integrate multiple Microservices.

One possible solution is to develop new features initially for the web. Each Microservice can directly bring functionalities into the web. Upon a release of the client application these features will also be available there. However, in that case each Microservice needs to support a certain set of features for the web application and, where required, another set for the client application. In exchange this approach can keep the web application and the mobile application uniform. It supports an approach where the domain-based teams provide features of the Microservices to mobile users as well as to web users. Mobile applications and web applications are only two channels to offer the same functionalities.

Backend for each Frontend

However, the requirements can also be entirely different. For instance, the mobile application can be a largely independent application which is supposed to be developed further as independently of the Microservices and the web user interface as possible. Often the use cases of the mobile application are so different from the use cases of the web application that a separate development is required due to the differences in the features.

In such cases the approach depicted in Fig. 43 can be sensible: The team for the mobile app resp. the Rich Client has a number of developers who implement a special backend. This allows to also develop functionalities of the mobile app

independently in the backend, because at least a part of the requirements for the Microservices can be implemented by developers from the same team. In that case it should not happen that logic for the mobile app is implemented in the Microservice, which really belongs into a backend Microservice. However, the backend for a mobile application differs from other APIs. Mobile clients have little bandwidth and a high latency. Therefore, APIs for mobile devices are optimized for getting by with as few calls as possible and for only transfering really essential data. This is also true for Rich Clients, however not exactly to the same extent. The adaption of an API to the specific requirements of a mobile applications can be implemented in a Microservice, which is implemented by the frontend team.

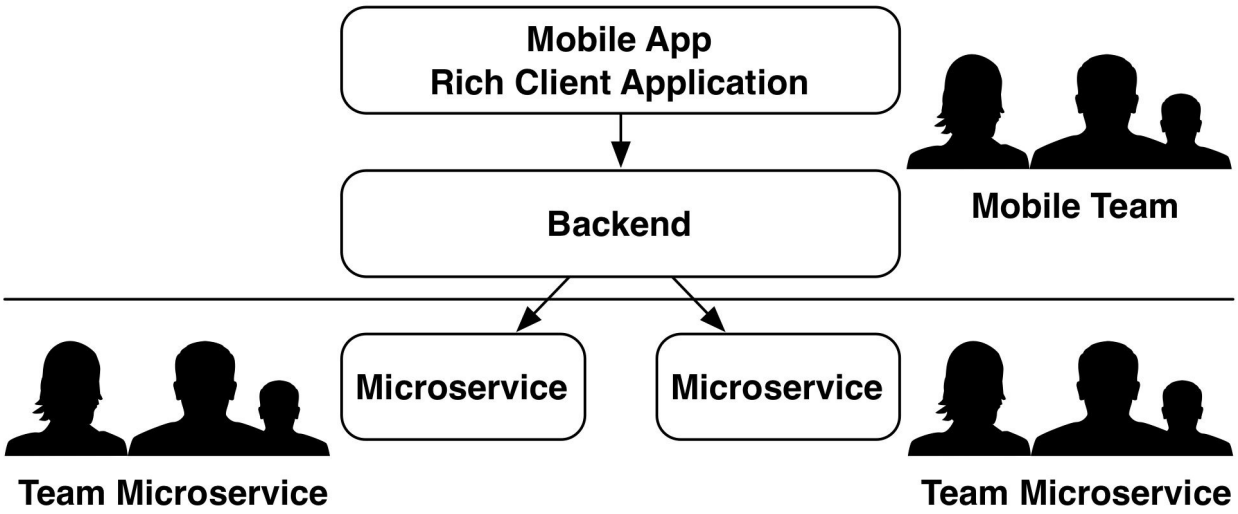


Fig. 43: Mobile Apps or Rich Clients with their own backend

In a mobile app a user interaction should rapidly lead to a reaction of the app. When it is necessary to call a Microservice as reaction to a user interaction, this can already conflict with this aim. If there are multiple calls, the latency will increase further. Therefore, the API for a mobile App should be optimized for delivering the required data with as few calls as possible. Also these optimizations can be implemented by a backend for the mobile app.

The optimizations can be implemented by the team which is responsible for the mobile app. Thereby the Microservices can offer universally valid interfaces while the teams for the mobile apps can assemble their special APIs by themselves. Due to that the mobile app teams are not so dependent anymore on the teams which are responsible for the implementation of the Microservices.

To modularize web applications is simpler than the modularization of mobile apps, especially when the web applications are based on HTML and not on SPAs. For mobile apps or Rich Client Apps it is much more difficult since they form an individual deployment unit and cannot be easily divided.

The architecture shown in Fig. 43 makes it possible to reuse Microservices for different clients. At the same time, it is an entry into a layered architecture. The UI layer is separated from the Microservices and is implemented by another team. In that case requirements have to be implemented by multiple teams. Microservices were meant to avoid exactly this. Besides this architecture entails the danger that logic is implemented in the services for the client application, which really belongs in the Microservices. Therefore, this solution does not only have advantages.

Try and Experiment

This section presented as alternative for web applications a SPA per Microservice, a SPA with modules per Microservice, an HTML application per Microservice and a frontend server with HTML snippets. Which of these approaches would you choose? Why?

How would you deal with mobile apps? One option would be a team with backend developers – or would you rather choose a team without backend developers?

9.2 REST

Microservices have to be able to call each other in order to implement logic together. This can be supported by different technologies.

REST (Representational State Transfer) is one option to enable communication between Microservices. REST is the term for the fundamental approaches of the WWW:

- There is a plethora of resources which can be identified via URIs. URI stands for Uniform Resource Identifier. It unambiguously and globally identifies resources. URLs are practically the same as URIs.

- The resources can be manipulated via a fixed set of methods. In the case of HTTP these are for instance GET for requesting a resource, PUT for storing a resource and DELETE for deleting a resource. The methods semantics are rigidly defined.
- There can be different representations for resources – for instance as PDF or HTML. HTTP supports the so-called Content Negotiation via the Accept Header. In this manner the client can determine which data representation it can process. The Content Negotiation allows for instance to display resources in a way that is human-readable and to provide them at the same time under the same URL in a machine-readable manner. The client can communicate via an Accept Header whether it only accepts human-readable HTML or only JSON.
- Relationships between resources can be represented by links. Links can point to other Microservices thereby enabling the integration of logic of different Microservices.
- The servers in a REST system are supposed to be stateless. Therefore HTTP implements a stateless protocol.

The limited vocabulary represents the exact opposite of what object-oriented systems employ. Object-orientation focuses on a specific vocabulary with specific methods for each class. The REST vocabulary can likewise execute complex logic. When data validations are necessary, this can be checked at the POST or PUT of new data. If complex processes are supposed to be represented, a POST can start the process, and subsequently the state can be updated. The current state of the process can be fetched by the client under the known URL via GET. Likewise, POST or PUT can be used to initiate the next state.

Cache and Load Balancer

A RESTful HTTP interface can very easily be supplemented with a cache: Since RESTful HTTP uses the same HTTP protocol like the web, a simple web cache is sufficient. Likewise, the usual HTTP Load Balancer can also be used for RESTful HTTP. The power of these concepts is impressively illustrated by the size of the WWW. This size is only possible due to the properties of HTTP. HTTP for instance also possesses simple and useful mechanisms for security – not only encryption via HTTPS, but also authentication with HTTP Headers.

HATEOAS

HATEOAS (Hypermedia as the Engine of Application State) is another important component of REST. The relationships between the resources are modeled by

links. Therefore, a client only has to know an entry point. From there it can go on navigating at will and thereby locate all data in a stepwise manner. In the WWW it is for instance possible to start from Google and from there to reach practically the entire web via links.

REST describes the architecture of the WWW and thereby the largest integrated computer system. However, REST could also be implemented with other protocols. It is an architecture which can be implemented with different technologies. The implementation of REST with HTTP is called RESTful HTTP. When RESTful HTTP services exchange data as JSON or XML instead as HTML, this approach allows to exchange data and not only to access web pages.

Microservices can also profit from HATEOAS. HATEOAS does not have a central coordination, just links. This fits very well to the concept that Microservices should have as little central coordination as possible. In case of REST clients know only entry points based on which they can discover the entire system. Therefore, in a REST-based architecture services can be moved in a manner that is transparent for the client. The client simply gets new links. A central coordination is likewise not necessary for this. The REST service just has to return different links. In the ideal case the client only has to understand the fundamentals of HATEOAS and then can navigate via links to any data in the Microservice system. The Microservice-based systems on the other hand can modify their links and thereby change the distribution of functionalities between Microservices. Even extensive architecture changes can be kept transparent.

HAL

HATEOAS is a concept. HAL is a possibility to implement it. It is a standard describing how the links to other documents should be contained in a JSON document. Thereby HATEOAS is very easy to implement especially in JSON/RESTful HTTP services. The links are separate from the actual document. This allows to implement links to details or to independent data sets.

XML

XML has a long history as data format. It is easy to use together with RESTful HTTP. There are different type systems for XML which can determine whether an XML document is valid. This is very useful for the definition of an interface. Among the languages for the definition of valid data is for instance XML Schema (XSD) or RelaxNG. Some frameworks allow for the generation of code in order to administrate XML data, which correspond to such a schema. Via XLink XML

documents can contain links to other documents. This enables the implementation of HATEOAS.

HTML

XML was designed to transfer data and documents. To display the information is the task of different software. Meanwhile HTML has a similar approach as XML: HTML defines only the structures. The display occurs via CSS. For the communication between processes HTML documents can be sufficient because in modern web applications documents contain only data - just like XML. In a Microservices world this approach has the advantage that the communication to the user and between the Microservices employs the same format. This reduces the effort. Thereby it gets even easier to implement Microservices which contain a UI and a communication option for other Microservices.

JSON

JSON (JavaScript Object Notation) is a representation of data which is especially optimized for JavaScript. Like JavaScript the data are dynamically typed. However, meanwhile there are in fact suitable JSON libraries for all programming languages. In addition there are type systems likeJSON Schema, which supplement JSON with an appropriate validation. With that JSON is not inferior at all anymore to data formats like XML.

Protocol Buffer

Binary protocols like Protocol Buffer can also be used instead of text-based data representations. This technology has been designed by Google to represent data more efficiently and to achieve a higher performance. There are implementations for many different programming languages so that Protocol Buffer can be universally used similar to JSON or XML.

RESTful HTTP is synchronous.

RESTful HTTP is synchronous: Typically a service sends out a request and waits for a response which is subsequently analyzed in order to continue with the program sequence. This can cause problems in case of long latency times within the network. It can lengthen the processing of a request since responses of other services have to be waited for. Besides, after a certain waiting time the request has to be aborted because it is likely that the request is not going to be answered at all. Possible reasons are that the server is not available at the moment or that

the network has a problem. Correctly handled timeouts increase the stability of the system (section 10.5).

The failure may not result in the failure of additional services. Therefore, via the timeout it has to be ensured that the particular system still responds and the failure does not propagate.

9.3 SOAP and RPC

It is possible to build a Microservices-based architecture on SOAP. SOAP uses also HTTP like REST, but employs only POST messages to transfer data to a server. In the end a SOAP calls a method on a certain object on the server. Therefore SOAP is an RPC mechanism (Remote Procedure Call), which calls methods in a different process.

SOAP lacks mechanisms like HATEOAS, which allow to flexibly handle relationships between Microservices. The interfaces have to be completely defined by the server and known on the client.

Flexible Transport

SOAP can convey messages with different transport mechanisms. It is for instance possible to receive a message via HTTP and to subsequently sent it on as message via JMS or as email via SMTP/POP. SOAP-based technologies also support the forwarding of requests. For example, the security standard WS-Security can encrypt or sign parts of a message. Afterwards the parts can be sent on to different services without having to be decrypted. The sender can send a message in which some parts are encrypted. This message can be processed via different stations. Each station can process a part of the message or send it to other recipients. Finally, the encrypted parts will arrive at their final recipients – and only there they have to be decrypted and processed.

SOAP has many extensions for special use contexts. The different extensions from the WS-*-environment comprise for instance transactions and the coordination of web services. In this way a complex protocol stack can arise. The interoperability between the different services and solutions can suffer due to the complexity. Some technologies are also not very sensible for Microservices. For example, a coordination of different Microservices is problematic as this will result in a coordination layer, and modifications of a business process will probably concern the coordination of the Microservices and also the Microservices themselves.

When the coordination layer comprises all Microservices, a Monolith is created which also has to be changed upon each modification. This contradicts the Microservices idea of independent deployment. WS-* is rather in line with such concepts as SOA.

Thrift

Another communication possibility is Apache Thrift. It uses a very efficient binary encoding like Protocol Buffer. Furthermore, Thrift can forward requests from a process with a programming language via the network to other processes. The interface is described in an interface definition specific for Thrift. Based on this definition different client and server technologies can communicate with each other.

9.4 Messaging

Another option for the communication between Microservices are messages and messaging systems. As the name suggests, these systems are based upon the sending of messages. The messages can result in a response which again is sent as message. Messages can go to one or multiple recipients.

Especially in case of distributed systems messaging solutions can demonstrate their advantages:

- Messages can still be transferred in case of network failures. The messaging system buffers them and delivers them when the network is available again.
- The guarantees can be further strengthened: The messaging system cannot only guarantee the correct transfer of the messages, but even their processing. If there was a problem during the processing of the message, the message can be transferred anew. A successful processing is possible when the error disappears after some time. Otherwise it will be attempted a couple more times to process the message until finally the message is discarded because it cannot be processed successfully.
- In a messaging architecture responses are transferred and processed asynchronously. Such architectures are well tuned to high latency times like they occur in the network. Waiting for a response is the usual case in such an architecture. Therefore the programming model always acts on the assumption of a high latency.
- The call of another service does not block the further processing. Even if the response has not been received yet, the service can continue working and for

instance call additional services.

- The sender does not know the recipient of the message. The sender sends the message to a queue or a topic. There the recipient registers. Thereby sender and recipient are decoupled. There can even be multiple recipients without that the sender is aware of this. Besides the messages can be modified on their way. Data can be for instance supplemented or removed. In addition, messages can also be forwarded to entirely different recipients.

Messaging is also a good basis for certain architectures of Microservice-based systems like Event Sourcing (compare section 10.3) or Event-driven Architecture (section 8.6).

Messages and Transactions

Messaging offers a solution for transactional systems with Microservices. In a Microservice-based system the guarantees for transactions are hard to ensure when the Microservices call each other. In that case all Microservices would have to participate in a transaction. They are only allowed to write changes when all Microservices in the transaction have processed the logic without errors. This means that the changes would have to be held back for a very long time. That is bad for the performance since no new transaction can change the data meanwhile. Besides in a network it is always possible that a participant fails. In that case the transaction will remain open for a long time or might even not be closed at all. This will block changes to the data for a long time. Such problems arise for instance when the calling system crashes.

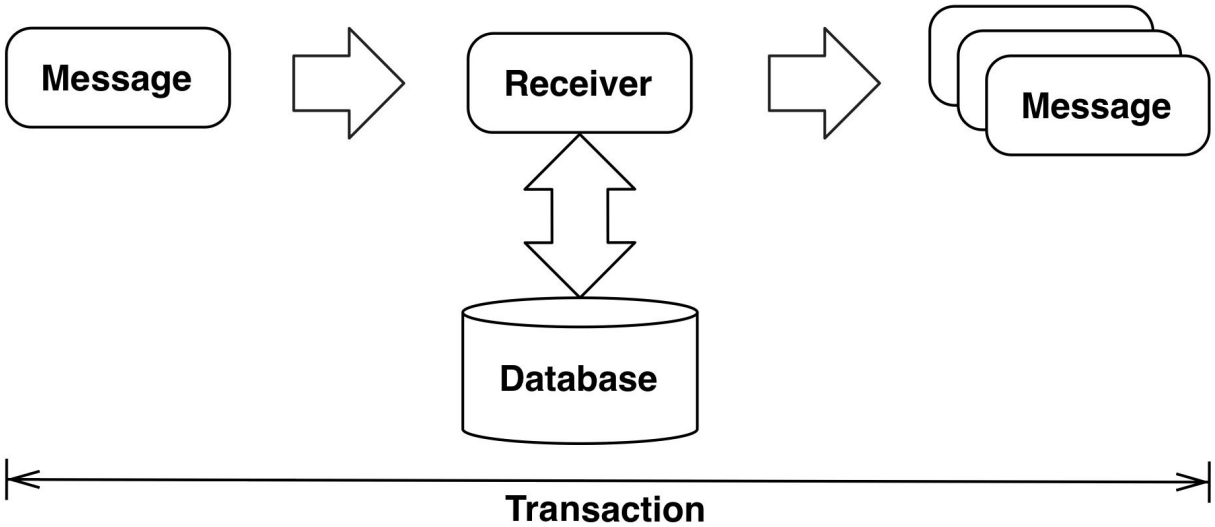


Fig. 44: Transactions and Messaging

In a messaging system transactions can be treated differently: The sending and receiving of messages is part of a transaction – just as for instance the writing and reading from the database (Fig. 44). When an error occurs during the processing of the message, all outgoing messages are canceled and the database changes are rolled back. In the case of success all these actions take place. The recipients of the messages can likewise be safeguarded transactionally. In that case the processing of the outgoing messages is subject to the same transactional guarantees.

The important point is that the sending and receiving of messages and the transactions on the database can be combined in one transaction. The coordination is taken care of by the infrastructure. No extra code needs to be written. For the coordination of messaging and databases the protocol Two Phase Commit (2PC) can be employed. This protocol is the usual solution for coordinating transactional systems like databases and messaging systems with each other. An alternative are products like Oracle AQ or ActiveMQ. They store the messages in a database. Then the coordination between database and messaging can simply be achieved by writing the messages as well as the data modifications in the same database transaction. Messaging and database are in the end the same systems in that case.

Messaging allows to implement transactions without the need for a global coordination. Each Microservice is transactional. The transactional sending of messages is ensured by the messaging technology. However, when a message cannot be processed, for instance due to invalid values, there is no possibility to roll the already processed messages back. Therefore, the correct processing of transactions is not given under all circumstances.

Messaging Technology

For the implementation of messaging a technology has to be used:

- AMQP (Advanced Message Queuing Protocol) is a standard. It defines a protocol with which messaging solutions can communicate on the wire with each other and with clients. An implementation of this standard is RabbitMQ, which is written in Erlang and is under Mozilla licence. Another implementation is for instance Apache Qpid.
- Apache Kafka focuses on high throughput, replication and fail safeness. Therefore, it is well suited for distributed systems like Microservices, especially the fail safeness is very helpful in this use context.

- 0MQ (also called ZeroMQ or ZMQ) gets along without a server and is therefore very light-weight. It has some primitivs which can be assembled into complexer systems. 0MQ is under the LGPL licence and written in C++.
- JMS (Java Messaging Service) defines an API, with which a Java application can receive messages and send them. In contrast to AMQP the specification does not define how the technology transfers messages on the wire. Since it is a standard, Java-EE server implement this API. Well known implementations are ActiveMQ and HornetQ.
- It is also possible to use ATOM Feeds for messaging. This technology is normally used to transfer blog contents. Clients can relatively easily request new entries of a blog. In the same manner a client can use ATOM to request new messages. ATOM is based on HTTP and therefore fits well in a REST environment. However, ATOM has only functionalities for delivering new information. It does not support more complex techniques like transactions.

For many messaging solutions a messaging server and therefore an additional infrastructure are required. This infrastructure has to be operated in a manner that prevents failures because failures would cause the communication in the entire Microservice-based system to break down. However, messaging solutions are mostly designed to achieve high availability for instance via clustering.

For many developers messaging is rather unfamiliar since it requires asynchronous communication. This makes it appear as rather complex. In most cases the calling of a method in a different process is easier to understand. With approaches like Reactive (compare section 10.6) asynchronous development is introduced into the Microservices themselves. Also the AJAX model from JavaScript development resembles the asynchronous treatment of messages. More and more developers are therefore familiar with the asynchronous model.

Try and Experiment

REST, SOAP/RPC and messaging each have advantages and disadvantages. Collect the advantages and disadvantages and make up your mind which of the alternatives to use.

In a Microservice-based system there can be different types of communication – however, there should be one predominant communication type. Which would you choose? Which others would be allowed in addition? In which situations?

9.5 Data Replication

At the database level Microservices could share a database and thereby concertedly access data. This type of integration has already been in practice for a long time: It is not unusual that a database is used by several applications. Often databases last longer than applications so that not the application with its demands is focused on, but rather the database. Although the integration via a shared database is widespread, it has critical disadvantages:

- The data representation cannot easily be modified since several applications access the data. A change can cause one of the applications to break. Therefore, changes have to be coordinated across all applications.
- This makes it impossible to rapidly modify applications in cases where this entails changes to the database. However, rapid changeability is exactly the area where Microservices should bring advantages.
- Finally, it is also hardly possible to clear up the schema – i.e. to remove columns which are not needed anymore because it is unclear whether any system is still using these columns. In the long run the database will get more and more complex and harder to maintain.

In the end the shared use of a database is a violation of an important architecture rule. Components should be able to change their internal data representation without other components being affected. The database schema is an example for an internal data representation. When multiple components share the database, it is not possible anymore to change the data representation. Therefore, Microservices should have a strictly separate data storage and not share a database schema.

However, a database instance can be used for multiple Microservices when the data sets of the individual Microservices are completely separate. For instance, each Microservice can use its own schema within a shared database. However, in that case there may not be any relationships between the schemas.

Replication

Replicating data is one possible alternative for the integration of Microservices. However, the data replication must not introduce a dependency of the database schemas by the back door. When the data are just replicated and the same schema is used, the same problem occurs like in the case of a shared use of the database. A schema change will also affect other Microservices so that the Microservices are in the end coupled again. This has to be avoided.

The data should be transferred into another schema to ensure the independency of the schemas and therefore the Microservices. In addition, such a transformation is also in most cases desirable for domain-based reasons.

A typical example for the use of replication in classical IT are Data Warehouses. They replicate data, but store them differently. That is due to the fact that data accessing in the Data Warehouse has very different requirements: The aim is to analyze lots of data. The data are optimized for reading access and often also combined as not every single data set is relevant for statistics.

Because of *Bounded Context* in most cases different representations or subsets of data are relevant for different Microservices. When replicating data between Microservices it will for this reason frequently anyhow be necessary to transform the data or to replicate just subsets of the data.

Problems: Redundancy and Consistency

The replication causes a redundant storage of the data. This means that the data are not immediately consistent: It takes some time until changes will have been replicated to all locations.

However, immediate consistency can be dispensable. In case of analysis tasks like in a Data Warehouse an analysis which does not comprise the orders of the last few minutes, can be sufficient. There are also other cases in which consistency is not that important. When an order takes a little bit of time until it is visible in the delivery Microservice, this can be acceptable because maybe anyhow nobody will request the data in the meanwhile.

Consistency is a requirement for the system. High consistency requirements make replication difficult. When system requirements are determined, it is often not clear how consistent the data really have to be. This limits the possibilities for data replication.

Also for replication there has to be a leading system which contains the current data. All other replicates should obtain the data from this system. Then it is always clear which data are really up-to-date. Data modifications should not be triggered by different systems. This easily causes conflicts and a very complex implementation. Such conflicts are excluded when there is just one source for changes.

Implementation

Some databases offer replication as feature. However, this is not helpful for the replication of data between Microservices because the schemas of the Microservices should be different. The replication has to be self implemented. For this purpose, a custom interface can be implemented. This interface should allow for high performance access even to large data sets. To achieve the necessary performance, one can also directly write into the target schema. The interface does not necessarily have to use a protocol like REST, but can employ faster alternative protocols. To this end it can be necessary to use another communication mechanism than the one normally used by the Microservices.

Batch

The replication can be activated in a batch. In that case the entire data or at least changes from a longer time interval can be transferred. For the first replication run the amount of data can be large so that the replication takes a long time. It can still be sensible to transfer all the data each time. This allows to correct mistakes which happened during the last replication.

An easy implementation can assign a version to each data set. Based on the version data sets which have changed can be specifically selected and replicated. This approach can be easily restarted again in case of an interruption of the replication because the process itself does not hold a state. Instead the state is stored with the data itself.

Event

One alternative is to start the replication in case of certain events. For instance, when a data set is newly generated, the data can also immediately be copied into

the replicates. Such approaches are especially easy to implement with messaging (section 9.4).

Data replication is an especially good choice for high performance access to large amounts of data. Many Microservice-based systems get along without replicating data. Even those systems which use data replication will also employ other integration mechanisms.

Try and Experiment

Would you use data replication in a Microservice-based system? In which areas? How would you implement it?

9.6 Interfaces: Internal and External

Microservice-based systems have different types of interfaces:

- Each Microservice can have one or more interfaces for other Microservices. A change to the interface can require coordination with other Microservice teams.
- The interfaces between Microservices which are developed by the same team are a special case. Team members can closely work together so that these interfaces are easier to change.
- Besides the Microservice-based system can offer interfaces to the outside with which the system can also be used outside of the organization of the developers. In extreme cases this can be potentially every internet user when the system offers a public interface in the internet.

These interfaces are differently easy to change: It is very easy to ask a colleague in the same team for a change. This colleague is presumably even in the same room.

Changes to an interface of a Microservice of another team are more difficult. The change has to prevail against other changes and new features. When the change has to be coordinated with other teams, additional expenditures arise.

Interface changes between Microservices can be safeguarded by appropriate tests (Consumer-driven Contract Tests, section 11.7). These tests examine whether the

interface still fulfills the expectations of the interface users.

External Interfaces

In case of interfaces to the outside the coordination with users is more complicated. There might be very many users. For public interfaces the users might even be unknown. Therefore, techniques like Consumer-driven Contract Tests are hard to implement in such scenarios. However, for interfaces to the outside rules can be defined which determine for instance for how long a certain version of the interface is supported. A stronger focus on backwards compatibility can also be sensible for public interfaces.

For interfaces to the outside it can be necessary to support several versions of the interface in order to not force all users to perform changes. Between Microservices it should be an aim to accept multiple versions only for uncoupling deployments. When a Microservice changes an interface, it should still support the old interface. In that case the Microservices which depend on the old interface do not have to be instantly deployed anew. However, the next deployment should use the new interface. Afterwards the old interface can be removed. This reduces the number of interfaces which have to be supported and therefore the complexity of the system.

Separating Interfaces

Since the interfaces are differently easy to change, they should be implemented separately. When an interface of a Microservice is supposed to be used externally, it can subsequently only be changed when this change is coordinated with the external users. However, a new interface for internal use can be split off. In that case the interface which is exposed to the outside is the starting point for a separate internal interface which can be more easily changed again.

Besides, several versions of the same interface can be internally implemented together. In this way new parameters of a new version can in cases of calls to the old interface simply be set to default values so that both interfaces internally use the same implementation.

Implementing External Interfaces

Microservice-based systems can also offer interfaces to the outside in different ways. Apart from a web interface for users there can also be an API, which can be accessed from outside. For the web interface section 9.1 showed already how the

Microservices can be integrated in a way which allows that all Microservices can implement a part of the UI.

When the system offers a REST interface to the outside, the calls from outside can be forwarded to a Microservice with the help of a router. In the example application the router Zuul is used for this (section 14.9). Zuul is very flexible and can forward request to different Microservices based on very detailed rules. However, HATEOAS offers also the freedom to move resources. In that case routing is dispensable. The Microservices are accessible from the outside via URLs, but they can be moved at any time. In the end the URLs are dynamically determined by HATEOAS.

It would also be possible to offer an adaptor for the external interface which modifies the external calls before they reach the Microservices. However, in that case a change to the logic cannot always be limited to a Microservice, but could also affect the adaptor.

Semantic Versioning

To denote changes to an interface a version number can be used. Semantic Versioning defines a possible version number semantics. The version number is split into MAJOR.MINOR.PATCH. The components have the following meaning:

- A change in MAJOR indicates that the new version breaks backwards compatibility. The clients have to adjust to the new version.
- The MINOR version is changed when the interface offers new features. However, the changes should be backwards compatible. A change of the clients is only necessary if they want to use the new features.
- PATCH is increased in the case of bug fixes. Such changes should be completely backwards compatible and should not require any modifications of the clients.

In case of REST one should keep in mind that it is not sensible to encode the version in the URL. The URL should represent a resource – independent of the fact with which API version it is called. Therefore, the version can for instance also be defined in an Accept Header of the request.

Postel's Law or the Robustness Principle

Another important basis for the definition of interfaces is Postel's Law, which is also known as the Robustness Principle. It states that components should be strict

in regards to what they are passing on and liberal in regards to what they are accepting from others. Differently put: Each component should adhere as closely as possible to the defined interface when using other components, but should whenever possible compensate errors which arise during the use of its own interface.

When each component behaves according to the Robustness Principle the interoperability will improve: In fact, if each component adheres exactly to the defined interfaces, interoperability should already be ensured. If a deviation happens nevertheless, the used component will try to compensate for it and thereby attempt to “save” the interoperability. This concept is also known as Tolerant Reader.

In practice a called service should accept the calls as long as this is possible at all. One way to achieve this is to only readout those parameters from a call which are really necessary. On no account should a call be rejected just because it does not formally conform with the interface specification. However, the incoming calls should be validated. Such an approach makes it easier to ensure a smooth communication in distributed systems like Microservices.

9.7 Conclusion

The integration of Microservices can occur at different levels.

Client

One possible level for the integration is the web interface (section 9.1):

- Each Microservice can bring along its own Single-Page-App (SPA). The SPAs can be developed independently. The transition between the Microservices, however, starts a completely new SPA.
- There can be one SPA for the entire system. Each Microservice supplies one module for the SPA. Therefore, the transitions between the Microservices are very simple in the SPA. However, the Microservices get very tightly integrated so that a coordination of deployments can become necessary.
- Each Microservice can bring along an HTML application. The integration can occur via links. This approach is easy to implement and allows for a modularization of the web application.
- JavaScript can load HTML. The HTML can be supplied by different Microservices so that each Microservice can contribute a representation of

its data. In this way an order can load the presentation of a product from another Microservice.
- A skeleton can assemble individual HTML snippets. Thereby an E-commerce landing page can display the last order from one Microservice and recommendations from another Microservice. ESI (Edge Side Includes) or SSI (Server Side Includes) can be useful for this.

In case of a Rich Client or a mobile app the integration is difficult because the client application is a Deployment Monolith. Therefore, changes of different Microservices can in fact only be deployed together. The teams can modify the Microservices and then deliver a certain amount of fitting UI changes together as new release of the client application. There can also be a team for each client application which adopts new functionalities of the Microservices into the client application. From an organizational perspective there can even be developers in the team of the client application which develop a custom service. This service can for instance implement the interface in a way that allows the client application to use it in a high performance manner.

Logic Layer

REST is an option for the communication of the logic layer (section 9.2). REST uses the mechanisms of the WWW to enable communication between services. HATEOAS (Hypermedia as the Engine of Application State) means that the relationships between systems are represented as links. The client knows only an entry URL. All the other URLs can be changed because they are not directly contacted by the clients, but are found by them via links starting at the entry URL. HAL defines how links can be expressed and supports the implementation of REST. Other possible data formats for REST are XML, JSON, HTML or Protocol Buffer.

Classical protocols like SOAP or RPC (section 9.3) can also be used for the communication of Microservices. SOAP offers possibilities for forwarding a message to other Microservices. Thrift has an efficient binary protocol and can likewise forward calls between processes.

Messaging (section 9.4) has the advantage that it can handle network problems and high latency times very well. In addition, transactions are also very well supported by messaging.

Data Replication

At the database level a shared schema is not recommended (section 9.5). This would couple Microservices too tightly since they would have a shared internal data representation. The data have to be replicated into another schema. The schema can be in line with the requirements for the respective Microservice. As Microservices are *Bounded Contexts*, it is very unlikely that the Microservices should use the same data model.

Interfaces and Versions

Finally, interfaces are an important foundation for communication and integration (section 9.6). Not all interfaces are equally easy to change: Public interfaces are practically not changeable at all because too many systems depend on them. Internal interfaces can more easily be changed. Public interfaces in the simplest case just route certain functionalities to suitable Microservices. Semantic Versioning is useful for giving a meaning to version numbers. To ensure a high level of compatibility the Robustness Principle is helpful.

This section should have shown that Microservices are not just services which use RESTful HTTP. This is only one option for the communication between Microservices.

Essential Points

- At the UI level the integration with HTML user interfaces is especially simple. SPAs, desktop applications or mobile apps are Deployment Monoliths so that changes to the user interface for a Microservice have to be closely coordinated with other changes.
- Though REST or RPC approaches offer at the logic level a simple programming model, messaging allows for a looser coupling and can better cope with the challenges of distributed communication via the network.
- Data replication allows high performance access even to large amounts of data. The Microservices may on no account use the same schema for their data since in that case the internal data representation cannot be changed anymore.

10 Architecture of Individual Microservices

When implementing Microservices a number of points have to be heeded. This chapter addresses first the domain architecture of Microservices (section 10.1). For implementing a Microservice-based system CQRS (section 10.2) can be interesting. This approach separates writes to data from reading data. Event Sourcing (section 10.3) places events into the center of the modeling. The structure of a Microservice can correspond to a Hexagonal Architecture (section 10.4) which subdivides functionalities into a logic kernel and adaptors. Section 10.5 focuses on resilience and stability as essential requirements for Microservices. Technical possibilities for the implementation of Microservices such as Reactive are discussed in section 10.6.

10.1 Domain Architecture

The domain architecture of a Microservice defines how the Microservice implements its domain-based functionalities. A Microservice-based architecture aims at not predetermining this decision for all Microservices. Thereby, the internal structure of Microservices can be independently decided. This allows the teams to act largely independently of each other. It is for sure sensible to adhere to established rules in order to keep the Microservice easy to understand, simple to maintain and also replaceable. However, there is no strict need for regulations at this level.

This section shows how to identify potential problems with the domain architecture of a Microservice. Whether there really is a problem and how it can be solved, then has to be answered by the responsible team.

Cohesion

The domain architecture of the overall system influences the domain architecture of the individual Microservices. As presented in section 8.1, Microservices should be loosely coupled to each other. Besides, the Microservices should have a high internal cohesion. A Microservice should have only one responsibility in regards to the domain. Consequently, the parts of a Microservice have to be loosely coupled, and the Microservice has to have a high cohesion. If that is not the case, the Microservice will likely have more than one responsibility. If the

cohesion within the Microservice is not high enough, the Microservice can be split into several Microservices. Due to the split the Microservices remain small and thus easier to understand, to maintain and to replace.

Encapsulation

Encapsulation means that a part of the architecture hides internal information from the outside – especially all internal data structures. Instead, the access is supposed to occur via an interface. Thereby the software remains easy to modify: Internal structures can be changed without influencing other parts of the system. For this reason, Microservices may in no case allow other Microservices access to their internal data structures. Otherwise these data structures cannot be modified anymore. Besides, in this manner, every Microservice needs only to understand the interface of another Microservice. This improves the structure and intelligibility of the system.

Domain-Driven Design

Domain-driven Design (DDD) is a possibility to internally structure Microservices. Each Microservice can have a DDD domain model. The necessary patterns from Domain-Driven Design were already introduced in section 4.3. Especially when Domain-driven Design and Strategic Design define the structure of the overall system (section 8.1), the Microservices should also use these approaches. During the development of the overall system Strategic Design orientates itself to the fact which domain models there are and how they are distributed across the Microservices.

Transactions

Transactions bundle multiple actions so that they can only be executed together or not at all. A transaction can hardly comprise more than one Microservice. Only messaging is able to support transactions across Microservices (compare section 9.4). The domain-based design within a Microservice ensures that each operation at the interface corresponds to one transaction. In this way it can be avoided that multiple Microservices have to participate in one transaction. This would be very hard to implement technically.

10.2 CQRS

Systems usually save a state. Operations can change data or read them. These two types of operations can be separated: Operations that change data and therefore have side effects (commands) can be distinguished from operations that just read

data (queries). An operation may not simultaneously change the state and return data. This distinction makes the system easier to understand: When an operation returns a value, it is a query and does not change any values. This entails additional advantages. Queries can for example be provided with a cache. If read operations changed also data, the addition of a cache would not be so easy since operations with side effects still have to be executed in spite of a cache. The separation between queries and commands is called CQS (Command Query Separation). This principle is not limited to Microservices, but can be applied in general. For example, classes in an object-oriented system can divide operations in the same manner.

CQRS

CQRS (Command Query Responsibility Segregation) is more drastic than CQS and completely separates the processing of queries and commands.

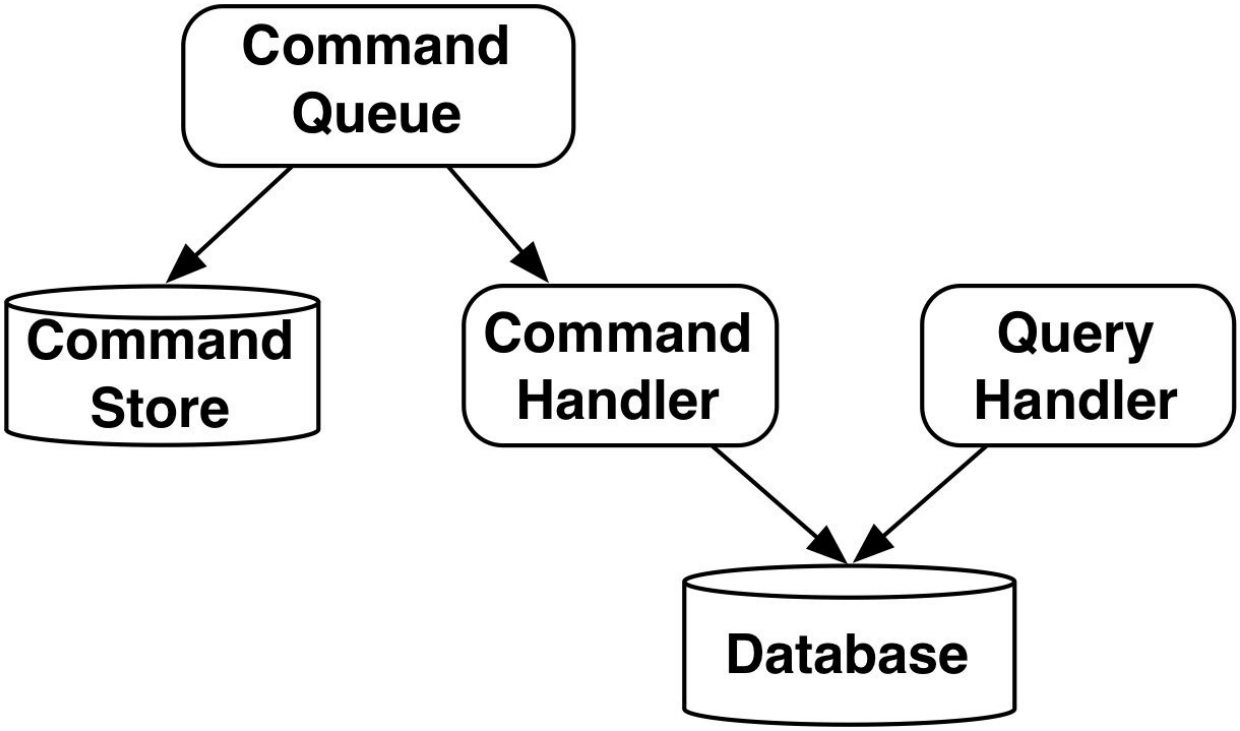


Fig. 45: Overview of CQRS

Fig. 45 shows the structure of a CQRS system. Each command is stored in the Command Store. In addition, there can be Command Handlers. The Command Handler in the example uses the commands for storing the current state of the data in a database. A Query Handler uses this database to process queries. The database can be adjusted to the needs of the Query Handler. For example, a database for the analysis of order processes can look completely different from a

database which customers use for displaying their own order processes. Entirely different technologies can be employed for the query database. It is for instance possible to use an In-Memory-Cache which loses the data in case of a server failure. The information persistency is ensured by the Command Store. In an emergency the content of the cache can be reconstructed by the Command Store.

Microservices and CQRS

CQRS can be implemented with Microservices:

- The communication infrastructure can implement the Command Queue when a messaging solution is used. In case of approaches like REST a Microservice has to forward the commands to all interested Command Handlers and implement the Command Queue that way.
- Each Command Handler can be a separate Microservice. It can handle the commands with its own logic. Thereby logic can very easily be distributed to multiple Microservices.
- Likewise, a Query Handler can be a separate Microservice. The changes to the data which the Query Handler uses can be introduced by a Command Handler in the same Microservice. However, the Command Handler can also be a separate Microservice. In that case the Query Handler has to offer a suitable interface for accessing the database so that the Command Handler can change the data.

Advantages

CQRS has a number of advantages especially in the interplay with Microservices:

- Reading and writing of data can be separated into individual Microservices. This allows for even smaller Microservices. When the writing and reading is that complex that a single Microservice for both would get too large and too hard to understand, a split might be very sensible.
- Likewise, another model can be used for writing and reading. Microservices can each represent a *Bounded Context* and therefore use different data models. For instance, in an E-commerce shop a lot of data can be written for an online purchase while statistical evaluations read only few data for each purchase. From a technical perspective the data can be optimized for reading operations via denormalization or via other means for certain queries.
- Writing and reading can be scaled differently by starting different numbers of Query Handler Microservices and Command Handler Microservices. This supports the fine granular scalability of Microservices.

- The Command Queue facilitates the handling of load peaks during writing. The queue buffers the changes which are then processed later on. However, in that case a change to the data will not be immediately taken into consideration by the queries.
- It is easy to run different versions of the Command Handlers in parallel. This facilitates the deployment of Microservices in new versions.

CQRS can serve to make Microservices even smaller, even when operations and data are really very closely connected. Each Microservice can independently decide for or against CQRS. There are different ways to implement an interface which offers operations for changing and reading data. CQRS is only one option. Both aspects can also be implemented without CQRS in just one Microservice. The freedom to be able to use different approaches is one of the main advantages of Microservice-based architectures.

Challenges

CQRS causes also some challenges:

- Transactions which comprise read and write operations are hard to implement. The respective operations can be implemented in different Microservices. In that case it is hardly possible to combine the operations into one transaction since transactions across Microservices are usually impossible.
- It is hard to ensure data consistency across different systems. The processing of events is asynchronous so that different nodes can finish processing at different points in time.
- The expenditure for development and infrastructure is higher. More system components and more complex communication technologies are required.

It is not sensible to implement each Microservice with CQRS. However, the approach represents in many circumstances a good supplement for Microservice-based architectures.

10.3 Event Sourcing

Event Sourcing has a similar approach like CQRS. However, the events from Event Sourcing differ from the commands from CQRS. Commands are specific: They exactly define what is to be changed in an object. Events contain information about something that has happened. Both approaches can also be combined: A

command can change data. This will result in events to which other components of the system can react.

Instead of the state itself Event Sourcing stores events which have lead to the current state. While the state itself is not saved, it can be reconstructed from the events.

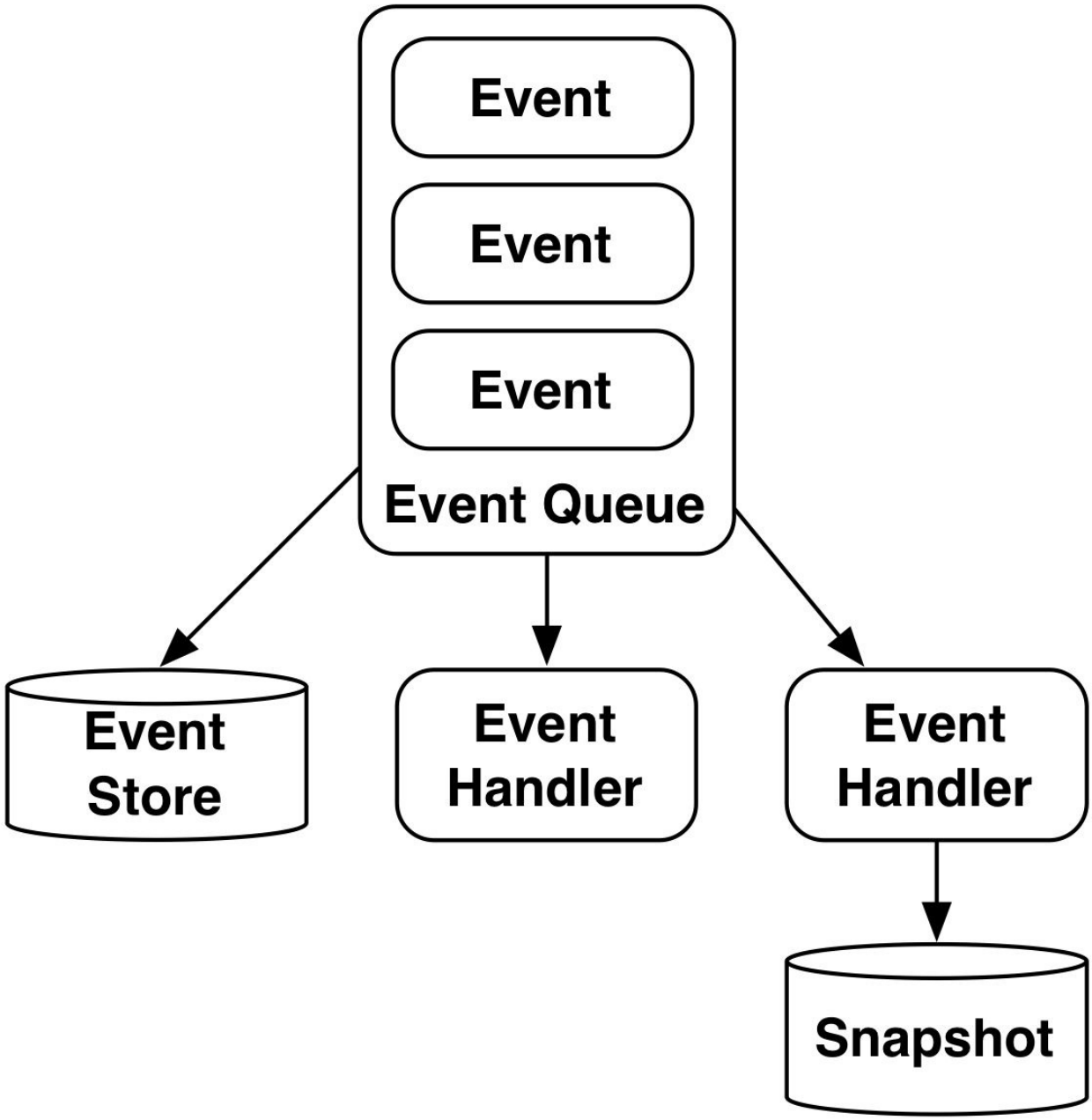


Fig. 46: Overview of Event Sourcing

Fig. 46 gives an overview of Event Sourcing:

- The **Event Queue** sends all events to the different recipients. It can for instance be implemented with messaging middleware.
- The **Event Store** saves all events. Therefore, it is always possible to reconstruct the chain of events and the events themselves.
- An **Event Handler** reacts to the events. It can contain business logic which reacts to events.

- In such a system it is only the events which are easy to trace. The current state of the system is not easy to follow up on. Therefore, it can be sensible to maintain a **Snapshot** which contains the current state. At each event or after a certain time the data in the Snapshot will be changed in line with the new events. The Snapshot is optional. It is also possible to ad hoc reconstruct the state from the events.

Events may not be changed afterwards. Erroneous events have to be corrected by new events.

Event Sourcing is based on Domain-Driven Design (compare section 4.3). Therefore, in line with *Ubiquitous Language*, the events should have names which are also sensible in the business context. In some domains an event-based model is especially sensible from a domain perspective. For instance, bookings to an account can be considered as events. Requirements like auditing are very easy to implement with Event Sourcing: Since the booking is modeled as an event, it is very easy to trace who has performed which booking. In addition, it is relatively easy to reconstruct a historical state of the system and old versions of the data. Event Sourcing can be a good option from a domain perspective. Generally, approaches like Event Sourcing are sensible in complex domains which also profit from Domain-driven Design.

Event Sourcing has similar advantages and disadvantages like CQRS, and both approaches can easily be combined. Event Sourcing is especially sensible when the overall system works with an Event-driven Architecture (section 8.6). In that case the Microservices anyhow send already events concerning changes of the state and it is sensible to use this approach also in the Microservices.

Try and Experiment

Choose a project you know.

In which places would Event Sourcing be sensible? Why? Would Event Sourcing be useable in an isolated manner at some places or would the entire system have to be changed to Events?

Where could CQRS be helpful? Why?

Do the interfaces adhere to the CQR rule? In that case the read and write operations would have to be separate in all interfaces.

10.4 Hexagonal Architecture

A Hexagonal Architecture focuses on the logic of the application (Fig. 47). The logic contains only the business functionalities. It has different interfaces which are each represented by an edge of the hexagon. In the example these are the interface for the interaction with users and the interface for administrators. Users can utilize these interfaces via a web interface which is implemented by HTTP adaptors. For tests there are special adaptors. They enable the tests to simulate users. Finally, there is an adaptor which makes the logic also accessible via REST. This allows other Microservices to call the logic.

Interfaces do not only take requests from other systems. In addition, also other systems are contacted via such interfaces: the database via the DB adaptor which in fact uses a database. The alternative is an adaptor for test data. Finally, another application can be contacted via a REST adaptor. Instead of these adaptors a test system can be used which simulates the used system.

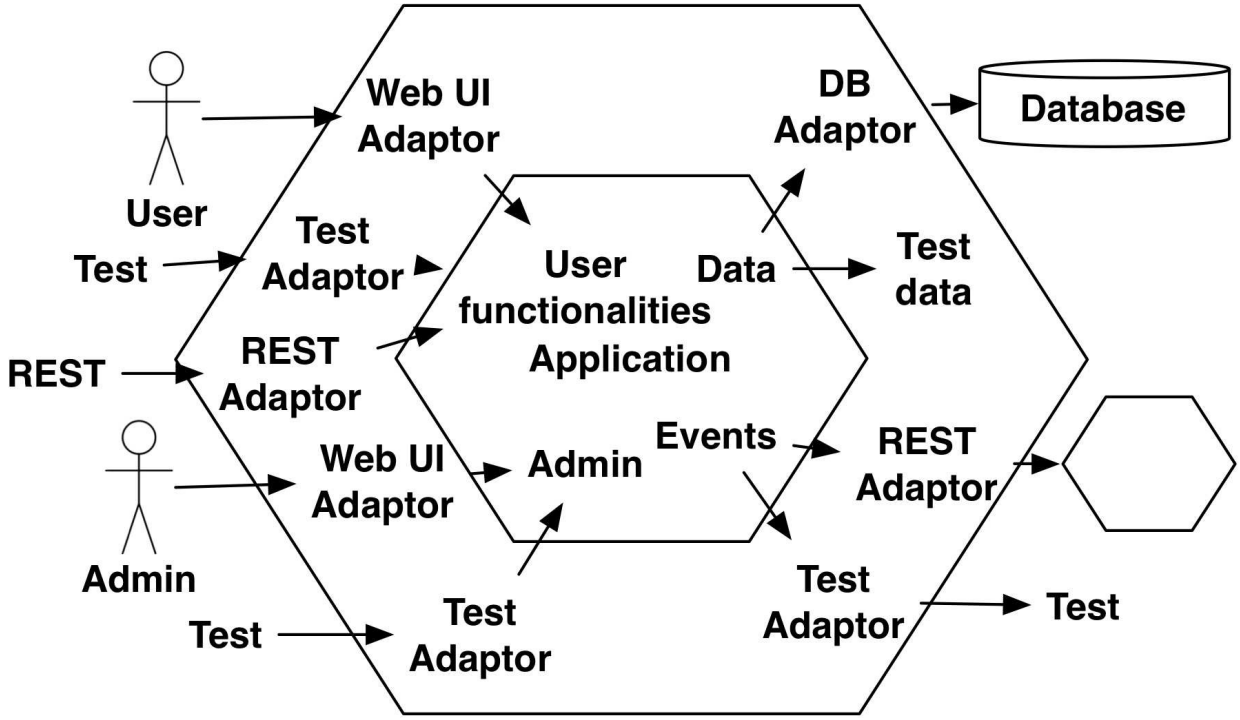


Fig. 47: Overview of Hexagonal Architecture

Another name for Hexagonal Architectures is “Ports and Adaptors”. Each facet of the application like user, admin, data or event is a port. The adaptors implement the ports based on technologies like REST or web user interfaces. Via the ports on the right side of the hexagon the application fetches data, while via the ports on the left side its functionalities and data for user and other systems are offered.

The Hexagonal Architecture divides a system into a logic kernel and adaptors. Only the adaptors enable the communication to the outside.

Hexagons or Layers?

A Hexagonal Architecture is an alternative to a layered architecture. In a layered architecture there is a layer in which the UI is implemented and a layer in which the persistence is implemented. In a Hexagonal Architecture there are adaptors which are connected to the logic via ports. A Hexagonal Architecture clearly shows that there can be more ports than just persistence and UI. Besides the term “adaptor” illustrates that the logic and the ports are supposed to be separate from the concrete protocols and implementations of the adaptors.

Hexagonal Architectures and Microservices

It is very natural for Hexagonal Architectures to offer logic not only for other Microservices via a REST interface, but also for users via a web UI. Exactly this

idea is also the basis of Microservices. They are not only supposed to provide logic for other Microservices, but should also support the direct interaction of users via a UI.

Since individual test implementations can be implemented for all ports, the isolated testing of a Microservice is easier with a Hexagonal Architecture. For this purpose, test adaptors just have to be used instead of the actual implementation. Especially the independent testing of individual Microservices is an important prerequisite for the independent implementation and the independent deployment of Microservices.

The necessary logic for resilience and stability (compare section 10.5) or Load Balancing (section 8.10) can also be implemented in the adaptor.

It is likewise imaginable to distribute the adaptors and the actual logic into individual Microservices. This will result in more distributed communication and therefore into an overhead. However, on the other hand the implementation of adaptor and kernel can be distributed to different teams. For instance, one team which develops a mobile client can implement a specific adaptor which is adapted to the bandwidth restrictions of mobile applications (compare also section 9.1).

An Example

A Microservice for orders shall serve as example for a Hexagonal Architecture (Fig. 48). The user can utilize the functionalities of the Microservice via the web UI to place orders. Likewise there is a REST interface with which other Microservices or external clients can use the “user functionalities”. The web UI, the REST interface and the test adaptor are three adaptors for the “user functionalities” of the Microservice. The implementation with three adaptors emphasizes that REST and web UI are just two options to use the same functionalities. Besides, in this manner Microservices are implemented which integrate UI and REST. Technically the adaptors can still be implemented in separate Microservices.

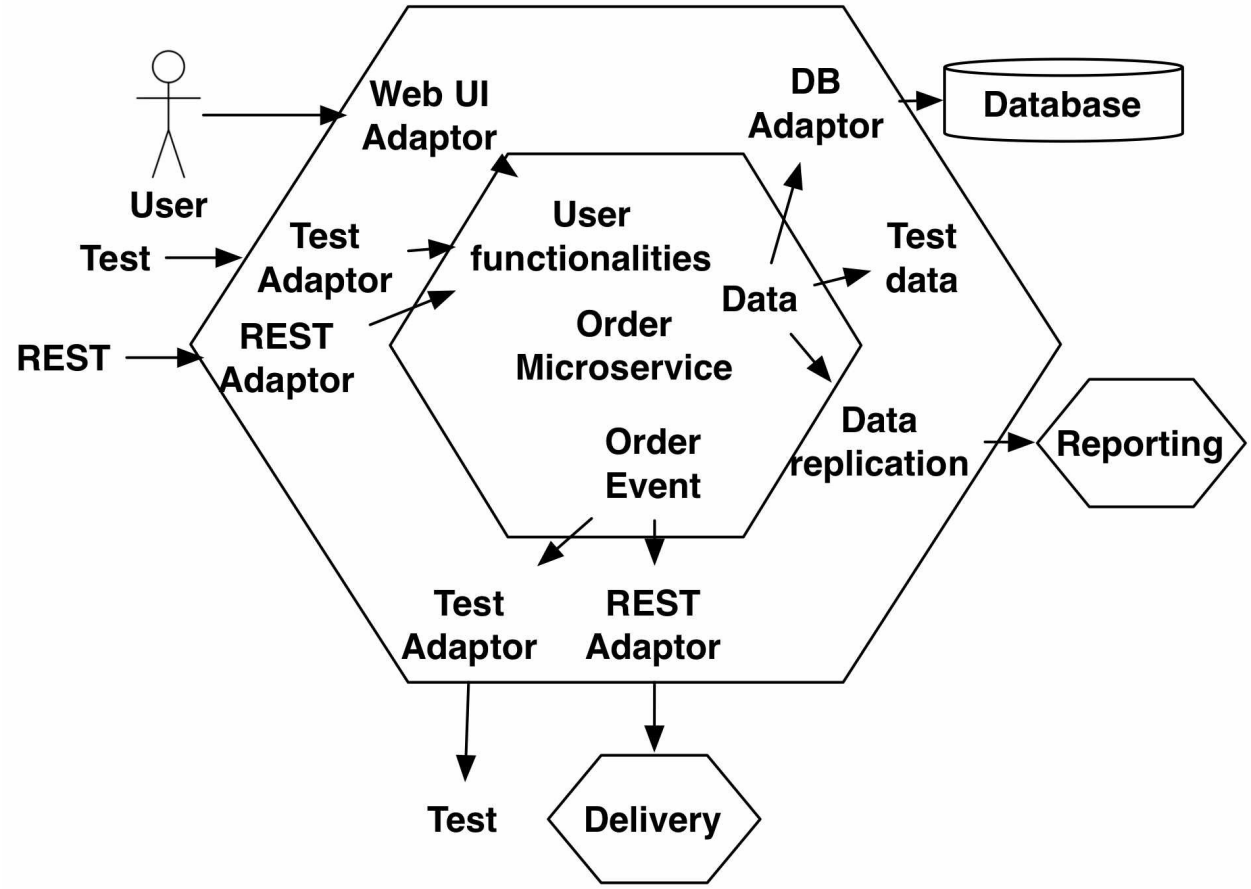


Fig. 48: The order Microservice as an example for Hexagonal Architecture

Another interface are the order events. They announce to the Microservice "Delivery" when new orders have arrived so that the orders can be delivered. Via this interface the Microservice "Delivery" communicates also when an order has been delivered or when delays have occurred. In addition, this interface can be served by an adaptor for tests. Therefore, the interface to the Microservice "Delivery" does not just simply write data, but can also introduce changes to the orders. This means that the interface uses other Microservices, but does also itself take changes.

The Hexagonal Architecture has a domain-based distribution into an interface for user functionalities and an interface for order events. Thereby the architecture underlines the domain-based design.

The state of the orders is saved in a database. Also in this case there is an interface where test data can be used for tests instead of the database. This interface corresponds to the persistence layer of a classical architecture.

Finally, there is an interface which via data replication transmits the information regarding the order to reporting. There statistics can be generated from the orders. Reporting appears to be a persistence interface, but is really more: The data are not just stored, but changed to enable quick generation of statistics.

As the example shows, a Hexagonal Architecture creates a good domain-based distribution into different domain-based interfaces. Each domain-based interface and each adaptor can be implemented as a separate Microservice. This allows to divide the application into numerous Microservices, if necessary.

Try and Experiment

Choose a project you know.

Which individual hexagons would there be?

Which ports and adaptors would the hexagons have?

Which advantages would a Hexagonal Architecture offer?

What would the implementation look like?

10.5 Resilience and Stability

The failure of a Microservice should affect the availability of other Microservices as little as possible. As a Microservice-based system is a distributed system, the danger of a failure is fundamentally higher: Network and servers are unreliable. As Microservices are distributed on multiple servers, the number of servers is higher per system and therefore also the probability of a failure. When the failure

of one Microservice can result in the failure of additional Microservices, step by step the entire system can break down. This has to be avoided.

For this reason, Microservices have to be shielded from the failure of other Microservices. This property is called resilience. The necessary measures to achieve resilience have to be part of the Microservice. Stability is a broader term which denotes a high software availability. “Release It!” [1] lists several patterns to this topic:

Timeout

Timeouts help to detect unavailability when communicating with another system. If no response has been returned after the timeout, the system is considered unavailable. Unfortunately, many APIs do not have the possibility to define timeouts, and some default timeouts are very high. At the level of the operating system default TCP timeouts can be e.g. five minutes. During this time the Microservice does not respond to callers since the service is waiting for the other Microservice. Therefore, also this Microservice seems to have failed. Besides the request can block a thread during this time. At some point all threads are blocked, and the Microservice cannot receive any additional requests anymore. Exactly such a domino effect has to be avoided. When the API intends a timeout for accessing another system or a database, this timeout should be set. An alternative option is to let all requests to external systems or databases take place in a extra thread and to terminate this thread after a timeout.

Circuit Breaker

A Circuit Breaker is a safety measure in an electricity circuit. In case of a short circuit the Circuit Breaker interrupts the flow of electricity to avoid dangerous consequences like overheating or fire. This idea can be applied to software as well: When another system is not available anymore or returns only errors, a Circuit Breaker prevents calling the system. Calls are anyhow meaningless in this scenario.

Normally, the Circuit Breaker is closed, and calls are forwarded to the other system. When an error occurs, depending on the error frequency the Circuit Breaker will be opened. In that case calls are not send on to the other system, but run directly into an error.

This takes load off the other system. Also there is no need for a timeout as the error is instantaneous. After some time the Circuit Breaker will close again.

Incoming calls will now be forwarded again to the other system. If the error persists, the Circuit Breaker will open again.

The Circuit Breaker can be combined with a timeout. A timeout can open the Circuit Breaker. The state of the Circuit Breakers shows operations where currently problems in the system are. An open Circuit Breaker indicates that a Microservice is not able to communicate with another Microservice anymore. Therefore, the state of the Circuit Breakers should be displayed in monitoring for operations.

When the Circuit Breaker is open, an error does not necessarily have to be generated. It is also possible to just degrade the functionality. Let us assume that a Automated Teller Machine (ATM) cannot verify whether an account contains enough money for the desired withdrawal, because the responsible system is not reachable. Nevertheless, cash withdrawals can be permitted up to a certain limit so that customers will not be dissatisfied. In addition, the bank will make less profit if all cash withdrawals are prohibited as it will not get the withdrawal-associated fees. Whether and up to which limit a cash withdrawal is still permitted is a business decision. The possible damage has to be balanced against the potential for profit. There can also be other rules to be applied in case of the failure of another system. Calls can for instance be answered from a cache. More important than the technical possibilities is the domain-based requirement for deciding on the appropriate handling of a system failure.

Bulkhead

A Bulkhead is a special door on a ship which can be closed in a watertight manner. It divides the ship into several areas. When water gets in, only a part of the ship is affected, and thus the ship will not sink.

Similar approaches are applicable to software: The entire system has to be divided into individual areas. A breakdown or a problem in one area may not affect the other areas. For example, there can be several instances of a Microservice for different clients. When a client overloads the Microservices, the other clients will not be negatively affected. The same is true for resources like database connections or threads. When different parts of a Microservice use different pools for these resources, one part cannot block the other parts even if it uses up all its resources.

In Microservices-based architectures the Microservices themselves form separate areas. This is especially the case when each Microservice brings its own virtual machine along. Even if the Microservice causes the entire virtual machine to crash or overloads it, the other Microservices will hardly be affected. They run on different virtual machines and are therefore separate.

Steady State

The term Steady State stands for the fact that systems should be built in a manner that allows for their permanent operation. This means for instance that systems should not store increasing amounts of data. Otherwise the system will have used up its entire capacity at some point and therefore breakdown. Log files for example have to be deleted at some point. Usually they are anyhow only interesting during a certain time interval. Another example is caching: When a cache always keeps growing, it will at some point have filled all storage space. Therefore values also have to be deleted again from cache at some point to keep the cache from permanently growing.

Fail Fast

Timeouts are only necessary because another system requires a long time to respond. The idea behind Fail Fast is to address the problem from the other side: Each system is supposed to recognize errors as fast as possible and to indicate them immediately. When a call requires a certain service and this service is unavailable for the moment, the call can be directly answered with an error message. The same is true when other resources are not available at the time. Moreover, the call can be validated right at the start. When it contains errors, there is anyhow nothing gained by processing it. Therefore, an error message can be returned immediately. The advantages of Fail Fast are identical with the ones offered by timeout: A rapid failure uses up less resources and therefore results in a more stabile system.

Handshaking

Handshaking in a protocol serves to initiate communication. Thereby protocols allow that a server rejects additional calls in cases of overload. This avoids additional overload, a breakdown or too slow responses. Unfortunately, protocols like HTTP do not support this. Therefore, the application has to mimic the functionality for instance with Health Checks. An application can signal in that way that it is principally reachable, but has right now so much load that it is not sensible to send more calls to it. Protocols which build on socket connections can implement such approaches by themselves.

Test Harness

A Test Harness can be used to find out how an application behaves in certain error situations. Among those can be problems at the level of the TCP/IP or for instance responses of other systems which contain HTTP header, but no HTTP body. Something like that should in fact not occur since operating system or network stack should deal with it. Nevertheless, such errors can occur in practice and can have dramatic consequences since applications are not at all prepared for handling them. A Test Harness can be an extension of the tests which are discussed in section 11.8.

Uncoupling via Middleware

Calls in one program only function on the same host at the same time in the same process. Synchronous distributed communication (e.g. REST) allows for communication between different hosts and different processes at the same time. Asynchronous communication via messaging systems (section 9.4) also allows an uncoupling over time. A system should not wait for a response of an asynchronous process. The system should continue working on other tasks instead of just waiting for a response. Errors which cause one system after another to break down like domino stones are much less likely in case of asynchronous communication. The systems are forced to deal with long response times since asynchronous communication anyhow can result in long response times.

Stability and Microservices

Stability patterns like Bulkhead restrict failures to a unit. Microservices are the obvious choice for a unit. They run on separate virtual machines and accordingly are already isolated in regards to most issues. Thereby the Bulkhead pattern arises very naturally in a Microservices-based architecture. Fig. 49 shows an overview: A Microservice can via Bulkhead, Circuit Breaker and timeouts safeguard the use of other Microservices. The used Microservice can additionally implement Fail Fast. The safeguarding can be implemented via patterns in those parts of a Microservice which are responsible for communicating with other Microservices. Thereby this aspect is implemented in one area of the code and not distributed across the entire code.

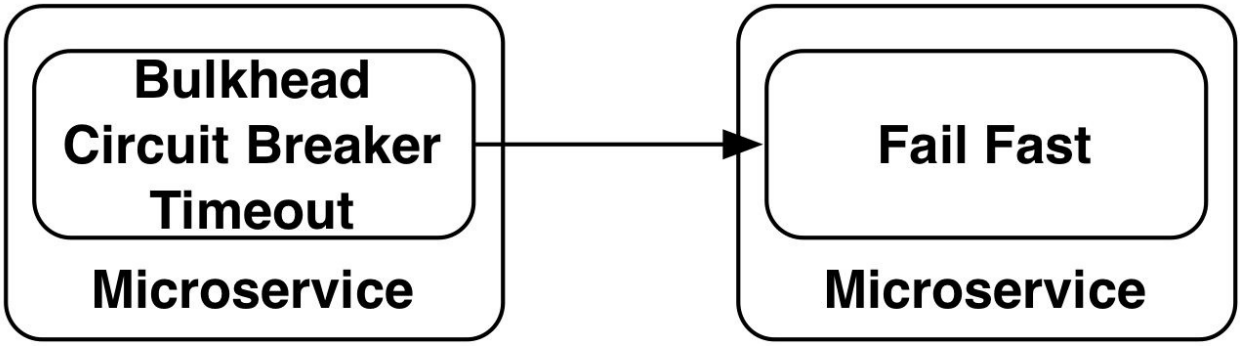


Fig. 49: Stability in the case of Microservices

On a technical level the patterns can be implemented differently. For Microservices there are the following options:

- Timeouts are easy to implement: For accessing the other system an individual thread is started which is terminated after a timeout.
- At the first glance Circuit Breakers are not very complex and can be developed in your own code. However, the implementation has also to work under high load and has to offer an interface for operations to allow monitoring. This is not trivial. Therefore a home-grown implementation is not very sensible.
- Bulkheads are brought along by Microservices since a problem is in many cases already limited to just one Microservice. For instance, a memory leak will only cause one Microservice to fail.
- Steady State, Fail Fast, Handshaking and Test Harness have to be implemented by each Microservice.
- Uncoupling via Middleware is an option for the shared communication of Microservices.

Resilience and Reactive

The Reactive Manifesto lists Resilience as essential property of a Reactive application. Resilience can be implemented in an application by processing calls asynchronously. Each part of an application which processes messages (“actor”) has to be monitored. When an actor does not react anymore, it can be restarted. This allows to handle errors and to make applications more resilient.

Hystrix

Hystrix implements timeout and Circuit Breaker. For this purpose, developers have to encapsulate calls in commands. Alternatively, Java annotations can be used. The calls take place in individual thread pools. Several thread pools can be

created. If there is one thread pool per called Microservice, the calls of the Microservices can be separated from each other in such a manner that a problem with one Microservice does not affect the use of the other Microservices. This is in line with the Bulkhead idea. Hystrix is a Java library which is under Apache license and originates from the Netflix stack. The example application uses Hystrix together with Spring Cloud (compare section 14.10). In combination with a Sidecar Hystrix can also be used for applications which are not written in Java (compare section 8.7). Hystrix supplies information about the state of the thread pools and the Circuit Breaker for monitoring and operation. This information can be displayed in a special monitoring tool – the Hystrix dashboard. Internally Hystrix uses the Reactive Extensions for Java (RxJava). Hystrix is the most widely used library in the area of Resilience.

Try and Experiment

This chapter introduced eight patterns for stability. Prioritize these patterns. Which properties are indispensable? Which are important? Which are unimportant?

How can be verified whether the Microservices really implement the patterns?

10.6 Technical Architecture

The technical architecture of a Microservice can be individually designed. Frameworks or programming languages do not have to be uniform for all Microservices. Therefore, each Microservice can well use different platforms. However, certain technical infrastructures fit better to Microservices than others.

Process Engines

Process engines which normally serve to orchestrate services in a SOA (section 7.1) can be used in a Microservice to model a business process. The important point is that one Microservice implements only one domain – for instance one *Bounded Context*. A Microservice should not end up as a pure integration or orchestration of other Microservices without its own logic. Otherwise changes will require that not only this one Microservice is modified, but also the integrated Microservices. However, it is a central aim of Microservice-based

architectures to limit changes to one Microservice if possible. If multiple business processes have to be implemented, different Microservices should be used for it. Each of these Microservices should implement one business process together with the dependent services. Of course, it will not always be possible to avoid that other Microservices have to be integrated to implement a business process. However, a Microservice which just represents an integration is not sensible.

Statelessness

Stateless Microservices are very advantageous. To put it more clearly: Microservices should not save any state in their logic layer. States in the database or on the client are acceptable. When using this approach the failure of an individual instance does not have a big impact. The instance can just be replaced by a new instance. In addition, the load can be distributed between multiple instances – without having to take into consideration which instance processed the previous calls of the user. And finally, the deployment of a new version is easier since the old version can just be stopped and replaced without having to migrate its state.

Reactive

Implementing Microservices with Reactive technologies can be especially useful. These approaches are comparable to Erlang (compare section 15.7): Applications consist of actors. In Erlang they are called processes. Work in each actor is sequential, however, different actors can work in parallel on different messages. This enables the parallel processing of tasks. Actors can send messages to other actors which end up in the mailboxes of these actors. I/O operations are not blocking in Reactive applications: A request for data is sent out. When the data are there, the actor is called and can process the data. In the meantime the actors can work on other requests.

Essential properties are according to the Reactive Manifesto:

- **Responsive:** The system should react to requests as fast as possible. This has among others advantages for Fail Fast and therefore for stability (compare section 10.5). Once the mailbox is filled to a certain predetermined degree, the actor can for instance reject to accept additional messages. Thereby the sender is slowed down, and the system as such does not get overloaded. Other requests can still be processed. The aim to be responsive is also supported by the abdication of blocking I/O operations.

- **Resilience** and its relationship with Reactive applications has already been discussed in section 10.5.
- **Elastic** means that new systems can be started at run time which share the load. For that purpose the system has to be scalable, and at the same time it has to be possible to change the system at run time in such a way that the load can be distributed to the different nodes.
- **Message Driven** means that the individual components communicate with each other via messaging. As described in section 9.4, this communication fits well to Microservices. Reactive applications use very similar approaches also within the application itself.

Reactive can implement Microservices especially easily since the ideas from the Reactive area fit very well to Microservices. However, similarly good results can also be achieved by the use of classical technologies.

Technologies from the area of Reactive are for instance:

- The programming language Scala with the Reactive framework Akka and web framework Play which is based on it. These frameworks can also be used with Java.
- There are Reactive extensions for practically all popular programming languages. Among those are RxJava for Java or RxJS for JavaScript.
- Similar approaches are also supported by Vert.x (compare also section 15.6). Even though this framework is based on the JVM, it supports many different programming languages like Java, Groovy, Scala, JavaScript, Clojure, Ruby or Python.

Microservices without Reactive?

Reactive is only one option for implementing a system with Microservices. The classical programming model with blocking I/O, without actors and with synchronous calls is likewise suitable for this type of system. As previously discussed, Resilience can be implemented via libraries. Elastic can be achieved by starting new instances of the Microservices for instance as virtual machines or Docker containers. And classical applications can also communicate with each other via messages. Reactive applications have advantages for Responsive. However, in that case it has to be ensured that operations really do not block. For I/O operations the Reactive solutions can usually guarantee that. However, a complex calculation can block the system. So in that case no messages can be processed anymore, and the entire system is blocked. A Microservice does not

have to be implemented with Reactive technologies, but they are for sure an interesting alternative.

Try and Experiment

Get more information about Reactive and Microservices.

How exactly are the advantages implemented?

Is there a Reactive extension for your preferred programming language? Which features does it offer? How does this help with implementing Microservices?

10.7 Conclusion

The team implementing a certain Microservice is also responsible for its domain-based architecture. There should be few guidelines restricting team decisions so that the independence of the teams is ensured.

Low cohesion can be an indication for a problem with the domain-based design of a Microservice. Domain-driven Design (DDD) is an interesting option for structuring a Microservice. Likewise transactions can provide clues for a sensible domain-based division: An operation of a Microservice should be a transaction (section 10.1).

CQS (Command Query Separation) divides operations of a Microservice or a class into read operations (queries) and write operations (commands). CQRS (Command Query Responsibility Segregation) (section 10.2) separates data changes via commands from Query Handlers which can process requests. Thereby Microservices or classes are created which can only implement reading or writing access. Event Sourcing (Section 10.3) stores events and thereby does not focus on the current state, but on the history of all events. These approaches are useful for building up Microservices because they allow for the creation of smaller Microservices which can implement only read or write operations. This enables an independent scaling and optimizations for both types of operations.

Hexagonal Architecture (section 10.4) focuses on a kernel which can be called via adaptors for instance by a UI or an API, as the center point of each Microservice. Likewise adaptors can enable the use of other Microservices or of databases. For Microservices this results in an architecture which supports a UI and a REST interface in a Microservice.

Section 10.5 has presented some patterns for Resilience and Stability. The most important of those are Circuit Breaker, Timeout and Bulkhead. A popular implementation is Hystrix.

Section 10.6 introduced certain technical options for Microservices: The use of Process Engines is for instance an option for a Microservice. Statelessness is advantageous. And finally, Reactive approaches are a good basis for the implementation of Microservices.

In summary, the chapter explained essential factors for the implementation of individual Microservices.

Essential Points

- Microservices within a Microservice-based system can have different domain-based architectures.
- Microservices can internally be implemented with Event Sourcing, CQRS or Hexagonal Architectures.
- Technical properties like stability can only be implemented individually by each Microservice.

1. Michael T. Nygard: Release It!: Design and Deploy Production-Ready Software, Pragmatic Programmers, 2007, ISBN 978-0-97873-921-8↩

11 Testing Microservices and Microservice-based Systems

The separation of a system into Microservices has consequences for testing. Section 11.1 explains the motivation behind software tests. Section 11.2 discusses fundamental approaches for tests, not only in regards to Microservices. Section 11.3 illustrates why there are special challenges when testing Microservices which are not present in this form in other systems. One example: In a Microservice-based system the entire system comprising all Microservices has to be tested (section 11.4). This is laborious since there can be a multitude of Microservices. Section 11.5 describes the special case of a legacy application which is supposed to be replaced by Microservices. In that case the integration of Microservices and legacy application has to be tested. Testing just the Microservices is not sufficient. Another possibility to safeguard the interfaces between Microservices are consumer-driven contract tests (section 11.7). They reduce the expenditure for testing the entire system. Of course, the individual Microservices have to be tested as well. In this context the question arises how individual Microservices can at all be run in isolation without other Microservices (section 11.6). Microservices provide technology freedom, nevertheless there have to be certain standards. Therefore tests can comprise technical standards (section 11.8) which have been defined in the architecture.

11.1 Why Tests?

Testing software is an essential part of every software development project. Nevertheless, questions about the goal of the testing are hardly asked. In the end tests are risk management. They are supposed to minimize the risk that errors appear in production and are noticed by users – or that other damage is done.

This answer entails a number of consequences:

- Each test has to be evaluated based on the question which risk it minimizes. In the end a test is only meaningful when it helps to avoid concrete error scenarios which otherwise would occur in production.

- Tests represent only one option to deal with risk. Consequences of an error occurring in production can also be minimized in different ways. An important point is how long it will take until a certain error is corrected in production. The longer an error persists in production, the more profound are usually the consequences. How long it takes to put a corrected version of the services into production depends on the deployment approach. Therefore, there is a connection between tests and deployment strategies.
- Likewise, it is a very important aspect how long it will take until an error in production is noticed. This depends on the quality of monitoring and logging.

In the end many measures can address errors in production. Just focusing on tests is not sufficient in order to be able to offer high quality software to customers.

Tests Minimize Expenditure

Tests can do more than just minimize risk. They can help to minimize or avoid expenditure. An error in production can generate a high expenditure. The error can influence the customer service and can cause extra expenditure there. Identifying and correcting errors in production is usually more laborious than during tests. Access to the systems is often restricted. Besides the developers will have implemented other features meanwhile so that they will first have to familiarize themselves again with the erroneous code.

In addition, the approach for tests can help to avoid or reduce expenditure. Automating tests only appears laborious at first glance. When tests are so well defined that results are reproducible, the step to a complete formalization and automation is not huge. In that case the costs for the execution of the tests will be negligible. This allows to test more frequently, and this will promote quality.

Test = Documentation

A test defines what the code is supposed to do. Thereby it represents a kind of documentation. Unit tests define how the productive code is supposed to be used and also how it is supposed to behave in exceptional and borderline cases. Acceptance tests reflect the requirements of the customers. The advantage of tests in comparison to documentation is that they are executed. This ensures that the tests really reflect the current behavior and not an outdated state or a state which will only be reached in the future.

Test-driven Development

Test-driven development exploits the fact that tests represent requirements: In this approach developers initially write tests and subsequently the implementation. This ensures that the entire code is safeguarded by tests. Besides, in that case tests are not influenced by knowledge about the code since the code does not even exist yet when the test is written. If tests are only implemented afterwards, developers might not test for certain potential problems due to their knowledge about the implementation. In case of test-driven development this is very unlikely. Thereby tests turn into a very important basis for the development process. They push the development: Prior to each change there has to be a test which does not work. Code may only be adjusted when the test was successful. This is true at the level of individual classes, which are safeguarded by previously written unit tests, but also at the level of requirements which are ensured by previously written acceptance tests.

11.2 How to Test?

There are different types of tests which handle different risks:

Unit Tests

Unit tests examine the units the system consists of - just like their name suggests. They minimize the risk that the individual units contain errors. Unit tests check especially small units – individual methods or functions. For this purpose, all dependencies have to replaced because otherwise not only the individual unit but also the dependent units are tested. To replace the dependencies there are two possibilities:

- **Mocks** simulate a certain call with a certain result. After the call the test can verify whether the expected calls really have taken place. A test can for instance define a Mock which will return a defined customer for a certain customer number. After the test it can evaluate whether the customer has really been readout by the code. In another test scenario the Mock can simulate an error if asked for a customer. Thereby unit tests can simulate error situations which otherwise would be hard to reproduce.
- **Stubs** on the other hand simulate the entire Microservice – however, with a limited functionality. For example, the Stub can return a constant value. Thereby a test can be performed without the really dependent Microservice. For instance, a Stub can be implemented which returns test customers for certain customer numbers – each with certain properties.

Unit tests are within the responsibility of the developers. There are unit test frameworks for all popular programming languages. The tests use knowledge about the internal structure of the units. For example they replace dependencies by Mocks or Stubs. Besides, the knowledge can be employed to run through all code paths for code branches within the tests. The tests are White Box Tests because they exploit knowledge about the structure of the units. Actually, one would have to talk of a transparent box, however, “White Box” is the commonly used term.

One advantage of unit tests is their speed: Even for a complex project the unit tests can be completed within a few minutes. Thereby literally each code change can be safeguarded by unit tests.

Integration Tests

Integration tests check the interplay of the components. Thereby they are supposed to minimize the risk that the integration of the components contains errors. They do not use Stubs or Mocks. The components can be tested as applications via the UI or via special test frameworks. Integration tests evaluate at least whether the individual parts are able to communicate with each other. Furthermore, they can for instance test the logic based on business processes.

In cases where they test business processes the integration tests are similar to acceptance tests which check the requirements of the customers. This area is covered by tools for BDD (Behavior-Driven Design) and ATDD (Acceptance Test-Driven Design). These tools enable a test-driven approach where first the tests are written and afterwards the implementation - even for integration and acceptance tests.

Integration tests do not use information about the system which is to be tested. They are called Black Box Tests since they do not exploit knowledge about the internal structure of the system.

UI Tests

UI tests check the application via the user interface. In principle, they only have to test whether the user interface works correctly. There are numerous frameworks and tools for testing the user interface. Among those are tools for web UIs, but also for desktop or mobile applications. The tests are Blackbox tests. Since they test the user interface, the tests are fragile: Changes to the user interface can cause problems even if the logic remains unchanged. Besides, the tests usually require a complete system setup so that they are slow.

Manual Tests

Finally there can be manual tests. They can either minimize the risk of errors in new features or check certain aspects like security, performance or features which have previously exposed quality problems. They should be explorative: They look at problems in certain areas of the applications. Tests which are aimed at detecting whether a certain error shows up again (regression tests), should never be done manually since automated tests find such errors easier and in a more cost-efficient and reproducible manner. Manual testing is limited to explorative tests.

Load Tests

Load tests analyze the behavior of the application under load. Performance tests on the other hand check the speed, and capacity tests examine how many users or requests the system is able to process. All of these tests evaluate the efficiency of the application. For this purpose, they use similar tools which measure response times and generate load. Besides, such tests can also monitor the use of resources or check whether errors occur upon a certain load. Tests which investigate whether a system is able to cope with a high load in the long term are called endurance tests.

Test Pyramid

The distribution of tests is illustrated by the Test Pyramid ((Fig. 50)[#Fig50]): The broad basis of the Pyramid demonstrates that there are many unit tests. They can be rapidly performed, and most errors can be detected at this level. There are fewer integration tests since they are more laborious and run longer. In addition, there are usually not too many potential errors upon the integration of the parts. The logic itself is also safeguarded by the unit tests. UI tests only have to verify the correctness of the graphical user interface. They are even more laborious since automating UI is complicated, and a complete environment is necessary. Manual tests are only required now and then.

Test-driven development usually results in a Test Pyramid: For each requirement there is an integration test written and for each change to a class a unit test. Thereby automatically many integration tests are created and even more unit tests.

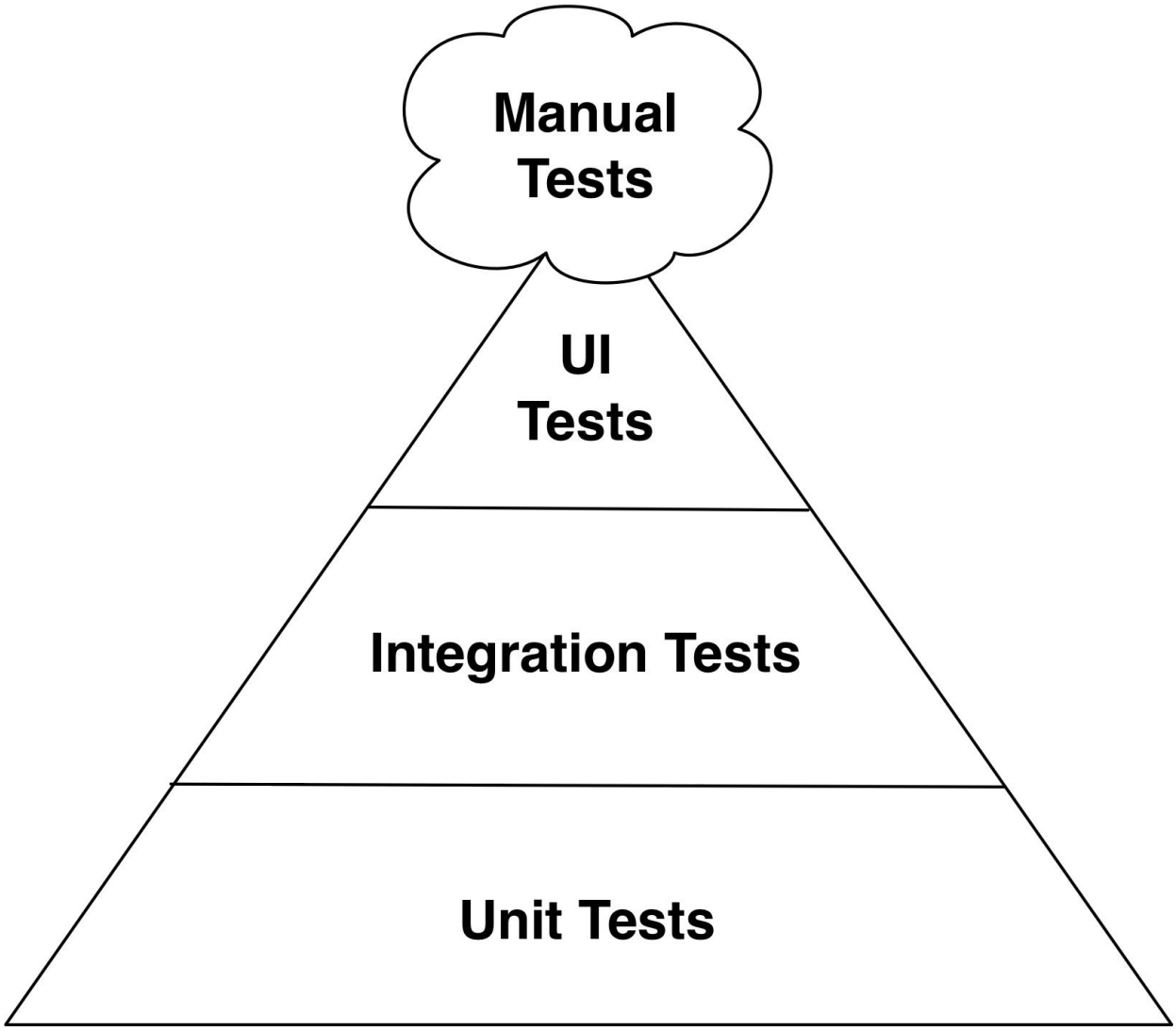


Fig. 50: Test Pyramide: The ideal

The Test Pyramid achieves high quality with low expenditure. The tests are automated as much as possible. Each risk is addressed with a test that is as simple as possible: Logic is tested by simple and rapid unit tests. More laborious tests are restricted to areas which cannot be tested with less effort.

Many projects are very remote from the ideal of the Test Pyramid. Unfortunately, in reality tests are often rather like an ice-cream cone (Fig. 51. In that case there are the following challenges:

- There are comprehensive manual tests since such tests are very easy. Besides, many testers do not have sufficient experience with test automation.

Especially if the testers are not able to write maintainable test code, it is hardly possible to automate tests.

- Tests via the user interface are the easiest type of automation because they are very similar to the manual tests. When there are automated tests, it is mostly UI tests. Unfortunately, automated UI tests are fragile: Changes to the graphical user interface often already lead to problems. Since the tests are testing the entire system, they are slow. If the tests are parallelized, there are often failures because the system experiences a too high load.
- There are rather few integration tests. Such tests require a comprehensive knowledge about the system and about automation techniques, which testers often lack.
- There can be in fact more unit tests than presented in the schema. However, their quality is often bad since developers frequently lack experience in writing unit tests.

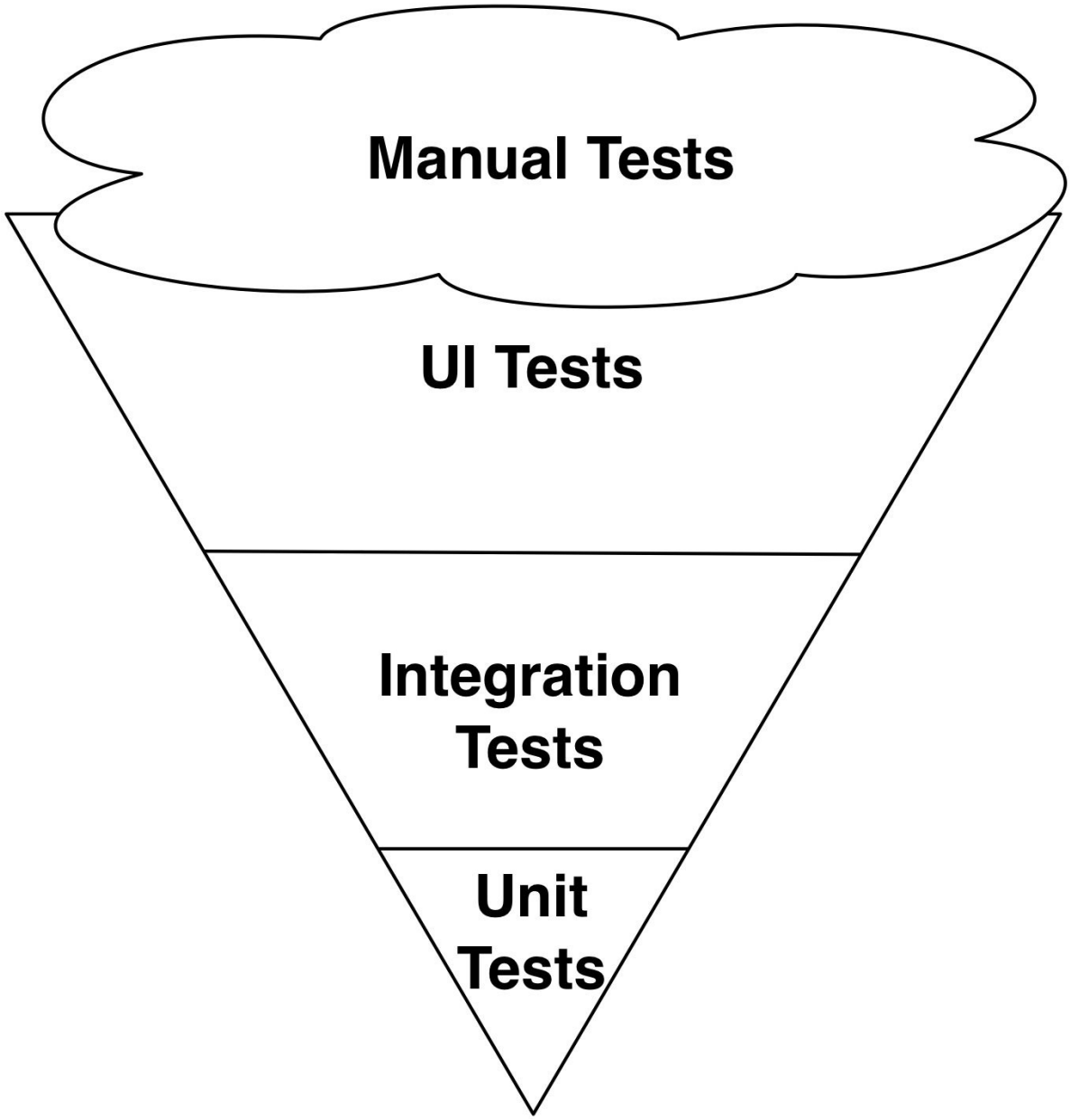


Fig. 51: Test Ice-Cream Cone: Far too common

In addition, unnecessarily complex tests are often used for certain error sources. UI tests or manual tests are used to test logic. For this purpose, however, unit tests would be sufficient and much faster. When testing, developers should try to avoid these problems and the ice-cream cone and instead attempt to implement a Test Pyramid.

Besides, the test concept has to be adjusted to the risks of the respective software and provide tests for the right properties. For example, a project which is

predominantly evaluated based on performance should have automated load or capacity tests. Functional tests might not be so important in this scenario.

Try and Experiment

In which places does the approach in your current project not correspond to the Test Pyramid, but to the Test Ice-Cream Cone?

- Where are manual tests used? Are at least the most important tests automated?
- What is the relationship between UI to integration and unit tests?
- How is the quality of the different tests?
- Is test-driven development used? For individual classes or also for requirements?

Continuous Delivery Pipeline

The Continuous Delivery Pipeline (Fig. 11, section 5.1) defines the different test phases. Therefore, it is interesting for the testing of the Microservices and not as much for the deployment. The unit tests from the Test Pyramid are executed in the commit phase. UI tests can be part of the acceptance tests or can likewise be run in the commit phase. The capacity tests use the complete system and therefore can be regarded as integration tests from the Test Pyramid. The explorative tests are the manual tests from the Test Pyramid.

Automating tests is even more important for Microservices than in other software architectures. The main objective of Microservice-based architectures is independent and frequent software deployment. This can only be implemented when the quality of Microservices is safeguarded by tests. Otherwise the deployment into production is too risky.

11.3 Mitigate Risks at Deployment

An important advantage of Microservices is their fast deployment due to the small size of the deployable units. Besides Resilience avoids that the failure of an individual Microservice causes other Microservices or the entire system to fail. Thereby the risk is lower if an error occurs in production in spite of the tests.

However, there are additional reasons why Microservices minimize the risk of a deployment:

- It is much faster to correct an error since only one Microservice has to be deployed anew. This is by far faster and easier than the deployment of a Deployment Monolith.
- Approaches like Blue/Green Deployment or Canary Releasing (section 12.4) further reduce the risk associated with deployments. Using these techniques a Microservice that contains a bug can be removed from production again with little expenditure and time loss. These approaches are easier to implement with Microservices since it is less effort to provide the required environments for a Microservice than for an entire Deployment Monolith.
- The service can participate in production without doing actual work. Although it will get the same requests like the version in production, all changes to data which the new service would trigger are not actually performed on the data but only compared to the modifications from the service in production. This can for example be achieved by manipulations to the database driver or the database itself. The service can also use a copy of the database. The main point is that in this phase the Microservice will not change the data in production. In addition, messages which the Microservice sends to the outside can be compared with the messages of the Microservices in production instead of sending them really to the recipients. With this approach the Microservice runs already with all special cases of the data in production which even the best test cannot all cover. Moreover, such a procedure can provide more reliable information in regards to performance, although the writes of the data do not occur so that the performance is not entirely comparable. Such approaches can hardly be implemented for a Deployment Monolith since it is hardly possible to have the entire Deployment Monolith run in another instance in production. This would require a lot of resources and a very complex configuration because the Deployment Monolith can introduce changes to data in numerous locations. Even with Microservices this approach is still complex since comprehensive support is necessary in software and deployment. Extra code has to be written for calling the old and the new version and to compare the changes and outgoing messages of both versions. However, this approach is at least feasible.
- Finally, the service can be closely examined via monitoring in order to rapidly recognize and solve problems. This shortens the time until a problem is noticed and addressed. The monitoring fulfills to a certain degree the function of acceptance criteria of load tests. Code which fails in a load test should also create an alarm during monitoring in production. Therefore a close coordination between monitoring and tests is sensible.

In the end the idea behind these approaches is to reduce the risk associated with bringing a Microservice into production instead of addressing the risk by tests. When the new version of a Microservice cannot change any data, its deployment is practically free of risk. This is hardly possible for Deployment Monoliths since the deployment process is much more laborious and time consuming, and requires more resources. Therefore, the deployment cannot be performed fast. Accordingly, the deployment cannot easily be rolled back when errors occur.

The approach is also interesting because some risks can hardly be eliminated by tests. For example, load and performance tests can be an indicator for the behavior of the application in production. However, these tests cannot be completely reliable since the amount of data is different in production, the user behavior is different and the hardware is differently sized. It is not feasible to cover all these aspects in one test environment. In addition, there can be errors which only occur with data sets from production. They are hard to simulate with tests. Monitoring and rapid deployment can in fact be an alternative to tests in a Microservices environment. It is important to think about which risk can be reduced with which type of measure - tests or optimizations of the deployment pipeline.

11.4 Testing the Overall System

In addition to the tests of the individual Microservices also the overall system has to be tested. So there are multiple Test Pyramids: one for each individual Microservice and one for the system in its entirety. For the complete system there are integration tests of the Microservices, UI tests of the entire application and manual tests. Unit tests at this level are the tests of the Microservices since they are the units of the overall system. These tests consist of a complete Test Pyramid of the individual Microservices.

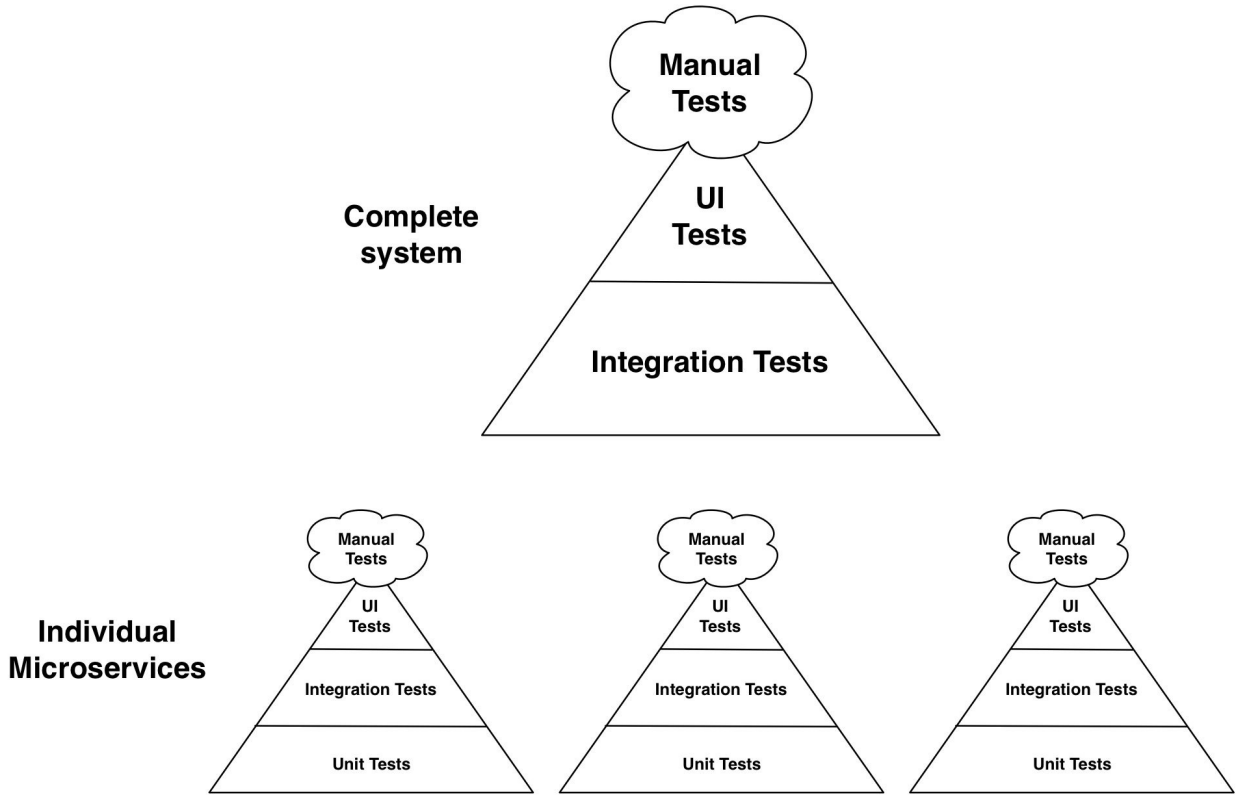


Fig. 52: Test Pyramid for Microservices

The tests of the overall system are responsible for identifying problems which occur in the interplay of the different Microservices. Microservices are distributed systems. Calls can require the interplay of multiple Microservices to return a result to the user. This is a challenge for testing: Distributed systems have many more sources of errors. Tests of the overall system have to address these risks. However, when testing Microservices another approach is chosen: Due to Resilience the individual Microservices should still work in case of problems with other Microservices. Functional tests can be performed with Stubs or Mocks of the other Microservices. In this way Microservices can be tested without the need to build up a complex distributed system and examine it in regards to all possible error scenarios.

Shared Integration Tests

Still each Microservice should be tested prior to its deployment in production in regards to its integration with the other Microservices. This necessitates changes to the Continuous Delivery Pipeline as it was described section 5.1: At the end of the deployment pipeline each Microservice should be tested together with the other Microservices. Each Microservice should run through this step on its own. When new versions of multiple Microservices are tested together at this step, it

will not be clear which Microservice might have caused the failure of the test. Only if in case of a failure it is still clear which Microservice triggered it, is it possible to test multiple Microservices together at this step. But in practice such optimizations are hardly feasible.

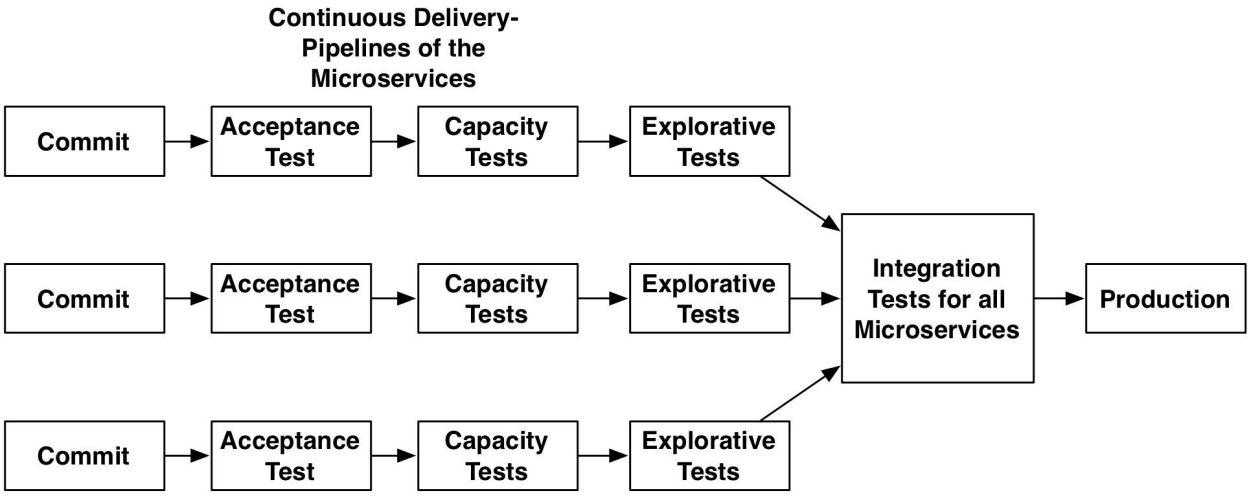


Fig. 53: Integration tests at the end of the Continuous Delivery Pipelines

This reasoning leads to the procedure depicted in Fig. 53: The Continuous Delivery Pipelines of the Microservices end in a common integration test into which each Microservice has to enter separately. When a Microservice is in the integration test phase, the other Microservices have to wait until the integration test is completed. To ensure that indeed only one Microservice at at time runs through the integration tests the tests can be performed in an extra environment. In that case only one Microservice may be delivered in a new version in this environment at a given point in time. The environment enforces the serialized processing of the integration tests of the Microservices.

Such a synchronization point slows down the deployment and therefore the entire process. If the integration test lasts for example one hour, it will only be possible to put eight Microservices through the integration test and into production per eight hours work day. If there are eight teams in the project, each team will be able to deploy a Microservice exactly once per day. This is not sufficient to achieve a rapid error correction in production. Besides, this weakens an essential advantage of Microservices: It should be possible to deploy each Microservice independently. Even though this is in principle still possible, the deployment takes too long. Moreover, the Microservices have now dependencies to each other due to the integration tests – not at the code level, but in the deployment pipelines. In addition, things are not balanced when the Continuous Delivery without the last

integration test requires for instance only one hour, but it is still not possible to get more than one release into production per day.

Avoiding Integration Tests of the Overall System

This problem can be solved by the Test Pyramid. It moves the focus from integration tests of the overall system to integration tests of the individual Microservices and unit tests. When there are few integration tests of the overall system, they will not take as much time. In addition, less synchronization is necessary, and the deployment in production is faster. The integration tests are only meant to test the interplay between Microservices. It is sufficient when each Microservices can reach all dependent Microservices. All other risks can be taken care of prior to this last test. With consumer-driven contract tests (section 11.7) it is even possible to exclude errors in the communication between the Microservices without having to test the Microservices together. All these measure help to reduce the number of integration tests and thereby their total duration. In the end there is no reduction in overall testing – the testing is just moved to other phases: to the tests of the individual Microservices and to the unit tests.

The tests for the overall system can be developed by all teams together. Consistently, they form part of the macro architecture because they concern the system as such and therefore cannot be the responsibility of an individual team (compare section 13.3).

The complete system can also be tested manually. However, it is not feasible that each new version of a Microservice only goes into production after a manual test with the other Microservices. The delays will just be too large. Manual tests of the system as such can for example address features which are not yet activated in production. Alternatively, certain aspects like security can be tested in this manner if problems occurred in these areas previously.

11.5 Testing Legacy Applications and Microservices

Microservices are often used to replace legacy applications. The legacy applications are usually Deployment Monoliths. Therefore the Continuous Delivery Pipeline of the legacy application tests many functionalities which have to be split into Microservices. Because of the many functionalities the test steps of the Continuous Delivery Pipeline take very long for Deployment Monoliths. Accordingly, the deployment in production is very complex and takes long. Under

such conditions it is unrealistic that each small code change to the legacy application goes into production. Often there are deployments at the end of a sprint of 14 days or even only one release per quarter. Nightly tests inspect the current state of the system. Tests can be transferred from the Continuous Delivery Pipeline into the nightly tests. In that case the Continuous Delivery Pipeline will be faster but certain errors are only recognized during the night-time testing. Then the question arises which of the changes of the past day is responsible for the error.

Relocating Tests of the Legacy Application

When migrating from a legacy application to Microservices, tests are especially important. If just the tests of the legacy application are used, they will test a number of functionalities which meanwhile have been moved to Microservices. In that case these tests have to be run at each release of a Microservice – which takes much too long. The tests have to be relocated. They can turn into integration tests for the Microservices (Fig. 54). However, the integration tests of the Microservices should run rapidly. In this phase it is not necessary to use tests for functionalities, which reside in a single Microservice. Then the tests of the legacy application have to turn into integration tests of the individual Microservices or even into unit tests. In that case they are much faster. And they run as tests for a single Microservice so that they do not slow down the shared tests of the Microservices.

Not only the legacy application has to be migrated, but also the tests. Otherwise fast deployments will not be possible in spite of the migration of the legacy application.

The tests for the functionalities which have been transferred to Microservices can be removed from the tests of the legacy application. Step by step this will speed up the deployment of the legacy application. Consequently, changes to the legacy application will also get increasingly easier.

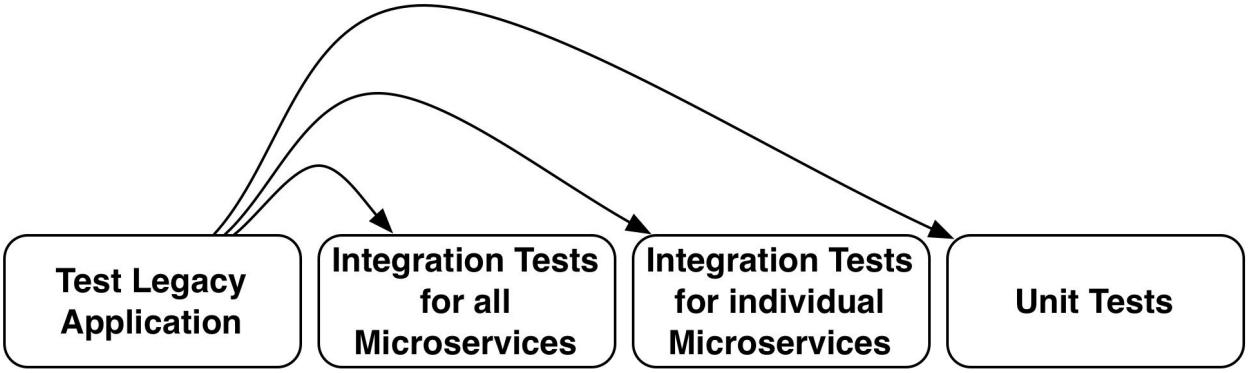


Fig. 54: Relocating tests of legacy applications

Integration Test: Legacy Application and Microservices

The legacy application also has to be tested together with the Microservices. The Microservices have to be tested together with the version of the legacy production which is in production. This ensures that the Microservices will also work in production together with the legacy application. For this purpose, the version of the legacy application running in production can be integrated into the integration tests of the Microservices. It is the responsibility of each Microservice to pass the tests without any errors with this version (Fig. 55).

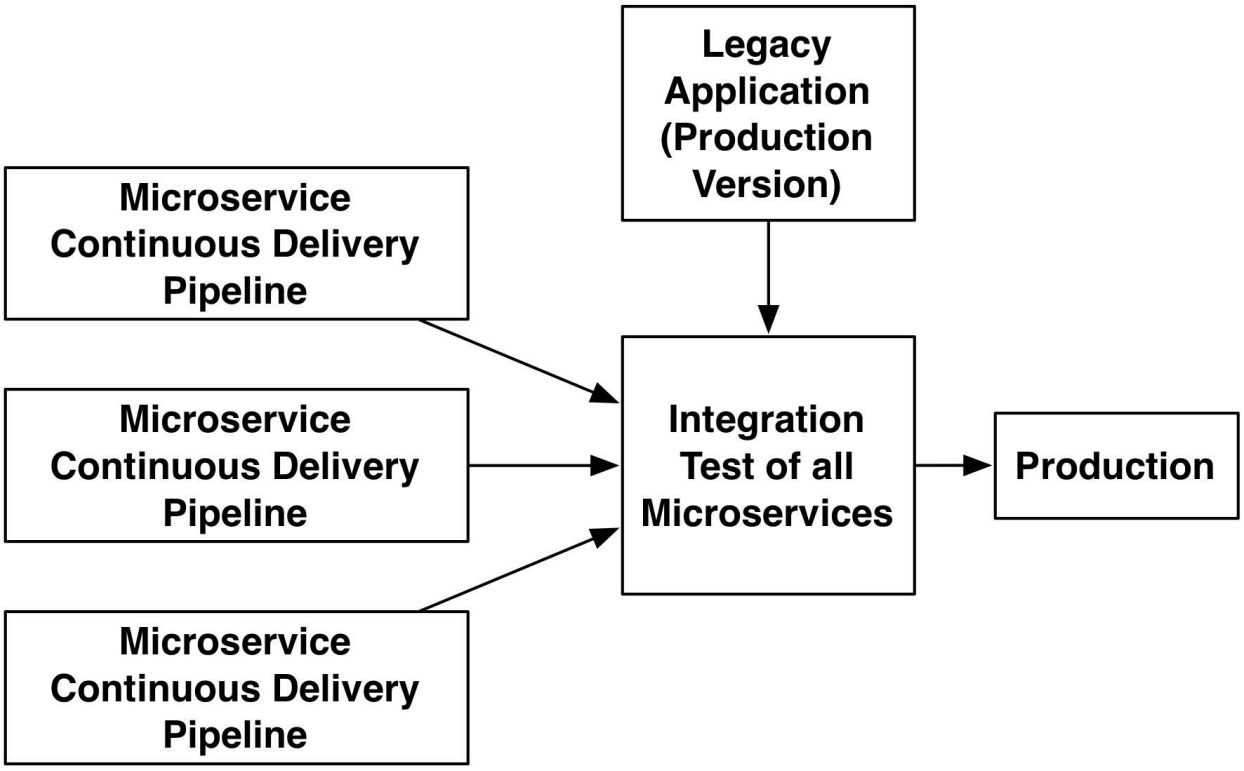


Fig. 55: Legacy Application in the Continuous Delivery Pipelines

When the deployment cycles of the legacy application last days or weeks, a new version of the legacy application will be in development in parallel. The Microservices also have to be tested with this version. This ensures that there will not suddenly be errors occurring upon the release of the new legacy application. The version of the legacy application which is currently in development runs an integration test with the current Microservices as part of its own deployment pipeline (Fig. 56). For this the versions of the Microservices which are in production have to be used.

The versions of the Microservices change much more frequently than the version of the legacy application. A new version of a Microservice can break the Continuous Delivery Pipeline of the legacy application. The team of the legacy application cannot solve these problems since it does not know the code of the Microservices. This version of the Microservice is possibly already in production though. In that case a new version of the Microservice has to be delivered to eliminate the error – although the Continuous Delivery Pipeline of the Microservice ran through successfully.

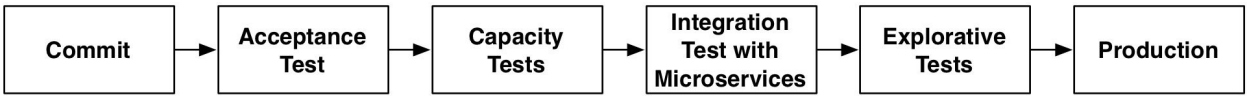


Fig. 56: Microservices in the Continuous Delivery Pipeline of the legacy application

An alternative would be to send the Microservices also through an integration test with the version of the legacy application which is currently in development. However, this prolongs the overarching integration test of the Microservices and therefore renders the development of the Microservices more complex.

The problem can be addressed by consumer-driven contract tests (section 11.7). The expectations of the legacy application to the Microservices and of the Microservices to the legacy application can be defined by consumer-driven contract tests so that the integration tests can be reduced to a minimum.

In addition, the legacy application can be tested together with a Stub of the Microservices. These tests are no integration tests since they only test the legacy application. This allows to reduce the number of overarching integration tests. This concept is illustrated in section 11.6 using tests of Microservices as example. However, this means that the tests of the legacy application have to be adjusted.

11.6 Testing Individual Microservices

Tests of the individual Microservices are the duty of the team which is responsible for the respective Microservice. The team has to implement the different tests such as unit tests, load tests and acceptance tests as part of their own Continuous Delivery Pipeline – as would also be the case for systems which are no Microservices.

However, Microservices require for some functionalities access to other Microservices. This poses a challenge for the tests: It is not sensible to provide a complete environment with all Microservices for each test of each Microservice. On the one hand this would use up too many resources. On the other hand, it is difficult to supply all these environments with the up-to-date software. Technically, light-weight virtualization approaches like Docker can at least reduce the expenditure in terms of resources. However, for 50 or 100 Microservices also this approach will not be sufficient anymore.

Reference Environment

A reference environment in which the Microservices are available in their current version is one possible solution. The tests of the different Microservices can use the Microservices from this environment. However, errors can occur when multiple teams test different Microservices in parallel with the Microservices

from the reference environment. The tests can influence each other and thereby create errors. Besides the reference environment has to be available. When a part of the reference environment breaks down due to a test, in extreme cases tests might be impossible for all teams. The Microservices have to be hold available in the reference environment in their current version. This generates additional expenditure. Therefore a reference environment is not a good solution for the isolated testing of Microservices.

Stubs

Another possibility is the simulation of the used Microservice. For the simulation of parts of a system for testing there are two different options as section 11.2 presented, namely Stubs and Mocks. Stubs are the better choice for the replacement of Microservices. They can support different test scenarios. The implementation of a single Stubs can support the development of all dependent Microservices.

If Stubs are used, the teams have to deliver Stubs for their Microservices. This ensures that the Microservices and the Stubs really behave largely identically. When consumer-driven contract tests also validate the Stubs (compare section 11.7), the correct simulation of the Microservices by the Stubs is ensured.

The Stubs should be implemented with a uniform technology. All teams which use a Microservice also have to use stubs for testing. Handling the stubs is facilitated by a uniform technology. Otherwise a team which employs several Microservices has to master a plethora of technologies for the tests.

Stubs could be implemented with the same technology as the associated Microservices. However, the Stubs should use less resources than the Microservices. Therefore, it is better when the Stubs utilize a simpler technology stack. The example in section 14.13 uses for the Stubs the same technology as the associated Microservices. However, the Stubs deliver only constant values and run in the same process as the Microservices which employ the Stub. Thereby the Stubs use up less resources.

There are technologies which specialize on implementing Stubs. Tools for client-driven contract tests can often also generate Stubs (compare section 11.7).

- mountebank is written in JavaScript with Node.js. It can provide Stubs for TCP, HTTP, HTTPS and SMTP. New Stubs can be generated at run time. The

definition of the Stubs is stored in a JSON file. It defines under which conditions which responses are supposed to be returned by the Stub. An extension with JavaScript is likewise possible. mountebank can also serve as proxy. In that case it forwards requests to a service – alternatively, only the first request is forwarded and the response is recorded. All subsequent requests will be answered by mountebank with the recorded response. In addition to Stubs mountebank also supports Mocks.

- WireMock is written in Java and is under Apache 2 license. This framework makes it very easy to return certain data for certain requests. The behavior is determined by Java code. WireMock supports HTTP and HTTPS. The Stub can run in an separate process, in a servlet container or directly in a JUnit test.
- Moco is likewise written in Java and is under the MIT license. The behavior of the Stubs can be expressed with Java code or with a JSON file. It supports HTTP, HTTPS and simple socket protocols. The Stubs can be started in a Java program or in an independent server.
- stubby4j is written in Java and under MIT license. It utilizes a YAML file for defining the behavior of the Stub. HTTPS is supported as protocol in addition to HTTP. The definition of the data takes place in YAML or JSON. It is also possible to start an interaction with a server or to program the behavior of Stubs with Java. Out of the request information can be copied into the response.

Try and Experiment

Use the example presented in chapter 14 and supplement Stubs with a Stub framework of your choice. The example application uses the configuration file **application-test.properties**. In this configuration it is defined which Stub is used for the tests.

11.7 Consumer-driven Contract Tests

Each interface of a component is ultimately a contract: The caller expects that certain side effects are triggered or that values are returned when it uses the interface. The contract is usually not formally defined. When a Microservice violates the expectations, this manifests itself as error which is either noticed in production or in integration tests. When the contract can be made explicit and tested independently, the integration tests can be freed from the obligation to test the contract without incurring a larger risk for errors during production. Besides,

then it would get easier to modify the Microservices because it would be easier to anticipate which changes cause problems with using other Microservices.

Often changes to system components are not performed because it is unclear which other components use that specific component and how they us it. There is a risk of errors during the interplay with other Microservices, and there are fears that the error will be noticed too late. When it is clear how a Microservice is used, changes are much easier to perform and to safeguard.

Components of the Contract

These aspects belong to the contract of a Microservice:

- The data formats define in which format information is expected by the other Microservices and how they are passed over to a Microservice.
- The interface determines which operations are available.
- Procedures or protocols define which operations can be performed in which sequence with which results.
- Finally, there is meta information associated with the calls which can comprise for example a user authentication.
- In addition, there can be certain non-functional aspects like the latency time or a certain throughput.

Contracts

There are different contracts between the consumers and the provider of a service:

- The **Provider Contract** comprises everything the service provider provides. There is one such contract per service provider. It completely defines the entire service. It can for instance change with the version of the service (compare section 9.6).
- The **Consumer Contract** comprises all functionalities which a service user really utilizes. There are several such contracts per service – at least one with each user. The contract comprises only the parts of the service which the user really employs. It can change through modifications to the service consumer.
- The **Consumer-driven Contract** (CDC) comprises all user contracts. Therefore, it contains all functionalities which any service consumer utilizes. There is only one such contract per service. Since it depends on the user contracts, it can change when the service consumers add new calls to the service provider or when there are new requirements for the calls.

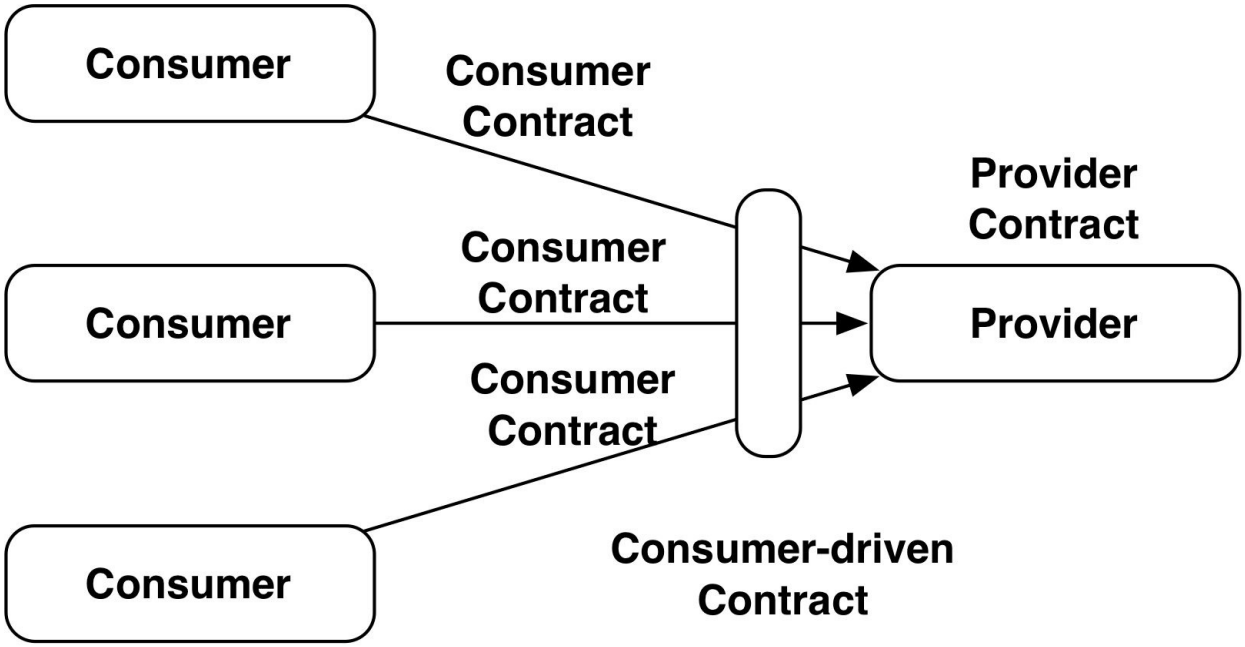


Fig. 57: Differences between Consumer and Provider Contracts

The Consumer-driven Contract makes clear which which components of the provider contracts are really used. This clarifies also where the Microservice can still change its interface and which components of the Microservice are not used.

Implementation

Ideally, a Consumer-driven Contract turns into a consumer-driven contract test which the service provider can perform. It has to be possible for the service consumer to change these tests. They can be stored together in the version control with the Microservice of the service provider. In that case the service consumers have to get access to the version control of the service provider. Otherwise the tests can also be stored in the version control of the service consumers. In that case the service provider has to fetch the tests out of the version control and execute them with each version of the software. However, in that case it is not possible to version the tests together with the tested software since tests and tested software are in two separate projects within the version control.

The entirety of all tests represents the Consumer-driven Contract. The tests of each team correspond to the Consumer Contract of each team. The consumer-driven contract tests can be performed as part of the tests of the Microservice. If they are successful, all service consumers should be able to successfully work together with the Microservice. The test precludes that errors will only be noticed during the integration test. Besides, modifications to the Microservices get easier because requirements for the interfaces are known and can be tested without

special expenditure. Therefore, the risk associated with changes which affect the interface is much smaller since problems will be noticed prior to integration tests and production.

Tools

To write consumer-driven contract tests a technology has to be defined. The technology should be uniform for all projects because a Microservice can use several other Microservices. In that case a team has to write tests for different other Microservices. This is easier when there is a uniform technology. Otherwise the teams have to know numerous different technologies. The technology for the tests can differ from the technology used for implementation.

- An **arbitrary test framework** is an option for implementing the consumer-driven contract tests. For load tests additional tools can be defined. In addition to the functional requirements there can also be requirements in regards to the load behavior. However, it has to be clearly defined how the Microservice is provided for the test. For example, it can be available at a certain port on the test machine. In this way the test can take place via the interface which is also used for access by other Microservices.
- In the example application (section 14.13) simple **JUnit** tests are used for testing the Microservice and for verifying whether the required functionalities are supported. When incompatible changes to data formats are performed or the interface is modified in a incompatible manner, the tests fail.
- There are tools especially designed for the implementation of consumer-driven contract tests. An example is **Pacto**. It is written in Ruby and under the MIT licence. Pacto supports REST/HTTP and supplements such interfaces with a contract. Pacto can be integrated into a test structure. In that case Pacto compares the header with expected values and the JSON data structures in the body with JSON schemas. This information represents the contract. The contract can also be generated out of a recorded interaction between a client and a server. Based on the contract Pacto can validate the calls and responses of a system. In addition, Pacto can create with this information simple Stubs. Moreover, Pacto can be used in conjunction with RSpec to write tests in Ruby. Also test systems which are written in other languages than Ruby can be tested in this way. Without RSpec Pacto offers the possibility to run a server. Thereby it is possible to use Pacto also outside of a Ruby system.

- **Pact** is likewise written in Ruby and under MIT licence. The service consumer can employ Pact to write a Stub for the service and to record the interaction with the Stub. This results in a Pact file which represents the contract. It can also be used for testing whether the actual service correctly implements the contract. Pact is especially useful for Ruby, however **pact-jvm** supports a similar approach for different JVM languages like Scala, Java, Groovy or Clojure.

Try and Experiment

Use the example presented in chapter 14 and supplement consumer-driven contracts with a framework of your choice. The example application has the configuration **application-test.properties**. In this configuration it is defined which Stub is used for the tests. Verify also the contracts in the production environment.

11.8 Testing Technical Standards

Microservices have to fulfill certain technical requirements. For example, Microservices should register themselves in Service Discovery and keep functioning even if other Microservices break down. Tests can verify these properties. This entails a number of advantages:

- The guidelines are unambiguously defined by the test. Therefore, there is no discussion how precisely the guidelines are meant.
- They can be tested in an automated fashion. Thereby it is clear at any time whether a Microservice fulfills the rules or not.
- New teams can test new components as to whether they comply with the rules or not.
- When Microservices do not employ the usual technology stack, it can still be ensured that they behave correctly from a technical point of view.

  Among the possible tests are:

- The Microservices have to register in the Service Discovery (section 8.9). The test can verify whether the component registers at service registry upon starting.
- Besides, the shared mechanisms for configuration and coordination have to be used (section 8.8). The test can control whether certain values from the

central configuration are read out. For this purpose, an individual test interface can be implemented.
- A shared security infrastructure can be checked by testing the use of the Microservice via a certain token (section 8.12).
- In regards to documentation and metadata (section 8.13) it can be tested whether a test can access the documentation via the defined path.
- In regards to monitoring (section 12.3) and logging (section 12.2) it can be examined whether the Microservice provides data to the monitoring interfaces upon starting and delivers values resp. log entries.
- In regards to deployment (section 12.4) it is sufficient to deploy and start the Microservice on a server. When the defined standard is used for this, this aspect is likewise correctly implemented.
- As test for control (section 12.5) the Microservice can simply be restarted.
- To test for Resilience (section 10.5) in the simplest scenario it can be checked whether the Microservice at least boots also in absence of the dependent Microservices and displays errors in monitoring. The correct functioning of the Microservice upon availability of the other Microservices is ensured by the functional tests. However, a scenario where the Microservice cannot reach any other service is not addressed in normal tests.

In the easiest case the technical test can just start and deploy the Microservice. Thereby deployment and control are already tested. Dependent Microservices do not have to be present for that. Starting the Microservice should also be possible without dependent Microservices due to Resilience. Subsequently, logging and monitoring can be examined which should also work and contain errors in this situation. Finally, the integration in the shared technical services like Service Discovery, configuration and coordination or security can be checked.

Such a test is not hard to write and can render many discussions about the precise interpretation of the guidelines superfluous. Therefore, this test is very useful. Besides, it tests scenarios which are usually not covered by automated tests – for instance the breakdown of dependent systems.

This test does not necessarily provide complete security that the Microservice complies with all rules. However, it can at least examine whether the fundamental mechanisms function.

Technical standards can easily be tested with scripts. The scripts should install the Microservice in the defined manner on a virtual machine and start it.

Afterwards the behavior, for instance in regards to logging and monitoring, can be tested. Since technical standards are specific for each project, a uniform approach is hardly possible. Under certain conditions a tool like Serverspec can be useful. It serves to examine the state of a server. Therefore, it can easily determine whether a certain port is used or whether a certain service is active.

11.9 Conclusion

Reasons for testing are on the one hand the risk that problems are only noticed in production and on the other hand that tests serve as an exact specification of the system (section 11.1).

Section 11.2 illustrated by using the concept of the Test Pyramid how tests should be structured: The focus should be on fast, easily automatable unit tests. They address the risk that there are errors in the logic. Integration tests and UI tests then only ensure the integration of the Microservices with each other and the correct integration of the Microservices into the UI.

As section 11.3 showed, Microservices can additionally deal with the risk of errors in production in a different manner: Microservice deployments are faster, they influence only a small part of the system, and Microservices can even run blindly in production. Thereby the risk of deployment decreases. Thus it can be sensible instead of comprehensive tests to rather optimize the deployment in production to such an extent that it is for all practical purposes free of risk. In addition, the section discussed that there are two types of Test Pyramids for Microservice-based systems: one per Microservice and one for the overall system.

Testing the overall system entails the problem that each change to a Microservice necessitates a run through this test. Therefore, this test can turn into a bottleneck and should be very fast. Thus, when testing Microservices, one objective is to reduce the number of integration tests across all Microservices (section 11.4).

When replacing legacy applications not only their functionality has to be transferred into Microservices, but also the tests for the functionalities have to be moved into the tests of the Microservices (section 11.5). Besides, each modification to a Microservice has to be tested in the integration with the version of the legacy application used in production. The legacy application normally has a much slower release cycle than the Microservices. Therefore, the version of the

legacy application which is at the time in development has to be tested together with the Microservices.

For testing individual Microservices the other Microservices have to be replaced by Stubs. This allows to uncouple the tests of the individual Microservices from each other. Section 11.6 introduced a number of concrete technologies for creating Stubs.

In section 11.7 client-driven contract tests were presented. With this approach the contracts between the Microservices get explicit. This allows a Microservice to check whether it fulfills the requirements of the other Microservices – without the need for an integration test. Also for this area a number of tool are available.

Finally, section 11.8 demonstrated that technical requirements to the Microservices can likewise be tested in an automated manner. This allows to unambiguously establish whether a Microservice fulfills all technical standards.

Essential Points

- Established best practices like the Test Pyramid are also sensible for Microservices.
- Common tests across all Microservices can turn into a bottleneck and therefore should be reduced, for example by performing more consumer-driven contract tests.
- With suitable tools Stubs can be created from Microservices.

12 Operations and Continuous Delivery of Microservices

Deployment and operation are additional components of the Continuous Delivery Pipeline (compare section 11.1). When the software has been tested in the context of the pipeline the Microservices go into production. There monitoring and logging collect information which can be used for the further development of the Microservices.

The operation of a Microservice-based system is more laborious than the operation of a Deployment Monolith. There are many more deployable artifacts which all have to be surveilled. Section 12.1 discusses the typical challenges associated with the operation of Microservice-based systems in detail. Logging is the topic of section 12.2. Section 12.3 focuses on the monitoring of the Microservices. Deployment is treated in section 12.4. Section 12.5 shows necessary measures for directing a Microservice from the outside, and section 12.6 finally describes suitable infrastructures for the operation of Microservices.

The challenges associated with operation should not be underestimated. It is in this area where the most complex problems associated with the use of Microservices frequently arise.

12.1 Challenges Associated with the Operation of Microservices

Challenge: Numerous Artifacts

Teams who have so far only run Deployment Monoliths are confronted with the problem that there are very many additional deployable artifacts in Microservices-based systems. Each Microservice is independently brought into production and therefore a separate deployable artifact. Fifty, hundred or more Microservices are definitely realistic. The concrete number depends on the size of the project and the size of the Microservices. Such a number of deployable artifacts is hardly met with outside of Microservices-based architectures. All these artifacts have to be versioned independently because only then it can be

tracked which code runs currently in production. Besides, this allows to bring each Microservice independently in a new version into production.

When there are so many artifacts, there has to be a correspondingly high number of Continuous Delivery Pipelines. They do not only comprise the deployment in production but also the different testing phases. In addition, many more artifacts have to be surveilled in production by logging and monitoring. This is only possible when all these processes are mostly automated. For a small number of artifacts manual interventions might still be acceptable. Such an approach is simply not possible anymore for the large number of artifacts contained in a Microservice-based architecture.

The challenges in the areas of deployment and infrastructure are for sure the most difficult ones encountered when introducing Microservices. Many organizations are not sufficiently proficient in automation although automation is also very advantageous in other architectural approaches and should already be routine.

There are different approaches for achieving the necessary automation:

Delegate into Teams

The easiest option is to delegate this challenge to the teams which are responsible for the development of the Microservices. In that case each team has not only to develop its Microservice, but also to take care of its operation. They have the choice to either use appropriate automation for it or to adopt automation approaches from other teams.

The team does not even have to cover all areas. When there is no need to evaluate log data to achieve reliable operation, the team can decide not to implement a system for evaluating log data. A reliable operation without surveilling the log output is hardly possible though. However, this risk is then within the responsibility of the respective team.

This approach only works when the teams have a lot of knowledge regarding operation. Another problem is that the wheel is invented over and over again by the different teams: Each team implements automation independently and might use different tools for it. This approach entails the danger that the anyhow laborious operation of the Microservices gets even more laborious due to the heterogeneous approaches taken by the teams. The teams have to do this work. This interferes

with the rapid implementation of new features. However, the decentralized decision which technologies to use increases the independence of the teams.

Unify Tools

Because of the higher efficiency, unification can be a sensible approach for deployment. The easiest way to obtain uniform tools is to prescribe one tool for each area – deployment, test, monitoring, logging, deployment pipeline. In addition, there will be guidelines and best practices like for instance immutable server or the separation of build environment and deployment environment. This allows for the identical implementation of all Microservices and will facilitate operation since the teams only need to be familiar with one tool for each area.

Specify Behavior

Another option is to specify the behavior of the system. One example: When log output is supposed to be evaluated in a uniform manner across services, it is sufficient to define a uniform log format. The log framework does not necessarily have to be prescribed. Of course, it is sensible to offer for at least one log framework a configuration which generates this output format. This increases the motivation of the teams to use this log framework. In this way uniformity is not forced, but emerges on its own since the teams will minimize their own effort. When a team regards the use of another log framework or programming language which necessitates another log framework as more advantageous, it can still use these technologies.

Defining uniform formats for log output has an additional advantage: The information can be delivered to different tools which process log files differently. This allows operations to screen log files for errors while the business stakeholders create statistics. Operation and business stakeholders can use different tools which use the uniform format as shared basis.

Similarly, behavior can be defined for the other areas of operation such as deployment, monitoring or the deployment pipeline.

Micro and Macro Architecture

Which decisions can be made by the team and which have to be made for the overall project corresponds to the separation of the architecture into micro and macro architecture (compare section 13.3). Decisions the team can make belong to micro architecture while decisions which are made across all teams for the

overall project are part of the macro architecture. Technologies or the desired behavior for logging can be either part of the macro or the micro architecture.

Templates

Templates offer the option to unify Microservices in these areas and to increase the productivity of the teams. Based on a very simple Microservice a template demonstrates how the technologies can be used and how Microservices are integrated into the operation infrastructure. The example can simply respond to a request with a constant value since the domain logic is not the point here.

The template will make it easy and fast for a team to implement a new Microservice. At the same time, each team can easily make use of the standard technology stack. So the uniform technical solution is at the same time the most attractive for the teams. Templates achieve a large degree of technical uniformity between Microservices without prescribing the used technology. In addition, a faulty use of the technology stack is avoided when the template demonstrates the correct use.

A template should contain the complete infrastructure in addition to the code for an exemplary Microservice. This refers to the Continuous Delivery Pipeline, the build, the Continuous Integration Platform, the deployment in production and the necessary resources for running the Microservice. Especially build and Continuous Delivery Pipeline are important since the deployment of a large number of Microservices is only possible when these are automated.

The template can be very complex when it really contains the complete infrastructure – even if the respective Microservice is very simple. It is not necessarily required to provide at once a complete and perfect solution. The template can also be built up in a stepwise manner.

The template can be copied into each project. This entails the problem that changes to the template are not propagated into the existing Microservices. On the other hand, this approach is much easier to implement than an approach which allows for the automated adoption of changes. Besides such an approach would create dependencies between the template and practically all Microservices. Such dependencies should be avoided for Microservices.

The templates fundamentally facilitate the generation of new Microservices. Accordingly, teams are more likely to create new Microservices. Thereby they

can more easily distribute Microservices in multiple smaller Microservices. Thus templates help to keep Microservices small. When the Microservices are rather small, the advantages of a Microservice-based architecture can be exploited even better.

12.2 Logging

By logging an application can easily provide information about which events occurred. These can be errors, but also events like the registration of a new user which are mostly interesting for statistics. Finally, log data can help developers to locate errors by providing detailed information.

In normal systems logs have the advantage that they can be written very easily and that the data can be persisted without huge effort. Besides, log files are human-readable and can be easily searched.

Logging for Microservices

For Microservices writing and analyzing log files is hardly sufficient:

- Many requests can only be handled by the interplay of multiple Microservices. In that case the log file of a single Microservice is not sufficient to understand the complete sequence of events.
- The load is often distributed across multiple instances of one Microservice. Therefore, the information contained in the log file of an individual instance is not very useful.
- Finally, due to increased load, new releases or crashes, new instances of a Microservice start constantly. The data from a log file can get lost when a virtual machine is shut down and its hard disc is subsequently deleted.

It is not necessary for Microservices to write logs into their file system because the information can anyhow not be analyzed there. Only writing to the central log server is definitely necessary. This has also the advantage that the Microservices utilize less local storage.

Usually, applications just log text strings. The centralized logging parses the string. During parsing relevant information like time stamps or server names are extracted. Often parsing goes even beyond that and scrutinizes the texts more closely. If it is possible to determine for instance the identity of the current user from the logs, all information about a user can be selected from the log data of the Microservices. In a way the Microservice hides the relevant information in a

string which the log system subsequently takes apart again. To facilitate the parsing log data can be transferred into a data format like JSON. In that case the data can already be structured during logging. They are not first packaged into a string which then has to be laboriously parsed. Likewise, it is sensible to have uniform standards: When a Microservice logs something as an error, then an error should really have occurred. In addition, the semantics of the other log levels should be uniform across all Microservices.

Technologies for Logging via the Network

Microservices can support central logging by sending log data directly via the network. Most log libraries support such an approach. Special protocols like GELF (Graylog Extended Log Format) can be used for this or long established protocols like syslog which is the basis for logging in UNIX systems. Tools like the logstash-forwarder, Beaver or Woodchuck are meant to send local files via the network to a central log server. They are sensible in cases where the log data are supposed to be also locally stored in files.

ELK for Centralized Logging

Logstash, Elasticsearch and Kibana can serve as tools for the collection and processing of logs on a central server.

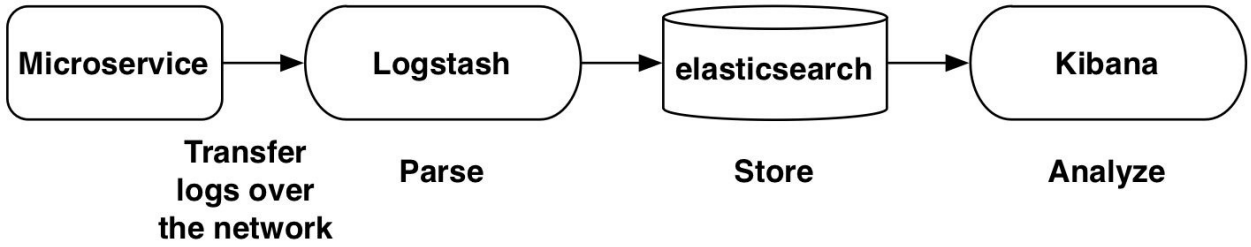


Fig. 58: ELK infrastructure for log analysis

- With the aid of **Logstash** log files can be parsed and collected by servers in the network. Logstash is a very powerful tool. It can read data from a source, modify or filter data, and finally write it into a sink. Apart from importing logs from the network and storage in Elasticsearch Logstash supports many other data sources and data sinks. For example, data can be read from message queues or databases or written into them. Finally, Logstash can also parse data and supplement it – for example time stamps can be added to each log entry or individual fields can be cut out and further processed.
- **Elasticsearch** stores log data and makes them available for analyses. Elasticsearch cannot only search the data with full text search, but it can also

search in individual fields of structured data and permanently store the data like a database. Finally, Elasticsearch offers statistical functions and can use those to analyze data. As a search engine Elasticsearch is optimized for fast response times so that the data can be analyzed quasi interactively.
- **Kibana** is a web user interface which allows to analyze data from Elasticsearch. In addition to simple queries also statistical evaluations, visualizations and diagrams can be created.

These tools form the ELK stack (Elasticsearch, Logstash, Kibana). All three are open source projects and are under Apache 2.0 license.

Scaling ELK

Especially in case of Microservices log data accumulate often in large amounts. Therefore, in Microservice-based architectures the system for the central processing of logs should be highly scalable. A good scalability is one of the advantages of the ELK stack:

- **Elasticsearch** can distribute the indices into shards. Each data set is stored in a single shard. As the shards can be located on different servers, this allows for load balancing. In addition, shards can be replicated across several servers to improve fail safeness. Besides, a read access can be directed to an arbitrary replica of the data. Thereby replicas can serve to scale read access.
- **Logstash** can write logs into different indices. Without an additional configuration Logstash would write the data for each day into a different index. Since the current data usually is read more frequently, this allows to reduce the amount of data which has to be searched for a typical request and therefore improves performance. Besides, there are still other possibilities to distribute the data to indices – for instance according to the geographic origin of the user. This also promotes the optimization of the data amounts which have to be searched.
- Log data can be buffered in a **Broker** prior to processing by Logstash. The Broker serves as buffer. It stores the messages when there are so many log messages that they cannot be immediately processed. Redis is often used as Broker – a fast in memory database.

Graylog

The ELK stack is not the only solution for the analysis of log files. Graylog is also an open source solution and likewise utilizes Elasticsearch for storing log data.

Besides it uses MongoDB for metadata. Graylog defines its own format for the log messages: The already mentioned GELF (Graylog Extended Log Format) standardizes the data which are transmitted via the network. For many log libraries and programming languages there are extensions for GELF. Likewise, the respective information can be extracted from the log data or surveyed with the UNIX tool syslog. Also Logstash supports GELF as in- and output format so that Logstash can be combined with Graylog. Graylog has a web interface which allows to analyze the information from the logs.

Splunk

Splunk is a commercial solution and already for a long time on the market. Splunk presents itself as a solution which does not only analyze log files, but can generally analyze machine data and big data. For processing logs Splunk gathers the data via a Forwarder, prepares it via an Indexer for searching, and Search Heads take over the processing of search requests. Its intention to serve as an enterprise solution is underlined by the security concept. Customized analysis, but also alerts in case of certain problems are possible. Splunk can be extended by numerous plug-ins. Besides there are apps which provide ready-made solutions for certain infrastructures such as Microsoft Windows Server. The software does not necessarily have to be installed in your own computing center, but is also available as Cloud solution.

Stakeholders for Logs

There are different stakeholders for logging. However, the analysis options of the log servers are so flexible and the analyses so similar that one tool is normally sufficient. The stakeholders can create their own dashboards with the information that is relevant to them. For specific requirements the log data can be passed on to other systems for evaluation.

Correlation IDs

Often multiple Microservices work together on a request. The path the request takes through the Microservices has to be traceable for analysis. For filtering all log entries to a certain customer or to a certain request a correlation ID can be used. This ID unambiguously identifies a request to the overall system and is passed along during all communication between Microservices. In this manner log entries for all systems to a single request are easy to find in the central log system, and the processing of the requests can be tracked across all Microservices.

Such an approach can for instance be implemented by transferring a request ID for each message within the headers or within the payloads. Many projects implement the transfer in their own code without using a framework. For Java there is the library tracee which implements the transfer of the IDs. Some log frameworks support a context which is logged together with each log message. In that case it is only necessary to put the correlation ID into the context when receiving a message. This obliterates the need to pass the correlation ID on from method to method. When the correlation ID is bound to the thread, problems can arise when the processing of a request involves several threads. Setting the correlation ID in the context ensures that all log messages contain the correlation ID. How the correlation ID is logged has to be uniform across all Microservices so that the search for a request in the logs works for all Microservices.

Zipkin: Distributed Tracing

Also in regards to performance evaluations have to be made across Microservices. When the complete path of the requests is traceable, it can be identified which Microservice represents a bottleneck and requires an especially long time for processing requests. With the aid of a distributed tracing it can be determined for a request which Microservice needs how much time for answering a request and where optimization should start. Zipkin enables exactly this type of investigations. It comprises support for different network protocols so that a request ID is automatically passed on via these protocols. In contrast to the correlation IDs the objective is not to correlate log entries, but to analyze the time behavior of the Microservices. For this purpose Zipkin offers suitable analysis tools.

Try and Experiment

Define a technology stack which enables a Microservice-based architecture to implement logging:

- How should the log messages be formatted?
- Define a logging framework if necessary.
- Determine a technology for collecting and evaluating logs.

This section listed a number of tools for the different areas. Which properties are especially important? The objective is not a complete product evaluation, but a general weighing of advantages and disadvantages.

Chapter 14 shows an example for a Microservice-based architecture and in section 14.14 there are suggestions how the architecture can be supplemented with a log analysis.

How does your current project handle logging? Is it maybe possible to implement parts of these approaches and technologies also in your project?

12.3 Monitoring

Monitoring surveils the metrics of a Microservice and uses other information sources than logging. Monitoring uses mostly numerical values which provide information about the current state of the application and indicate how this state changes over time. Such values can represent the number of processed calls over a certain time, the time needed for processing the calls or also system values like the CPU or memory utilization. If certain thresholds are surpassed or not reached, this indicates a problem and can trigger an alarm so that somebody can solve the problem. Or even better: The problem is solved automatically. For example, an overload can be addressed by starting additional instances.

Monitoring offers feedback from production which is not only relevant for operation, but also for developers or the users of the system. Based on the information from monitoring they can better understand the system and therefore make informed decisions about how the system should be developed further.

Basic Information

Basic monitoring information should be mandatory for all Microservices. This makes it easier to get an overview of the state of the system. All Microservices should deliver the required information in the same format. Besides components

of the Microservice system can likewise use the values. Load balancing for instance can use a health check to avoid accessing Microservices which cannot process calls.

The basic values all Microservices should provide can comprise the following:

- There should be a value which indicates the availability of the Microservice. In this manner the Microservice signals whether it is capable of processing calls at all (“alive”).
- Detailed information regarding the availability of the Microservice is another important metric. One relevant information is whether all Microservices used by the Microservice are accessible and whether all other resources are available (“health”). This information does not only indicate whether the Microservice functions, but also provide hints which part of a Microservice is currently unavailable and why it failed. Importantly, it becomes apparent whether the Microservice is unavailable because of the failure of another Microservice or because the respective Microservice itself is having a problem.
- Information about the version of a Microservice and additional meta information like the contact partner or used libraries and their versions as well as other artifacts can also be provided as metrics. This can cover part of the documentation (compare section 8.13). Alternatively, it can be checked which version of the Microservice is actually currently in production. This facilitates the search for errors. Besides, an automated continuous inventory of the Microservices and other used software is possible, which simply inquires after these values.

Additional Metrics

Additional metrics can likewise be recorded by monitoring. Among the possible values are for instance response times, the frequency of certain errors or the number of calls. These values are usually specific for a Microservice so that they do not necessarily have to be offered by all Microservices. An alarm can be triggered when certain thresholds are reached. Such thresholds are different for each Microservice.

Nevertheless, a uniform interface for accessing the values is sensible when all Microservices are supposed to use the same monitoring tool. Uniformity can tremendously reduce expenditure in this area.

Stakeholders

There are different stakeholders for the information from monitoring:

- **Operations** wants timely to be informed about problems to enable a smooth operation of the Microservice. In case of acute problems or failures it wants to get an alarm – at any day or night time – via different means like pager or SMS. Detailed information is only necessary when the error has to be analyzed more closely – often together with the developers. Operations is not only interested in the values from the Microservice itself, but also in monitoring values of the operating system, the hardware or the network.
- **Developers** mostly focus on information from the application. They want to understand how the application functions in production and how it is employed by the users. From this information they deduce optimizations, especially at the technical level. Therefore, they need very specific information. If the application is for instance too slow in responding to a certain type of call, the system has to be optimized for this type of call. To do so it is necessary to obtain as much information as possible about exactly this type of call. Other calls are not as interesting. Developers evaluate this information in detail. They might even be interested in analyzing calls of just one specific user or a circle of users.
- The **business stakeholders** are interested in the business success and the resulting business numbers. Such information can be provided by the application specifically for the business stakeholders. The business stakeholders then generate statistics based on this information and thereby prepare business decisions. On the other hand, they are usually not interested in technical details.

The different stakeholders are not only interested in different values, but also analyze them differently. Standardizing the data format is sensible to support different tools and nevertheless enable all stakeholders to access all data.

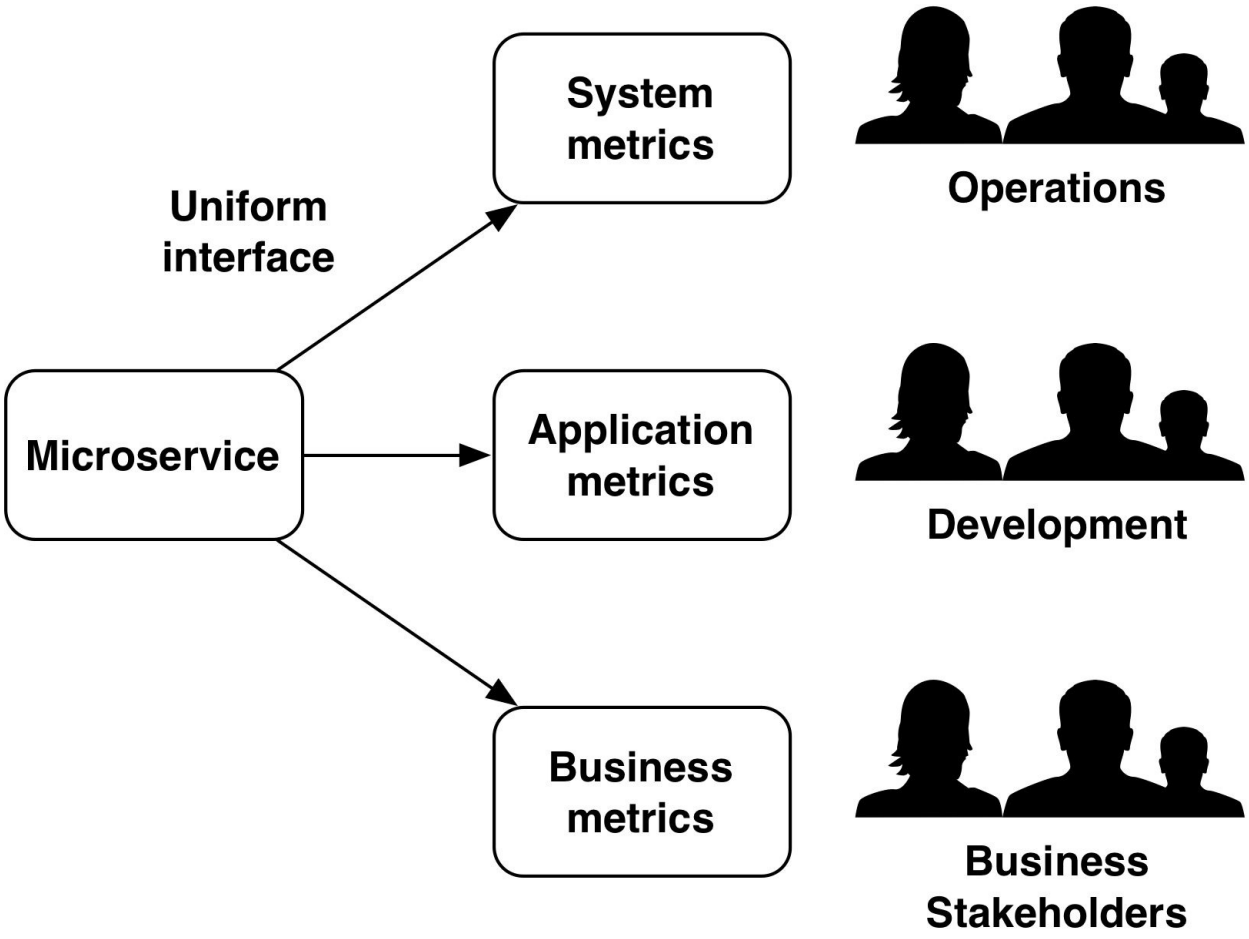


Fig. 59: Stakeholders and their monitoring data

Fig. 59 shows an overview of a possible monitoring of a Microservice-based system. The Microservice offers the data via a uniform interface. Operations uses monitoring to surveil for instance threshold values. Development utilizes a detailed monitoring to understand processes within the application. And the business stakeholders look at the business data. The individual stakeholders might use more or less similar approaches: The stakeholders can for instance use the same monitoring software with different dashboards or entirely different software.

Correlate with Events

In addition, it can be sensible to correlate data with an event such as a new release. This requires that information about the event has to be handed over to monitoring. When a new release creates markedly more revenue or causes decisively longer response times, this is for sure an interesting realization.

Monitoring = Tests?

In a certain way monitoring is another version of testing (compare section 11.4). While tests look at the correct functioning of a new release in a test environment, monitoring examines the behavior of the application in a production environment. The integration tests should also be reflected in monitoring. When a problem causes an integration test to fail, there can be an associated alarm in monitoring. Besides, monitoring should also be activated for test environments to pinpoint problems already in the tests. When the risk associated with deployments is reduced by suitable measures (compare section 12.4), the monitoring can even take over part of the tests.

Dynamic Environment

Another challenge when working with Microservice-based architectures is that Microservices come and go. During the deployment of a new release an instance can be stopped and started anew with a new software version. When servers fail, instances shut down, and new ones are started. For this reason monitoring has to occur separated from the Microservices. Otherwise the stopping of a Microservice would influence the monitoring infrastructure or may even cause it to fail. Besides, Microservices are a distributed system. The values of a single instance are not telling in themselves. Only by collecting values of multiple instances the monitoring information gets relevant.

Concrete Technologies

Different technologies can be used for monitoring Microservices:

- **Graphite** can store numerical data and is optimized for processing time series data. Such data occur frequently during monitoring. The data can be analyzed in a web application. Graphite stores the data in its own database. After some time the data are automatically deleted. Monitoring values are accepted by Graphite in a very simple format via a socket interface.
- **Grafana** extends Graphite by alternative dashboards and other graphical elements.
- **Seyren** extends Graphite by a functionality for triggering alarms.
- **Nagios** is a comprehensive solution for monitoring and can be an alternative to Graphite.
- **Icinga** has originally been a fork of Nagios and therefore covers a very similar use case.
- **Riemann** focuses on the processing of event streams. It uses a functional programming language to define logic for the reaction to certain events. For

this purpose, a fitting dashboard can be configured. Messages can be sent by SMS or e-mail.

- **Packetbeat** uses an agent which records the network traffic on the computer to be monitored. This allows Packetbeat to determine with minimal effort which requests take how long and which nodes communicate with each other. It is especially interesting that Packetbeat uses Elasticsearch for data storage and Kibana for analysis. These tools are also widely used for analyzing log data (compare section 12.2). Having only one stack for the storage and analysis of logs and monitoring reduces the complexity of the environment.
- In addition, there are different commercial tools. Among those are **HP's Operations Manager**, **IBM Tivoli**, **CA Opscenter** and **BMC Remedy**. These tools are very comprehensive, have been on the market for a long time and offer support for many different software and hardware products. Such platforms are often used enterprise-wide and introducing them into an organization is usually a very complex project. Some of these solutions can also analyze and monitor log files. Due to their large number and the high dynamics of the environment it can be sensible for Microservices to establish their own monitoring tools even if an enterprise-wide standard exists already. When the established processes and tools require a high manual expenditure for administration, this expenditure might not be feasible anymore in the face of the large number of Microservices and the dynamics of the Microservice environment.
- Monitoring can be moved to the Cloud. In this manner no extra infrastructure has to be installed. This facilitates the introduction of tools and monitoring the applications. An example is NewRelic.

These tools are first of all useful for operations and for developers. Business monitoring can be performed with different tools. Such monitoring is not only based on current trends and data, but also on historical values. Therefore, the amount of data is markedly larger than for operations and development. The data can be exported into a separate database or investigated with Big Data solutions. In fact, the analysis of data from web servers is one of the areas where big data solutions have first been used.

Enabling Monitoring in Microservices

Microservices have to deliver data which are displayed in the monitoring solutions. It is possible to provide the data via a simple interface like HTTP with a data format such as JSON. Then the monitoring tools can read these data out and import them. For this purpose, adaptors can be written as scripts by the

developers. This makes it possible to provide different tools via the same interface with data.

Metrics

In the Java world the metrics framework can be used. It offers functionalities for recording custom values and sending them to a monitoring tool. This makes it possible to record metrics in the application and to hand them over to a monitoring tool.

StatsD

StatsD can collect values from different sources, perform calculations and hand over the results to monitoring tools. This allows to condense data before they are passed on to the monitoring tool in order to reduce the load on the monitoring tool. There are also many client libraries for StatsD which facilitate the sending of data to StatsD.

collectd

collectd collects statistics about a system – like for instance the CPU utilization. These data can be analyzed with the frontend or they can be stored in monitoring tools. collectd can collect data from a HTTP JSON data source and send them on to the monitoring tool. Via different plug-ins collectd can collect data from the operating system and the basic processes.

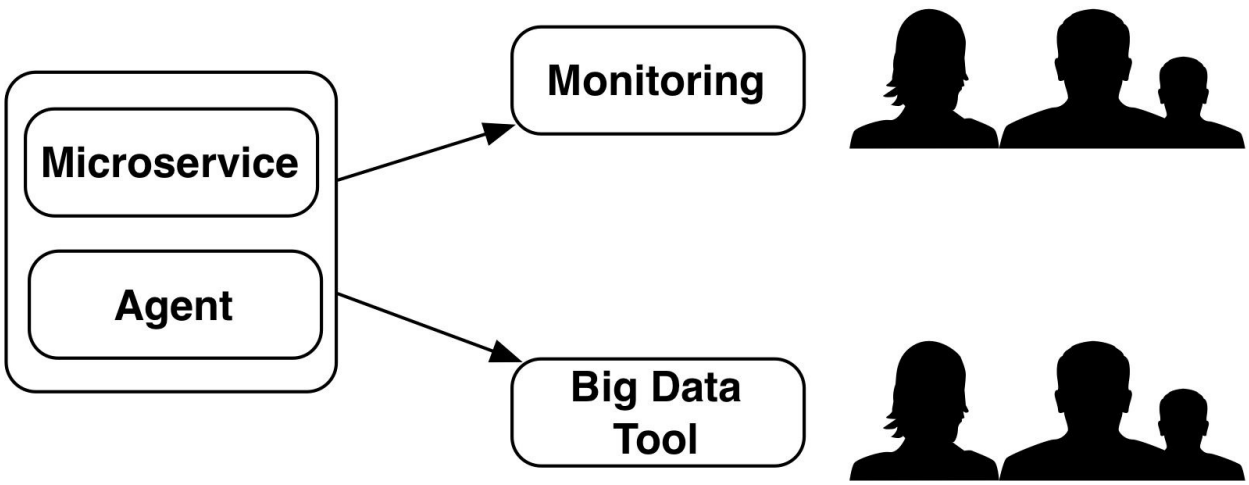


Fig. 60: Parts of a monitoring system

Technology Stack for Monitoring

A technology stack for monitoring comprises different components (Fig. 60):

- Within the Microservice itself data have to be recorded and provided to monitoring. For this purpose, a library can be used which directly contacts the monitoring tool. Alternatively, the data can be offered via a uniform interface – for example JSON via HTTP –, and another tool collects the data and sends them on to the monitoring tool.
- In addition, if necessary, there should be an agent to record the data from the operating system and the hardware and pass them on to monitoring.
- The monitoring tool stores and visualizes the data and can, if needed, trigger an alarm. Different aspects can be covered by different monitoring applications.
- For analyses of historical data or by complex algorithms a solution based on Big Data tools can be created in parallel.

Effects on the Individual Microservice

A Microservice also has to be integrated into the infrastructure. It has to hand over monitoring data to the monitoring infrastructure and provide some mandatory data. This can be ensured by a suitable template for the Microservice and by tests.

Try and Experiment

Define a technology stack which allows to implement monitoring in a Microservice-based architecture. To do so define the stakeholders and the data that are relevant for them. Each of the stakeholders needs to have a tool for analyzing the data that are relevant for him/her. Finally, it has to be defined with which tools the data can be recorded and how they are stored. This section listed a number of tools for the different areas. In conjunction with further research it is possible to assemble a technology stack that is well suited for individual projects.

Chapter 14 shows an example for a Microservice-based architecture, and in section 14.14 there is also a suggestion how the architecture can be extended by monitoring.

How does your current project handle monitoring? Can some of the technologies presented in this section also be advantageous for your project? Which? Why?

12.4 Deployment

Independent deployment is a central aim of Microservices. Besides, the deployment has to be automated because manual deployment or even just manual

corrections are not feasible due to the large number of Microservices.

Deployment Automation

There are different possibilities for automating deployment:

- **Installation scripts** can be used which only install the software on the computer. Such scripts can for instance be implemented as shell scripts. They can install necessary software packages, generate configuration files and create user accounts. Such scripts can be problematic when they are called repeatedly. In that case the installation finds a computer on which the software is already installed. However, an update is different from a fresh installation. In such a situation a script can fail for example because user accounts or configuration files might already be present and cannot easily be overwritten. When the scripts are supposed to handle updates, development and testing the scripts get more laborious.
- **Immutable Servers** are an option to handle these problems. Instead of updating the software on the servers, the server are completely deployed anew. This does not only facilitate the automation of deployment, but also the exact reproduction of the software installed on a server. It is sufficient to consider fresh installations. A fresh installation is easier to reproduce than an update, that can be started from many different configuration states and should lead to the same state from any of those. Approaches like **Docker** make it possible to tremendously reduce the expenditure for installing software. Docker is a kind of light-weight virtualization. It also optimizes the handling of virtual hard drives. If there is already a virtual hard drive with the correct data, it is recycled instead of installing the software anew. When installing a package like Java, first a virtual hard drive is looked for which already has this installation. Only when such a one does not exist, the installation is really performed. Should there only be a change in a configuration file when going from an old to a new version of an Immutable Server, Docker will recycle the old virtual hard drives behind the scenes and only supplement the new configuration file. This does not only reduce the consumption of hard drive space, but also profoundly speeds up the installation of the servers. Docker also decreases the time a virtual team needs for booting. These optimizations turn Immutable Server in conjunction with Docker into an interesting option. The new deployment of the servers is very fast with Docker, and the new server can also rapidly be booted.
- Another possibility are tools like **Puppet**, **Chef**, **Ansible** or **Salt**. They are specialized for installing software. Scripts for these tools describe what the

system is supposed to look like after the installation. During an installation run the tool will take the necessary steps to transfer the system into the desired state. During the first run on a fresh system the tool completely installs the software. If the installation is run a second time immediately afterwards, it will not change the system any further since the system is already in the desired state. Besides these tools can uniformly install a large number of servers in an automated manner and are also able to roll out changes to a large number of servers.

- Operating systems from the Linux area possess package manager like **rpm** (RedHat), **dpkg** (Debian/Ubuntu) or **zypper** (SuSE). They make it possible to centrally roll out software onto a large number of servers. The used file formats are very simple so that it is very easy to generate a package in a fitting format. The configuration of the software poses a problem though. Package managers usually support scripts which are executed during installation. Such scripts can generate the necessary configuration files. However, there can also be an extra package with the individual configurations for each host. The installation tools mentioned under the last bullet point can also use package manager for installing the actual software so that they themselves only generate the configuration files.

Installation and Configuration

Section 8.8 already described tools which can be used for configuring Microservices. In general, it is hard to separate the installation from the software configuration. The installation has to generate a configuration. Therefore, many of the tools like for instance Puppet, Chef, Ansible or Salt can also create configurations and roll them out onto servers. Thus these solutions are an alternative to the configuration solutions which are specialized for Microservices.

Risks Associated with Microservice Deployments

Microservices are supposed to allow for an easy and independent deployment. Nevertheless, it can never be excluded that problems arise in production. The Microservice-based architecture by itself will already help to reduce the risk. When a Microservice fails as result of a problem with a new version, this failure should be limited to the functionality of this Microservice. Apart from that the system should keep working. This is made possible by stability patterns and resilience described in section 10.5. Already for this reason the deployment of a Microservice is much less risky than the deployment of a monolith. In case of a monolith it is much harder to limit a failure to a certain functionality. If a new version of the Deployment Monolith has a memory leak, this will cause the entire

process to break down so that the entire monolith will not be available anymore. A memory leak in a Microservice only influences this Microservice. There are different challenges for which Microservices are not per se helpful: Schema changes in relational databases are for instance problematic because they often take very long and might fail – especially when the database is already containing a lot of data. As Microservices have their own data storage, a schema migration is always limited to just one Microservice.

Deployment Strategies

To further reduce the risk associated with a Microservice deployment there are different strategies:

- A **Rollback** brings the old version of a Microservice back into production. Handling the database can be problematic: Often the old version of the Microservice does not work anymore with the database schema created by the newer version. When there are already data in the database which use the new schema, it can get very difficult to recreate the old state without losing the new data. Besides the rollback is hard to test.
- A **Roll Forward** brings a new version of a Microservice in production, which does not contain the error anymore. The procedure is identical to the procedure for the deployment of any other new version of the Microservice so that no special measures are necessary. The change is rather small so that deployment and the passage through the Continuous Delivery Pipeline should rapidly take place.
- **Continuous Deployment** is even more radical: Each change to a Microservice is brought into production when the Continuous Delivery Pipeline was passed successfully. This further reduces the time necessary for the correction of errors. Besides, this entails that there are less changes per release which further decreases the risk and makes it easier to track which changes to the code caused a problem. Continuous Deployment is the logical consequence when the deployment process works so well that going into production is just a formality. Moreover, the team will pay more attention to the quality of their code when each change really goes into production.
- A **Blue/Green Deployment** builds up a completely new environment with the new version of a Microservice. The team can completely test the new version and then bring it into production. Should problems occur, the old version can be used again which is kept for this purpose. Also in this scenario there are challenges in case of changes to the database schema. When switching from the one version to the other version of the

Microservice, also the database has to be switched. Data which have been written into the old database between the built-up of the new environment and the switch have to be transferred into the new database.

- **Canary Releasing** is based on the idea to deploy the new version initially just on one server in a cluster. When the new version runs without trouble on one server, it can also be deployed on the other servers. The database has to support the old and the new version of the Microservice in parallel.
- **Microservices** can also run blindly in production. In that case they get all requests, but they may not change data, and calls which they send out are not passed on. By monitoring, log analyses and comparison with the old version it is possible to determine whether the new service has been correctly implemented.

Theoretically, such procedures can also be implemented with Deployment Monoliths. However, in practise this is very difficult. With Microservices it is easier since they are much smaller deployment units. Microservices require less comprehensive tests. Installing and starting Microservices is much faster. Therefore, Microservices can more rapidly pass through the Continuous Delivery Pipeline into production. This will have positive effects for Roll Forward or Rollback because problems require less time to fix. A Microservice needs less resources in operation. This is helpful for Canary Releasing or Blue/Green Deployment since new environments have to be built up. If this is possible with less resources, these approaches are easier to implement. For a Deployment Monolith it is often very difficult to build up an environment at all.

Combined or Separate Deployment? (Jörg Müller)

by Jörg Müller, Hypoport AG

The question whether different services are rolled out together or independently from each other is of greater relevance than sometimes suspected. This is an experience we had to make in the context of a project which started approximately five years ago.

The term Microservices was not yet important in our industry. However, achieving a good modularization was our goal right from the start. The entire application consisted initially of a number of web modules coming in the shape of typical Java web application archives (WAR). These comprised in turn multiple modules which had been split based on domain as well as technical criteria. In

addition to modularization we relied from the start on on Continuous Deployment as a method for rolling out the application. Each commit goes straight into production.

Initially, it seemed an obvious choice to build an integrated deployment pipeline for the entire application. This enabled integration tests across all components. A single version for the entire application enabled controlled behavior, even if multiple components of the applications were changed simultaneously. Finally, the pipeline itself was easier to implement. The latter was an important reason since there were relatively few tools for continuous deployment at the time so that we had to build most ourselves.

However, after some time the disadvantages of our approach became obvious. The first consequence was a longer and longer run time of our deployment pipeline. The larger the number of components that were built, tested and rolled out, the longer the process took. The advantages of continuous deployments rapidly diminished when the run time of the pipeline became longer. The first counter measure was the optimization that only changed components were built and tested. However, this increased the complexity of the deployment pipeline tremendously. At the same time other problems like the runtime for changes to central components or the size of the artifacts could not be improved this way.

But there was also a more subtle problem. A combined rollout with integrative tests offered a strong security net. It was easy to perform refactorings across multiple modules. However, this often changed interfaces between modules just because it was so easy to do. This is in principle a good thing. However, it had the consequence that it became very frequently necessary to start the entire system. Especially when working on the developer machine this turned into a burden. The requirements for the hardware got very high and the turnaround times lengthened considerably.

The approach got even more complicated when more than one team worked with this integrated pipeline. The more components were tested in one pipeline, the more frequently errors were uncovered. This blocked the pipeline since the errors had to be fixed first. At the time when only one team was dependent on the pipeline, it was easy to find somebody who took over responsibility and fixed the problem. When there were several teams this responsibility was not so clear anymore. This entailed that errors in the pipeline persisted for a longer time. Simultaneously the variety of technologies increased. Again the complexity rose.

This pipeline now needed very specialized solutions. Therefore, the expenditure for maintenance increased, and the stability decreased. The value of continuous deployment got hard to put into effect.

At this time point it became obvious that the combined deployment in one pipeline could not be continued anymore. All new services, regardless whether Microservices or larger modules, now had there own pipeline. However, it caused a lot of expenditure to separate the previous pipeline which was based on shared deployment into multiple pipelines.

In a new project it can be the right decision to start with a combined deployment. This especially holds true when the borders between the individual services and their interfaces are not yet well known. In such a case good integrative tests and simple refactoring can be very useful. However, starting at a certain size an independent deployment is obligatory. Indications for this are the number of modules or services, the run time and stability of the deployment pipeline and last, but not least the question how many teams work on the overall system. If these indications are overlooked and the right point in time to separate the deployment is missed, it can easily happen that one builds a monolith which consists of many small Microservices.

12.5 Control

Interventions in a Microservice might be necessary at run time. For instance, a problem with a Microservice might require to restart the respective Microservice. Likewise, a start or a stop of a Microservice might be necessary. These are ways for operation to intervene in case of a problem or for a load balancer to terminate instances which cannot process requests anymore.

Different measures can be used for control:

- When a Microservice runs in a **virtual machine**, the virtual machine can be shut down or restarted. In that case the Microservice itself does not have to make special arrangements.
- The operating system supports **services** which are started together with the operating system. Usually, services can also be stopped, started or restarted by means of the operating system. In that case the installation only has to register the Microservice as service. Working with services is nothing unusual for operation which is sufficient for this approach.

- Finally, an **interface** can be used which allows restarting or shutting down, for instance via REST. Such an interface has to be implemented by the Microservice itself. This is supported by several libraries in the Microservices area – for instance by Spring Boot which is used to implement the example in chapter 14. Such an interface can be called with simple HTTP tools like curl.

Technically, the implementation of control mechanisms is not a big problem, but they have to be present for operating the Microservices. When they are identically implemented for all Microservices, this can reduce the expenditure for operating the system.

12.6 Infrastructure

Microservices have to run on a suitable platform. It is best to run each Microservice in a separate virtual machine (VM). Otherwise it is difficult to assure an independent deployment of the individual Microservices.

When multiple Microservices run on a virtual machine, the deployment of one Microservice can influence another Microservice. The deployment can generate a high load or introduce changes to the virtual machine which also concern other Microservices running on the virtual machine.

Besides Microservices should be isolated from each other to achieve a better stability and resilience. When multiple Microservices are running on one virtual machine, one Microservice can generate so much load that the other Microservices fail. However, precisely that should be prevented: When one Microservice fails, this failure should be limited to this one Microservice and not affect additional Microservices. The isolation of virtual machines is helpful for limiting the failure or the load to one Microservice.

Scaling Microservices is likewise easier when each Microservice runs in an individual virtual machine. When the load is too high, it is sufficient to start a new virtual machine and register it with the load balancer.

In case of problems it is also easier to analyze the error when all processes on a virtual machine belong to one Microservice. Each metric on the system then unambiguously belongs to this Microservice.

Finally, the Microservice can be delivered as hard drive image when each Microservice runs on its own virtual machine. Such a deployment has the advantage that the entire environment of the virtual machine is exactly in line with the requirements of the Microservice and that the Microservice can bring along its own technology stack up to its own operating system.

Virtualization or Cloud

It is hardly possible to install new physical hardware upon the deployment of a new Microservice. Besides Microservices profit from virtualization or Cloud since this renders the infrastructures much more flexibel. New virtual machines for scaling or testing environments can easily be provided. In the Continuous Delivery Pipeline Microservices are constantly started to perform different tests. Moreover, in production new instances have to be started depending on the load.

Therefore it should be possible to start a new virtual machine in a completely automated manner. Starting new instances with simple API calls is exactly what a Cloud offers. A Cloud infrastructure should be available in order to really be able to implement a Microservice-based architecture. Virtual machines which are provided by operation via manual processes are not sufficient. This also demonstrates that Microservices can hardly be run without modern infrastructures.

Docker

When there is an individual virtual machine for each Microservice, it is laborious to generate a test environment containing all Microservices. Even creating an environment with relatively few Microservices can be a challenge for a developer machine. The usage of RAM and CPU is very high for such an environment. In fact, it is hardly sensible to use an entire virtual machine for one Microservice. In the end, the Microservice should just run and integrate in logging and monitoring. Therefore solutions like Docker are convenient: Docker does not comprise many of the normally common operating system features.

Instead Docker offers a very light-weight virtualization. To this purpose Docker uses different technologies:

- In place of a complete virtualization Docker employs Linux Containers (LXC – LinuX Container). Support for similar mechanisms in Microsoft Windows has been announced. This allows to implement a light-weight alternative to virtual machines: All containers use the same kernel. There is only one instance of the kernel in memory. Processes, networks, data systems and

users are separate from each other. In comparison to a virtual machine with its own kernel and often also many operating system services a container has a profoundly lower overhead. It is easily possible to run hundreds of Linux containers on a simple laptop. Besides a container starts much more rapidly than a virtual machine with its own kernel and complete operating system. The container does not have to boot an entire operating system; it just starts a new process. The container itself does not add a lot of overhead since it only requires a custom configuration of the operating system resources.

- In addition, the file system is optimized: Basic read-only file systems can be used. At the same time additional file systems can be added to the container which also allow for writing. One file system can be put on top of another file system. For instance, a basic file system can be generated which contains an operating system. If software is installed in the running container or if files are modified, the container only has to store these additional files in a small container-specific file system. In this way the memory requirement for the containers on the hard drive is significantly reduced.

Besides additional interesting possibilities arise: For example, a basic file system can be started with an operating system, and subsequently software can be installed. As mentioned, only changes to the file system are saved which are introduced upon the installation of the software. Based on this delta a file system can be generated. Then a container can be started which puts a file system with this delta on top of the basic file system containing the operating system – and afterwards additional software can be installed in yet another layer. In this manner each “layer” in the file system can contain specific changes. The real file system at run time can be composed from numerous such layers. This allows to recycle software installations very efficiently.

Fig. 61 shows an example for the file system of a running container: The lowest level is an Ubuntu Linux installation. On top there are changes which have been introduced by installing Java. Then there is the application. For the running container to be able to write changes into the file system, there is a file system on top into which the container writes files. When the container wants to read a file, it will move through the layers from top to bottom until it finds the respective data.

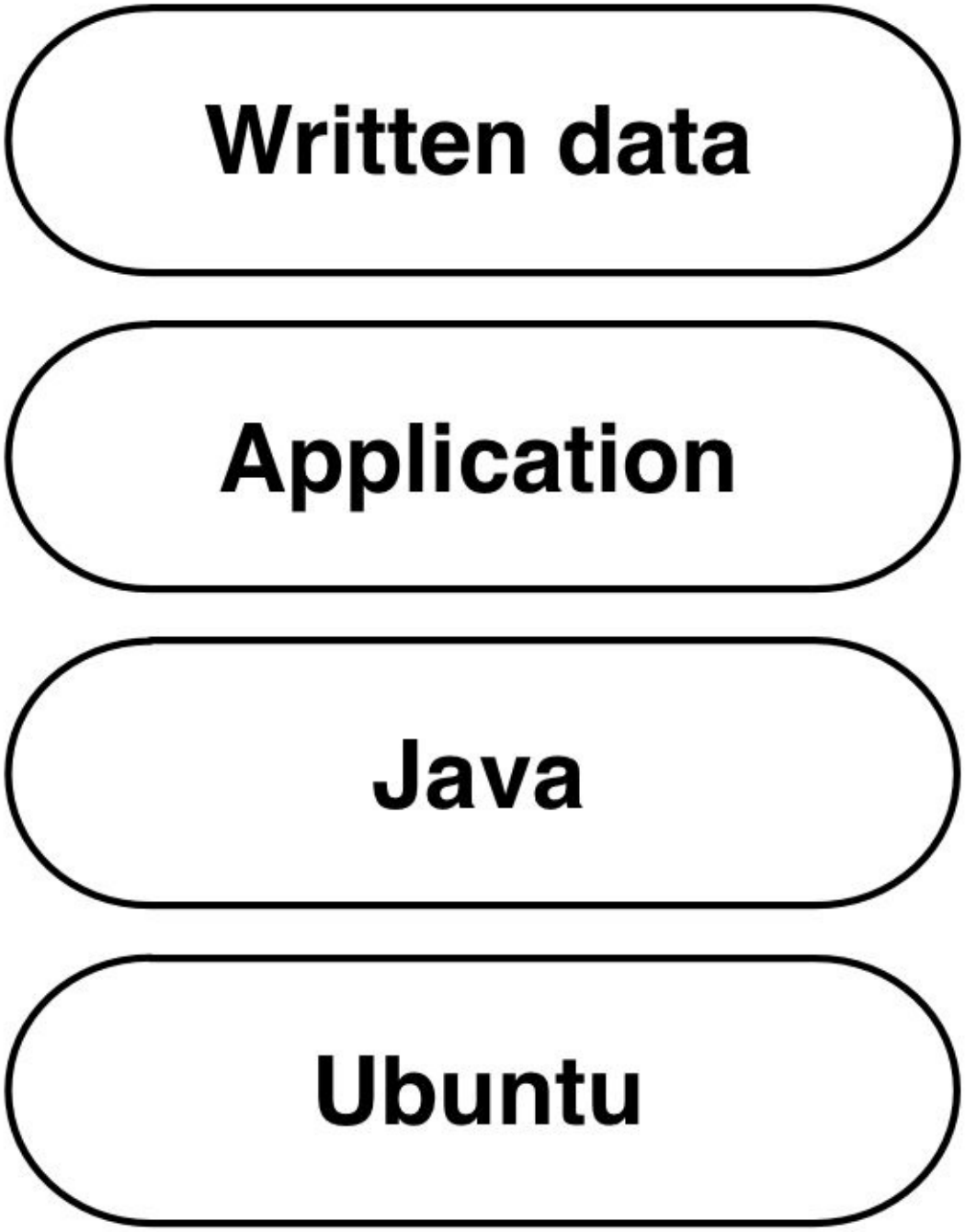


Fig. 61: Filesystems in Docker

Docker Container vs. Virtualization

Docker containers offer a very efficient alternative to virtualization. However, they are no “real” virtualization since each container has separate resources, its own memory, and its own file systems, but all share for instance one kernel. Therefore, this approach has some disadvantages. A Docker container can only use Linux and only the same kernel like the host operating system – consequently

Windows applications for instance cannot be run on a Linux machine this way. The separation of the containers is not as strict as in the case of real virtual machines. An error in the kernel would for example affect all containers. Moreover, Docker also does not run on Mac OS X or Windows. Nevertheless, Docker can directly be installed on these platforms. Behind the scenes a virtual machine with Linux is being used. Microsoft has announced a version for Windows which can run the Windows container.

Communication Between Docker Containers

Docker containers have to communicate with each other. For example, a web application communicates with its database. For this purpose, containers export network ports which other containers use. Besides file systems can be used together. There containers write data which can be read by other containers.

Docker Registry

Docker images comprise the data of a virtual hard drive. Docker registries allow to save and download Docker images. This makes it possible to save Docker images as result of a build process and subsequently to roll them out on servers. Because of the efficient storage of images, it is easily possible to distribute even complex installations in a performant manner. Besides many Cloud solutions can directly run Docker containers.

Docker and Microservices

Docker constitutes an ideal running environment for Microservices. It hardly limits the used technology as every type of Linux software can run in a Docker container. Docker registries allow to easily distribute Docker containers. At the same time the overhead of a Docker container is negligible in comparison to a normal process. Since Microservices require a multitude of virtual machines, these optimizations are very valuable. On the one hand Docker is very efficient, and on the other hand it does not limit the technology freedom.

Try and Experiment

At http://www.docker.com/tryit/ the Docker online tutorial can be found. Complete the tutorial – it demonstrates the basics of working with Docker. The tutorial is fast to complete.

Docker and Servers

There are different possibilities to use Docker for servers:

- On a **Linux server** Docker can be installed, and afterwards one or multiple Docker containers can be run. Docker then serves as solution for the provisioning of the software. For a cluster new servers are started on which again the Docker containers are installed. Docker only serves for the installation of the software on the servers.
- Docker containers are run directly on a **cluster**. On which physical computer a certain Docker is located is decided by the software for cluster administration. Such an approach is supported by the scheduler Apache Mesos. It administrates a cluster of servers and directs jobs to the respective servers. Mesosphere allows to run Docker containers with the aid of the Mesos scheduler. Besides Mesos supports many additional kinds of jobs.
- Kubernetes likewise supports the execution of Docker containers in a cluster. However, the approach taken is different from Mesos. Kubernetes offers a service which distributes pods in the cluster. Pods are interconnected Docker containers which are supposed to run on a physical server. As basis Kubernetes requires only a simple operating system installation – Kubernetes implements the cluster management.
- CoreOS is a very light-weight server operating system. With etcd it supports the cluster-wide distribution of configurations. fleetd enables the deployment of services in a cluster – up to redundant installation, failure security, dependencies and shared deployment on a node. All services have to be deployed as Docker containers while the operating system itself remains essentially unchanged.
- Docker Machine allows the installation of Docker on different virtualization and Cloud systems. Besides Docker machine can configure the Docker command line tool in such a manner that it communicates with such a system. Together with Docker Compose multiple Docker containers can be combined to an overall system. The example application employs this approach, compare section 14.6 and section 14.7. Docker Swarm adds a way to configure and run clusters with this tool stack: Individual servers can be installed with Docker Machine and combined to a cluster with Docker Swarm. Docker Compose can run each Docker container on a specific machine in the cluster.

Kubernetes, CoreOS, Docker Compose, Docker Machine, Docker Swarm and Mesos of course influence the running of the software so that the solutions require changes in the operation procedures in contrast to virtualization. These

technologies solve challenges which were previously addressed by virtualization solutions, namely to administrate a cluster of servers so that containers resp. virtual machines can be distributed in the cluster.

PaaS

PaaS (Platform as a Service) is based on a fundamentally different approach. The deployment of an application can be done simply by updating the application in version control. The PaaS fetches the changes, builds the application and rolls it out on the servers. These servers are installed by PaaS and represent a standardized environment. The actual infrastructure – i.e. the virtual machines – are hidden from the application. PaaS offers a standardized environment for the application. The environment takes for instance also care of the scaling and can offer services like databases and messaging systems. Because of the uniform platform PaaS systems limit the technology freedom which is normally an advantage of Microservices. Only technologies which are supported by PaaS can be used. On the other hand, deployment and scaling are further facilitated.

Microservices impose high demands on infrastructure. Automation is an essential prerequisite for operating the numerous Microservices. A PaaS offers a good basis for this since it profoundly facilitates automation. To use a PaaS can be especially sensible when the development of a home-grown automation is too laborious and there is not enough knowledge about how to build the necessary infrastructure. However, the Microservices have to restrict themselves to the features which are offered by the PaaS. When the Microservices have been developed for the PaaS from the start, this is not very laborious. However, if they have to be ported, considerable expenditure can ensue.

Nanoservices (chapter 15) have different operating environments, which for example even further restrict the technology choice. On the other hand they are often even easier to operate and even more efficient in regards to resource usage.

12.7 Conclusion

Operating a Microservice-based system is one of the central challenges when working with Microservices (section 12.1). A Microservice-based system contains a tremendous number of Microservices and therefore operating system processes. Fifty or one hundred virtual machines are no rarity. The responsibility for operation can be delegated to the teams. However, this approach creates a higher overall expenditure. Standardizing operations is a more sensible strategy.

Templates are a possibility to achieve uniformity without exerting pressure. Templates turn the uniform approach into the easiest one.

For logging (section 12.2) a central infrastructure has to be provided which collects logs from all Microservices. There are different technologies available for this. To trace a call across the different Microservices a Correlation ID can be used which unambiguously identifies a call.

Monitoring (section 12.3) has to offer at least basic information such as the availability of the Microservice. Additional metrics can for instance provide an overview of the overall system or can be useful for load balancing. Metrics can be individually defined for each Microservice. There are different stakeholders for the monitoring: Operations, developers and business stakeholders. They are interested in different values and use where necessary their own tools for evaluating the Microservices data. Each Microservice has to offer an interface with which the different tools can fetch values from the application. The interface should be identical for all Microservices.

The deployment of Microservices (section 12.4) has to be automated. Simple scripts, especially in conjunction with Immutable Server, special deployment tools and Package Manager can be used for this purpose.

Microservices are small deployment units. They are safeguarded by stability and resilience against the failure of other Microservices. Therefore, the risk associated with deployments is already reduced by the Microservice-based architecture itself. Strategies like Rollback, Roll Forward, Continuous Deployment, Blue/Green-Deployment or a blind moving along in production can further reduce the risk. Such strategies are easy to implement with Microservices since the deployment units are small and the consumption of resources by Microservices is low. Therefore, deployments are faster, and environments for Blue/Green-Deployment or Canary Releasing are much easier to provide.

Control (section 12.5) comprises simple intervention options like starting, stopping and restarting of Microservices.

Virtualization or Cloud are good options for infrastructures for Microservices (section 12.6). On each VM only a single Microservice should run to achieve a better isolation, stability and scaling. Especially interesting is Docker because the consumption of resources by a Docker Container is much lower than the one of a

VM. This makes it possible to provide each Microservice with its own Docker Container even if the number of Microservices is large. PaaS are likewise interesting. They allow for a very simple automation. However, they also restrict the choice of technologies.

This section only focuses on the specifics of Continuous Delivery and operation in a Microservices environment. Continuous Delivery is one of the most important reasons for the introduction of Microservices. At the same time operation poses the biggest challenges.

Essential Points

- Operation and Continuous Delivery are central challenges for Microservices.
- The Microservices should handle monitoring, logging and deployment in a uniform manner. This is the only way to keep the effort reasonable.
- Virtualization, Cloud, PaaS and Docker are interesting infrastructure alternatives for Microservices.

13 Organizational Effects of a Microservices-based Architecture

It is an essential feature of the Microservice-based approach that one team is responsible for each Microservice. Therefore, when working with Microservices, it is necessary to look not only at the architecture, but also at the organization of teams and the responsibilities for the individual Microservices. This chapter discusses the organizational effects of Microservices.

In section 13.1 organizational advantages of Microservices are described. Section 13.2 shows that collective code ownership presents an alternative to devising teams according to Conway's Law. The independence of the teams is an important consequence of Microservices. Section 13.3 defines micro and macro architecture and shows how these approaches offer a high degree of autonomy to the teams and let them make independent decisions. Closely connected is the question about the role of the technical leadership (section 13.4). DevOps is an organizational approach which combines development (Dev) and operations (Ops) (section 13.5). DevOps has synergies with Microservices. Since Microservices focus on independent development from a domain perspective, they influence also product owners and business stakeholders e.g. the departments of the business that uses the software. Section 13.6 discusses how these groups can handle Microservices. Reusable code can only be achieved in Microservice systems via organizational measures as illustrated in section 13.7. Finally, section 13.8 follows up on the question whether an introduction of Microservices is possible without changing the organization.

13.1 Organizational Benefits of Microservices

Microservices are an approach for tackling also large projects with small teams. As the teams are independent of each other, less coordination is necessary between them. Especially the communication overhead renders the work of large teams so inefficient. Microservices are an approach on the architectural level for solving this problem. The architecture helps to reduce the need for communication and to let many small teams work in the project instead of one large one. Each

domain-based team can have the ideal size: The Scrum guide recommends three to nine members.

Besides, modern enterprises stress self organization and teams which are themselves active directly at the market. Microservices support this approach because each service is in the responsibility of an individual team consistent with Conway's Law (Section 4.2). Therefore Microservices fit well to self organization. Each team can implement new features independently of other teams and can evaluate the success on the market by themselves.

On the other hand there is a conflict between independence and standardization: When the teams are supposed to work on their own, they have to be independent. Standardization restricts independence. This concerns for instance the decision which technologies should be used. If the project is standardized in regards to a certain technology stack, the teams cannot decide independently anymore which technology they want to use. In addition, independence conflicts with the wish to avoid redundancy: If the system is supposed to be free of redundancy, there has to be coordination between the teams in order to identify the redundancies and to eliminate them. This in turn limits the independence of the teams.

Technical Independence

An important aspect is the technological decoupling. Microservices can use different technologies and can have entirely different structures internally. This means that developers have less need to coordinate. Only fundamental decisions have to be made together. All other technical decisions can be made by the teams.

Separate Deployment

Each Microservice can be brought into production independently of the other Microservices. There is also no need to coordinate release dates or test phases across teams. Each team can choose its own speed and its own dates. A delayed release date of one team does not influence the other teams.

Separate Requirement Streams

The teams should each implement independent stories and requirements. This allows each team to pursue its own business objectives.

Three Levels of Independence

Microservices enable independence on three levels:

- Decoupling via independent releases: Each team takes care of one or multiple Microservices. The team can bring them into production independently of the other teams and the other Microservices.
- Technological decoupling: The technical decisions made by a certain team concern first of all their Microservices and none of the other Microservices.
- Domain-based decoupling: The distribution of the domain in separate components allows each team to implement their own requirements.

For Deployment Monoliths, in contrast, the technical coordination and deployment concerns the entire monolith (Fig. 62). This necessitates such a close coordination between the developers that in the end all developers working on the monolith have to act like one team.

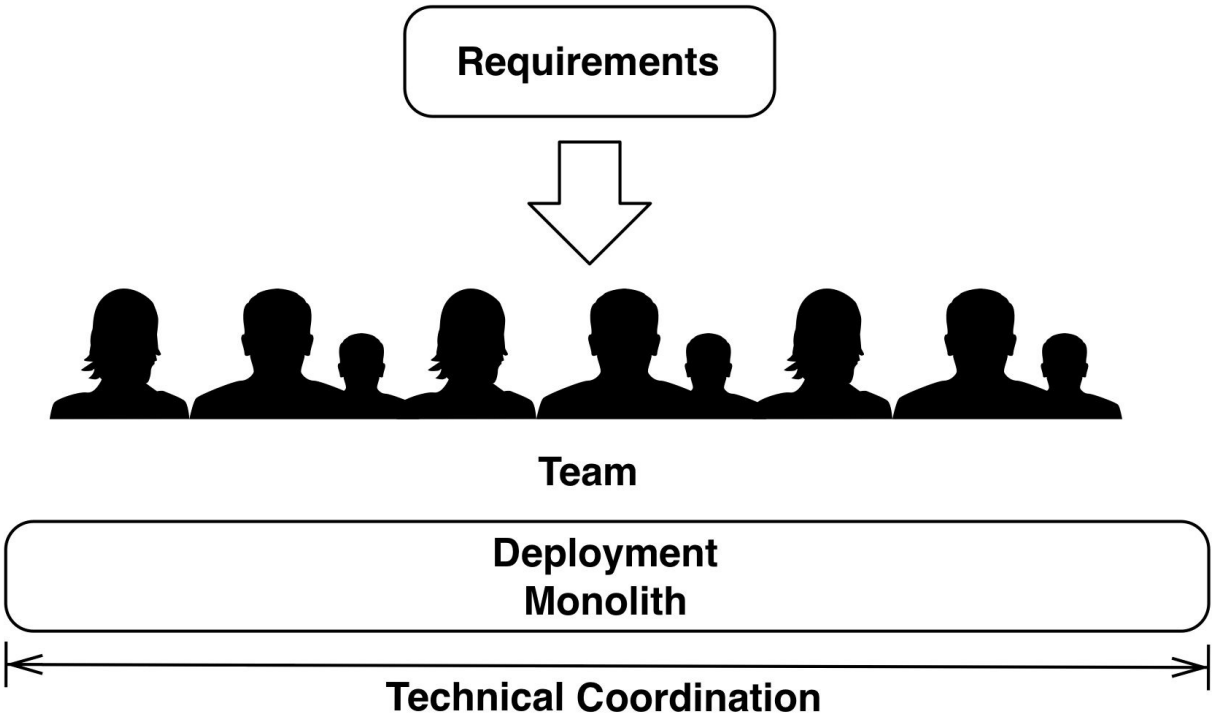


Fig. 62: Deployment Monolith

A prerequisite for the independence of the Microservice teams is that the architecture really offers the necessary independence of the Microservices. This requires first of all a good domain architecture. This architecture enables also independent requirement streams for each team.

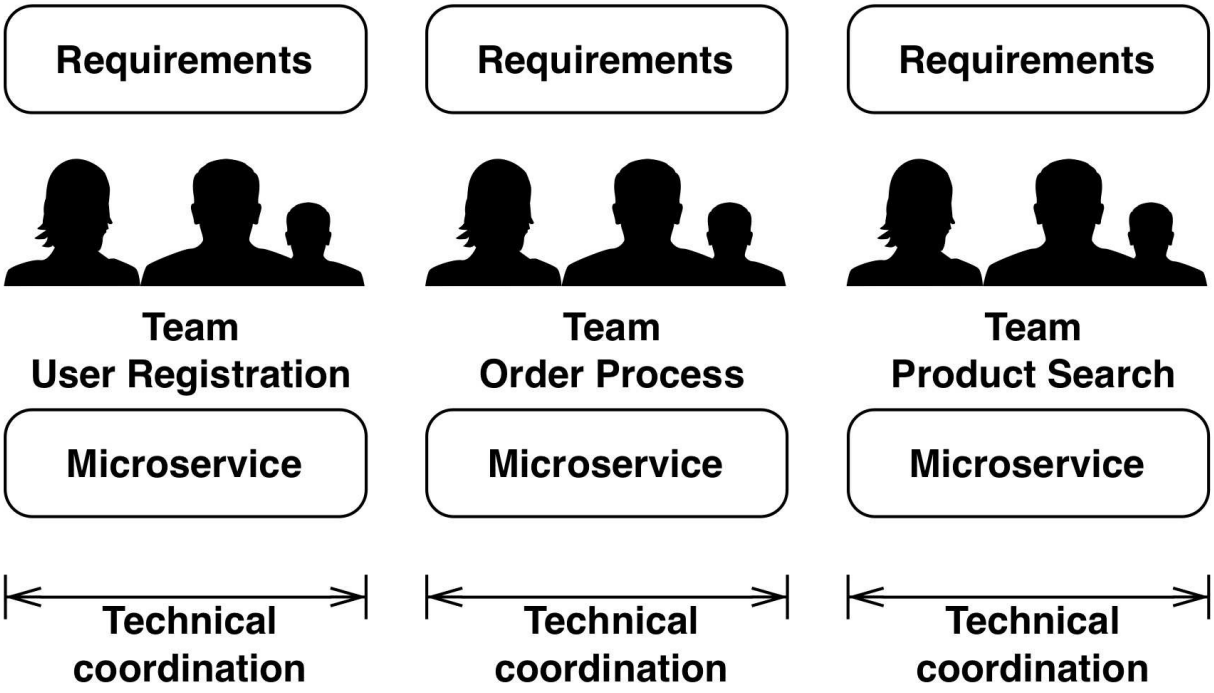


Fig. 63: Separation into Microservices

There are the following teams in the example from Fig. 63:

- The team “user registration” takes care of how users can register in the E-commerce shop. A possible business objective is to achieve a high number of registrations. New features aim at optimizing this number. The components of the team are the processes which are necessary for the registration and the UI elements. The team can change and optimize them at will.
- The team “order process” addresses how the shopping cart turns into an order. Here, a possible objective is that as many shopping carts as possible turn into orders. The entire process is implemented by this team.
- The team “product search” improves the search for products. The success of this team depends on how many search processes lead to items being put into a shopping cart.

Of course, there can be additional teams with other goals. Overall this approach distributes the task of developing an E-commerce shop onto multiple teams which all have their own objectives. The teams can largely independently pursue their objectives because the architecture of the system is distributed into Microservices which each team can develop independently – without much need for coordination.

In addition small projects have many more advantages:

- Estimations are more accurate since estimates concerning smaller efforts are easier to make.
- Small projects are better to plan.
- The risk decreases – because of the more accurate estimates and because of the better forecast reliability.
- If there still is a problem, its effects are smaller because the project is smaller.

In addition, Microservices offer much more flexibility. This makes decisions faster and easier because the risk is smaller and changes can be implemented more rapidly. This ideally supports agile software development which relies on such flexibility.

13.2 An Alternative Approach to Conway's Law

Section 4.2 introduced Conway's Law. According to this law, an organization can only generate architectures which mirror its communication structures. In Microservice-based architectures the teams are built according to the Microservices. Each team develops one or multiple Microservices. Thus each Microservice is only developed by exactly one team. This ensures that the domain architecture is not only implemented by the distribution into Microservices, but also supported by the organizational distribution. This renders violations of the architecture practically impossible. Moreover the teams can independently develop features when the features are limited to one Microservice. For this to work the distribution of domains between the Microservices has to be of very high quality.

The Challenges Associated with Conway's Law

However, this approach also has disadvantages:

- The teams have to remain stable in the long run. Especially when the Microservices use different technologies, the ramp up time for an individual Microservice is very long. Developers cannot easily switch between teams. Especially in teams containing external consultants long term stability is often hard to ensure. Already the usual fluctuation of personnel can turn into a challenge when working with Microservices. In the worst case, if there is nobody left to maintain a specific Microservice, it is still possible to rewrite

the respective Microservice. Microservices are easy to replace due to their limited size. Of course, this still entails some expenditure.

- Only the team understands the component. When team members quit, the knowledge about one or multiple Microservices can get lost. In that case the Microservice cannot be modified anymore. Such islands of knowledge need to be avoided. In such a case it will not be an option to replace the Microservice since an exact knowledge of the domain is necessary for this.
- Changes are difficult whenever they require the coordinated work of multiple teams. When a team can implement all changes for a feature in its own Microservices, architecture and scaling of development will work very well. However, when the feature concerns also another Microservice and therefore another team, the other team needs to implement the changes to the respective Microservice. This requires not only communication, but the necessary changes also have to be prioritized versus the other requirements of the team. If the teams work in sprints, a team can deliver the required changes without prematurely terminating the current sprint earliest in the following sprint – this causes a marked delay. In case of a sprint length of two weeks the delay can amount to two weeks – if the team prioritizes the change high enough so that it is taken care of in the next sprint. Otherwise the ensuing delay can be even longer.

Collective Code Ownership

When it is always only the responsible team which can introduce changes to a Microservice, a number of challenges result as described. Therefore it is worthwhile to consider alternatives. Agile processes have led to the concept of “Collective Code Ownership”. Here, each developer has not only the right, but even the duty to alter any code – for example when he/she considers the code quality as insufficient in a certain place. Thereby all developers take care of code quality. Besides technical decisions are better communicated because more developers understand them due to their reading and changing code. This leads to the critical questioning of decisions so that the overall quality of the system increases.

Collective Code Ownership can relate to a team and its Microservices. Since the teams are relatively free in their organization, such an approach is possible without much coordination.

Advantages of Collective Code Ownership

However, in principle teams can also modify Microservices which belong to other teams. This approach is used by some Microservice projects to deal with the discussed challenges because it entails a number of advantages:

- Changes to a Microservice of another team can be faster and more easily implemented. When a modification is necessary, the change has not to be introduced by another team. Instead the team requiring the change can implement it by itself. It is not necessary anymore to prioritize the change in regards to other changes to the component.
- Teams can be put together more flexibly. The developers are familiar with a larger part of the code – at least superficially due to changes which they have introduced in the code. This makes it easier to replace team members or even an entire team – or to enlarge a team. The developers do not have to ramp up from the very basics. A stable team is still the best option – however, often this cannot be achieved.
- The distribution in Microservices is easy to change. Because of the broader knowledge of the developers it is easier to move responsibility for a Microservice to a different team. This can be sensible when Microservices have a lot of dependencies on each other, but are in the responsibility of different teams which then have to closely and laboriously coordinate. If the responsibility for the Microservices is changed so that the same team is responsible for both of the closely coupled Microservices, coordination is easier than in the case where two teams were working on these Microservices. Within one team the team members often sit in the same office. Therefore they can easily and directly communicate with each other.

Disadvantages of Collective Code Ownership

However, there also disadvantages associated with this approach:

- Collective Code Ownerships is in contrast to technology freedom: When each team uses other technologies, it is difficult for developers outside of a team to change the respective Microservices. They might not even know the technology used in the Microservice.
- The teams can lose their focus. The developers acquire a larger overview of the full system. However, it might be better when the developers concentrate on their own Microservices instead.
- The architecture is not as solid anymore. By knowing the code of other components developers can exploit the internals and thereby rapidly create dependencies which had not been intended in the architecture. Finally, the

distribution of the teams according to Conway's Law is supposed to support the architecture by turning interfaces between domain components into interfaces between teams. However, the interfaces between the teams lose importance when everybody can change the code of every other team.

Pull Requests for Coordination

Communication between teams is still necessary: In the end the team responsible for the respective Microservice has the most knowledge about the Microservice. So changes should be coordinated with the respective team. This can be safeguarded technically: The changes of the external teams can initially be introduced separately from other changes and subsequently be sent to the responsible team via a pull request. Pull requests bundle changes to the source code. Especially in the open source community they are a popular approach to allow for external contributions without giving up control of the project. The responsible team can accept the pull request or demand fixes. This means that there is a review for each change by the responsible team. This allows the responsible team to ensure that the architecture and design of the Microservice remain sound.

Since there is still the need for communication between teams, Conway's Law is not violated by this approach. It is just a different way of playing the game. In case of a bad split among teams in a Microservice-based architecture all options are associated with tremendous disadvantages. To correct the distribution is difficult as larger changes across Microservices are laborious as discussed in section 8.4. Due to the unsuitable distribution the teams are forced to communicate a lot with each other. Thereby productivity is lost. Therefore it is also no option to leave the distribution as it is. Collective Code Ownership can be used to limit the need for communication. The teams directly implement requirements in the code of other teams. This causes less need for communication and better productivity. To do so the technology freedom should be restricted. The changes to the Microservices still have to be coordinated – at least reviews are definitely necessary. However, if the architecture had been set up appropriately from the start, this measure would not be necessary at all as workaround.

Try and Experiment

Did you already encounter Collective Code Ownership? Which experiences did you make with it?

Which restrictions are there in your current project when a developer wants to change some code which has been written by another developer in the same team or by a developer from another team? Are changes to the code of other teams not meant to occur? In that case, how is it still possible to implement the necessary changes? Which problems are associated with this course of action?

13.3 Micro and Macro Architecture

Microservices allow to largely avoid overarching architecture decisions. Each team can choose the optimal type of architecture for its Microservices.

Basis for this is the Microservices architecture. It allows a large degree of technical freedom. While normally due to technical reasons uniform technologies are mandatory, Microservices do not have these restrictions. However, there can be other reasons for uniformity. The question is which decision is made by whom. There are two layers of decision making:

- Macro architecture comprises the decisions which concern the overall system. These are at least the decisions presented in chapter 8 regarding the domain architecture and basic technologies, which have to be used by all Microservices, as well as communication protocols (chapter 9). The properties and technologies of individual Microservices can also be preset (chapter 10). However, this does not have to be the case. Decisions about the internals of the individual Microservices do not have to be made in the macro architecture.
- The micro architecture deals with decisions each team can make by itself. These should address topics which concern only the Microservices developed by the respective team. Among these topics can be all aspects presented in chapter 10 as long as they have not already been defined as part of the macro architecture.

The macro architecture cannot be defined once for all, but has to undergo continuous development. New features can require a different domain architecture

or new technologies. Optimizing the macro architecture is a permanent process.

Decision = Responsibility

The question is who defines macro and micro architecture and takes care of their optimization. It is important to keep in mind that each decision is linked to responsibility. Whoever makes a decision is responsible for its consequences - good or bad. In turn the responsibility for a Microservice entails the necessity to make the required decisions for its architecture. When the macro architecture defines a certain technology stack, the responsibility for this stack rests with the persons responsible for the macro architecture – not with the teams which use them in the Microservices and might later have problems with this technology stack. Therefore a strong restriction of the technology freedom of the individual Microservices by the macro architecture is often not helpful. It only shifts decisions and responsibility to a level which does not have much to do with the individual Microservices. This can lead to an ivory tower architecture that is not based on the real requirements. In the best case it is ignored. In the worst case it causes serious problems in the application. Microservices allow to largely do without macro architecture decisions in order to avoid such an ivory tower architecture.

Who Creates Macro Architecture?

For defining macro architecture decisions have to be made which affect all Microservices. Such decisions cannot be made by a single team since the teams only carry responsibility for their respective Microservices. Macro architecture decisions go beyond individual Microservices.

The macro architecture can be defined by a team which is composed from members of each individual team. This approach seems to be obvious at first glance: It allows all teams to voice their perspectives. Nobody dictates certain approaches. The teams are not left out of the decision process. There are many Microservice projects which very successfully employ this approach.

However, this approach has also disadvantages:

- For decisions at the macro architecture level an overview of the overall system is necessary and an interest to develop the system in its entirety. Members of the individual teams often have a strong focus on their own Microservices. That is of course very sensible since the development of these Microservices is their primary task. However, this can make it hard for

them to make overarching decisions since those require a different perspective.
- The group can be too large. Effective teams normally have five to ten members at maximum. If there are many teams and each is supposed to participate with at least one member, the macro architecture team will get too large and thus cannot work effectively anymore. Large teams are hardly able to define and maintain the macro architecture.

The alternative is to have a single architect or an architecture team which is exclusively responsible for shaping the macro architecture. For larger projects this task is so demanding that for sure an entire architecture team is needed to work on it. This architecture team takes the perspective of the overall project. However, there is a danger that the architecture team distances itself too much from the real work of the other teams and consequently makes ivory-tower decisions or solves problems the teams do not actually have. Therefore, the architecture team should mainly moderate the process of decision making and make sure that the view points of the different teams are all considered. It should not set a certain direction all by itself. In the end the different Microservices teams will have to live with the consequences of the architecture team's decisions.

Extent of the Macro Architecture

There is no one and only way to divide the architecture into micro and macro architecture. The company culture, the degree of self organization and other organizational criteria play a prominent role. A highly hierarchical organization will give the teams less freedom. When as many decisions as possible are made on the level of the micro architecture, the teams will gain more responsibility. This often has positive effects because the teams really feel responsible and will act accordingly.

The NUMMI car factory in the USA for instance was a very unproductive factory which was known for drug abuse and sabotage. By focusing more on teamwork and trust the same workers could be turned into a very productive workforce. When teams are able to make more decisions on their own and have more freedom of choice, the work climate as well as productivity will profoundly benefit.

Besides, by delegating decisions to teams less time is spent on coordination so that the teams can work more productively. To avoid the need for communication by delegating more decisions to the teams and therefore to micro architecture is an essential point for architecture scaling.

However, when the teams are very restricted in their choices, one of the main advantages of Microservices is not realized. Microservices increase the technical complexity of the system. This only makes sense if the advantages of Microservices are really exploited. Consequently, when the decision for Microservices has been made, there should also be a decision for having as much micro architecture and as little macro architecture as possible.

The decision for more or less macro architecture can be made for each area differently.

Technology: Macro/Micro Architecture

For the technologies the following decisions can be made concerning macro vs. micro architecture:

- Uniform security (section 8.12), service discovery (section 8.9) and communication protocols (chapter 9) are necessary to enable Microservices to communicate with each other. Therefore decisions in these areas clearly belong to macro architecture. Among these are also the decisions for the use and details of downwards compatible interfaces which are required for the independent deployment of microservices.
- Configuration and coordination (Section 8.8) do not necessarily have to be determined globally for the complete project. When each Microservice is operated by its respective team, the team can also handle the configuration and use its own tool of choice for it. However, a uniform tool for all Microservices has clear advantages. Besides there is hardly any sensible reason why each team should use a different mechanism.
- The use of resilience (section 10.5) or load balancing (section 8.10) can be defined in the macro architecture. The macro architecture can either define a certain standard technology or just enforce that these points have to be addressed during the implementation of the Microservices. This can for instance be ensured by tests (section 11.8). The tests can check whether a Microservice is still available after a dependent Microservice failed. In addition, they can check whether the load is distributed to multiple Microservices. The decision for the use of resilience or load balancing can be theoretically left to the teams. When they are responsible for the availability and the performance of their service, they have to have the freedom to use their choice of technologies for it. When their Microservices are sufficiently available without resilience and load balancing, their strategy

is acceptable. However, in the real world such scenarios are hard to imagine.

- In regards to platform and programming language the decision can be made at the level of macro or micro architecture. The decision might not only influence the teams but also operations since operations needs to understand the technologies and need to be able to deal with failures. It is not necessarily required to prescribe a programming language. Alternatively, the technology can be restricted e.g. to the JVM (Java Virtual Machine) which supports a number of programming languages. In regards to the platform a potential compromise is that a certain database is provided by operations, but that the teams can also use and operate different ones. Whether the macro architecture defines platform and programming language depends also on whether developers need to be able to change between teams. A shared platform facilitates transferring the responsibility for a Microservice from one team to another team.

Fig. 64 shows which decisions are part of the macro architecture - they are on the right side. The micro architecture parts are on the left side. The areas in the middle can be either part of the macro or micro architecture. Each project can handle them differently.

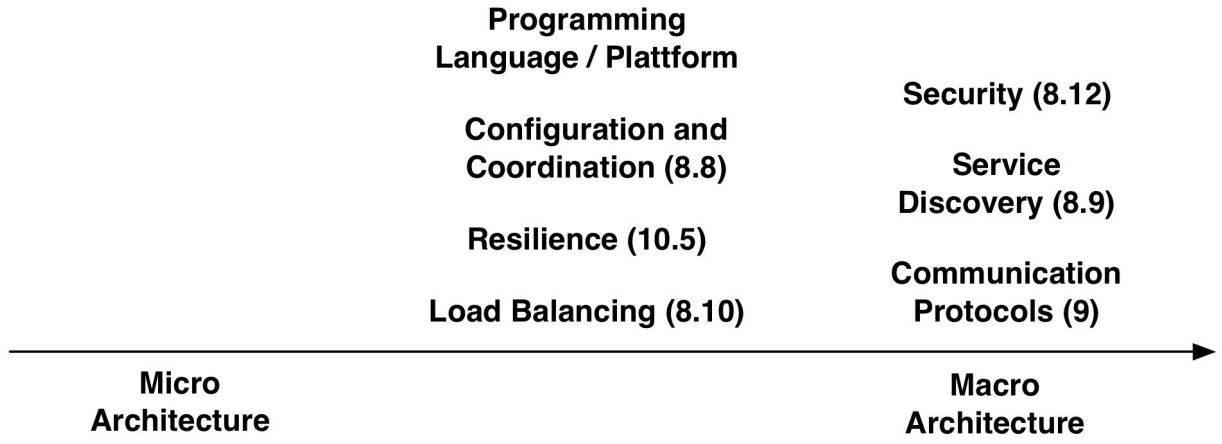


Fig. 64: Technology: macro and micro architecture

Operations

In the area of operations there is control (section 12.5), monitoring (section 12.3), logging (section 12.2) and deployment (section 12.4). To reduce the complexity of the environment and to enable a uniform operations solution these areas have to be defined by macro architecture. The same holds true for platform and programming language. However, standardizing is not obligatory: When the entire operations of

the Microservices rests with the teams, theoretically each team can use a different technology for each of the mentioned areas. But while this scenario does not generate many advantages, it creates a huge technological complexity. However, it is for example possible that the teams use their own special solution for certain tasks. When for instance the revenue is supposed to be transferred in a different way into the monitoring for the business stakeholders, this is certainly doable.

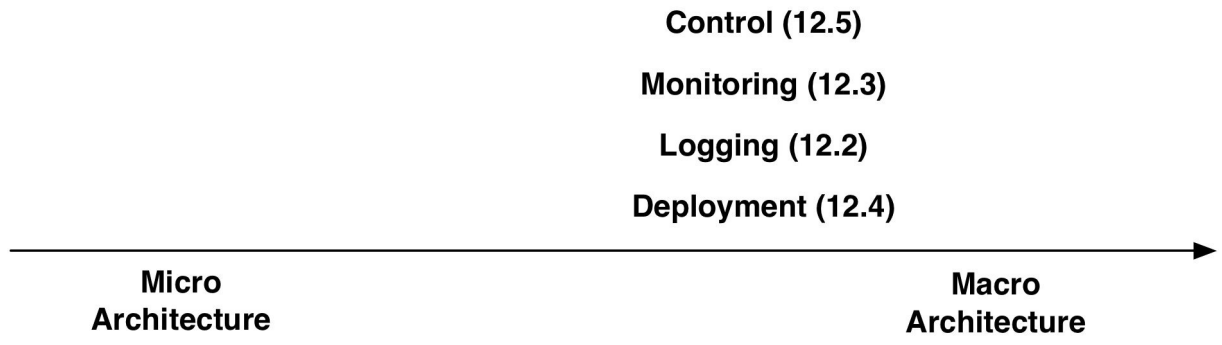


Fig. 65: Operations: macro and micro architecture

Domain Architecture

In the context of domain architecture the distribution of domains to teams is part of the macro architecture (section 8.1). It does not only influence the architecture, but decides also which teams are responsible for which domains. Therefore this task cannot be moved into the micro architecture. However, the domain architecture of the individual Microservices has to be left to the teams (section 10.1, 10.2, 10.3, 10.4). To dictate the domain architecture of the individual Microservices to the teams would be equivalent to treating Microservices at the organizational level like monoliths because the entire architecture is centrally coordinated. In that case one could as well develop a Deployment Monolith which is technically easier. Such a decision would not make sense.

Fig. 66: Architecture: macro and micro architecture

Tests

In the area of testing integration tests (section 11.4) belong to the macro architecture. In practice it has to be decided whether there should be an

integration test for a certain domain and who should implement it. Integration tests only make sense when they concern functionalities across teams. The respective teams can test all other functionalities on their own. Therefore integration tests have to be globally coordinated across teams. Technical tests (section 11.8) can be dictated to the teams by the macro architecture. They are a good option to enforce and control global standards and technical areas of macro architecture. Consumer-driven contract tests (CDC) (section 11.7) and Stubs (section 11.6) can be coordinated between the teams themselves. A shared technological foundation as part of macro architecture can profoundly facilitate development. Uniform technologies are especially sensible in this area since teams have to use the CDCs and Stubs of other teams. When only one technology is used, work is markedly easier. However, it is not obligatory that technologies are rigidly prescribed by the macro architecture.

How to test the respective Microservices should be up to the individual teams as they have the responsibility for the quality of the Microservices.

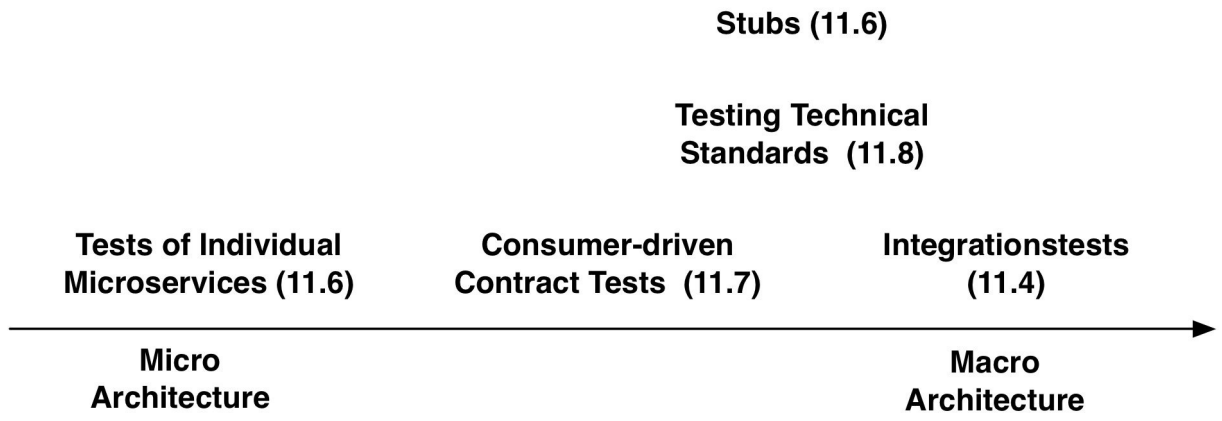


Fig. 67: Test: macro and micro architecture

In many areas decisions can be made either at the level of macro or at the level of micro architecture. It is a central objective of Microservice-based architectures to give the individual teams as much independence as possible. Therefore, as many decisions as possible should be made on the level of micro architecture and therefore by the individual teams. However, in regards to operations the question arises whether the teams really profit from the freedom to use their own distinct tools. It seems more likely that the technology zoo just gets bigger without real advantages. In this area there is a connection to DevOps (section 13.5). Depending on the degree of cooperation between developers and operations there can be different degrees of freedom. In case of a clear division between

development and operations operations will define many standards in macro architecture. In the end operations will have to take care of the Microservices in production. When all Microservices employ a uniform technology, this task is easier.

When defining programming language and platform one should likewise weigh the advantages of specialized technology stacks versus the disadvantages of having heterogeneous technologies in the overall system. Depending on the circumstances the decision to prescribe a technology stack might be as sensible as the decision to leave the technology choice to the individual teams. A uniform technology stack can facilitate operations and make it easier for developers to change between Microservices and teams. Specialized technology stacks make it easier to handle special challenges and motivate employees who thus have the possibility to use cutting edge technologies.

Whether a Microservice really conforms to the macro architecture can be evaluated by a test (compare section 11.8). This test can be an artifact which is likewise part of the macro architecture. The group responsible for the macro architecture can use this artifact to unambiguously define the macro architecture. This allows to check whether all Microservices are in line with macro architecture.

13.4 Technical Leadership

The division in micro and macro architecture completely changes the technical leadership teams and is an essential advantage of Microservices. The macro architecture defines technical duties and freedom. The freedom of choice entails also the responsibility for the respective decisions.

For example a database can be prescribed. In that case the team can delegate the responsibility for the database to the technical leadership team. If the database decision were part of the micro architecture, the database would be run by the team since it made the decision for the technology. No other team would need to deal with potential consequences of this decision (compare section 8.7). Whoever makes the decision, also has the responsibility. The technical leadership team for sure can make such decisions, but by doing so it takes away responsibility from the Microservices teams and thereby independence.

A larger degree of freedom entails more responsibility. The teams have to be able to deal with this and also have to want this freedom. Unfortunately, this is not

always the case. This can either argue for more macro architecture or for organizational improvements which in the end lead to more self organization and thus less macro architecture. It is one of the objectives of the technical leadership team to enable less macro architecture and to lead the way to more self organization.

Developer Anarchy

The approach Developer Anarchy is even more radical in regards to the freedom of the teams. It confers the entire responsibility to the developers. They cannot only freely choose technologies, but even rewrite code if they deem it necessary. Besides, they communicate directly with the stake holders. This approach is employed in very fast growing enterprises and works very well there. Behind this idea is Fred George who has collected more than 40 years of experience while working in many different companies. In a model like this macro architecture and Deployment Monoliths are abolished so that the developers can do what they think is best. This approach is very radical and shows how far the idea can be extended.

Try and Experiment

In Fig. 64, Fig. 65, Fig. 66 and Fig. 67 areas are marked which can belong to either micro or macro architecture. These are the elements which are depicted in the center of the respective figure. Look through these elements and decide whether you would place them in micro or macro architecture. Most important is your reasoning for the one or the other alternative. Take into consideration that making decisions at the level of the micro architecture rather corresponds to the Microservice idea of independent teams.

13.5 DevOps

DevOps denotes the concept that developments (Dev) and operations (Ops) merge into one team (DevOps). This is an organizational change: Each team has developers and operations experts. They work together in order to develop and operate a Microservice. This requires a different mindset since operations-associated topics are often unfamiliar to developers while people working in operations often do not work in projects, but usually run systems independently of projects. Ultimately, the technical skills become very similar: Operations works more on automation and associated suitable tests – and this is in the end software

development. At the same time monitoring, log analysis or deployment turn more and more also into topics for developers.

DevOps and Microservices

DevOps and Microservices ideally complement each other:

- The teams cannot only take care of the development, but also of the operations of the Microservices. This requires that the teams have knowledge in the areas of operations and development.
- Orienting the teams in line with features and Microservices represents a sensible organizational alternative to the division into operations and development.
- Communication between operations and development gets easier when members of both areas work together in one team. Communication within a team is easier than between teams. This is in line with the aim of Microservices to reduce the need for coordination and communication.

DevOps and Microservices fit very well together. In fact, the aim that teams deploy Microservices up to production and keep taking care of them in production can only be achieved with DevOps teams. This is the only way to ensure that teams have the necessary knowledge about both areas.

Do Microservices Necessitate DevOps?

DevOps is such a profound change in organization that many enterprises are still reluctant to take this step. Therefore the question arises whether Microservices can also be implemented without introducing DevOps. In fact, this is possible:

- Via the macro vs. micro architecture division operations can define standards. Then technical elements like logging, monitoring or deployment belong to the macro architecture. When these standards are conformed to, operations can take over the software and make it part of the standard operations processes.
- In addition, platform and programming language can be defined as much as needed for operations. When operations only feels comfortable with running Java applications on a Tomcat, this can be prescribed as platform in the macro architecture. The same holds true for infrastructure elements like databases or messaging systems.
- Moreover, there can be organizational requirements: For example, operations can ask that members of the Microservices teams are available at certain

> times so that problems arising in production can be referred to the teams. To put it concretely: Who wants to deploy on his/her own, has to provide a phone number and will also be called at night in case of problems. If the call is not answered, the management can be called next. This increases the likelihood that developers actually answer such calls.

In such a context the teams cannot be responsible anymore for bringing all Microservices up to production. Access and responsibility rest with operations. There has to be a point in the Continuous Delivery Pipeline where the Microservices are passed on to operations and then are rolled out in production. At this point the Microservice passes into the responsibility of operations which has to coordinate with the respective team about their Microservices. A typical point for the transfer to operations is immediately after the test phases, prior to possible explorative tests. Operations is at least responsible for the last phase, i.e. the rollout in production. Operations can turn into a bottleneck if a high number of modified Microservices have to be brought into production.

Overall DevOps and Microservices have synergies – however, it is not necessarily required to also introduce DevOps when deciding for Microservices.

When Microservices Meet Classical IT Organizations (Alexander Heusingfeld)

by Alexander Heusingfeld, innoQ

The “Microservices” topic has meanwhile reached numerous IT departments and is discussed there. Interestingly, initiatives for introducing Microservices are often started by middle management. However, frequently too little thought is spent on the effect a Microservice architecture has on the (IT) organization of enterprises. Because of this I would like to tell of a number of “surprises” which I experienced during the introduction of such an architecture approach.

Pets vs. Cattle

“Pets vs. Cattle” is a slogan which reached a certain fame at the outset of the DevOps movement. Its basic message is that in times of cloud and virtualization servers should not be treated like pets, but rather like a herd of cattle. If a pet gets sick, the owner will likely nurse it back to health. Sick cattle on the other hand is killed immediately in order not to endanger the health of the entire herd.

Thus the point is to avoid the personification of servers – e.g. by giving them names (like Leviathan, Pollux, Berlin or Lorsch). If you assign such “pet” names to servers, there will be a tendency to care for them like pets and thus provide individual updates, scripts, adjustments or other specific modifications. However, it is well known that this has negative consequences for the reproducibility of installations and server state. Especially considering auto-scaling and failover features as they are required for Microservice-based architectures, this is a K.O. criterion.

One of my projects addressed this problem in a very interesting manner: The server and virtual machines still had names. However, the administration of these systems was completely automated via Puppet. Puppet downloaded the respective scripts from an SVN repository. In this repository individual scripts for each server were stored. This scenario could be called “Puppets for automated pet care”. The advantage is that crashed servers can quickly be replaced by exact copies.

However, requirements for scalability are not taken into consideration at all, since there can always only be one instance of a “pet server” named Leviathan. An alternative is to switch to parameterized scripts and to use templates like “production VM for app XYZ”. At the same time this also allows more flexible deployment scenarios like Blue/Green Deployments. In that case it is not relevant anymore whether the VM app-xyz-prod08.zone1.company.com or app-xyz-prod045.zone1.company.com gets the job done. The only relevant point is that eight instances of this service are constantly available and at times of high load additional instances can be started. How these instances are named does not matter.

Us vs. Them

“Alarming is our concern!”

“You shouldn’t care about that!”

“That is none of your business, it’s our area!”

Unfortunately I frequently hear sentences like these in so-called cross-functional teams. These are teams composed of architects, developers, testers and administrators. Especially if the members previously worked in other, purely functional teams within the same company, old trench wars and prejudices are

carried along into the new team – often subconsciously. Therefore, it is important to be aware of the social aspects right from the start and to counter these proactively. For example, in my experience it has very positive effects to let newly setup teams work in the same office for the first two to four weeks. This allows the new team mates to get to know each other's human side and to directly experience the colleague's body language, character and humor. This will markedly facilitate communication during the later course of the project, misunderstandings are avoided.

In addition, team building measures during the first weeks which require that the team members rely on each other can help to break the ice, to get an idea of the strengths and weaknesses of the individual members and to build up and strengthen trust within the team. If these points are neglected, there will be noticeable adverse consequences throughout the run time of the project. People who do not like each other or do not trust each other, will not rely on each other, even if only subconsciously. And this means that they will not be able to work 100% as a team.

Development vs. Test vs. Operation: Change of Perspective

In many companies there are initiatives for a change of perspective. For example, employees from sales may work in the purchasing department for a day to get to know the people and the processes there. The expectation is that the employees will develop a better understanding for their colleagues and to let that become part of their daily work so that cross-department processes harmonize better. The motto is: "On 'the other side' you get to know a new perspective!"

Such a change of perspective can also be advantageous in IT. A developer could for instance get a new perspective with regards to the use cases or test cases. This might motivate them to enforce a modularization in the development which is easier to test. Or they might consider early in development which criteria will be needed later on to better monitor the software in production or to more easily find errors. A deeper insight into the internal processes of the application can help an administrator to develop a better understanding for implementing a more specific and more efficient monitoring. Each perspective, which deviates from one's own perspective, can raise questions which previously were not considered in this section of the application life cycle. And these questions will help the team to evolve as a whole and deliver better software.

For Ops there is Never an "Entirely Green Field"

For sure Microservices are a topical subject and bring along new technologies, concepts and organizational changes. However, one should always consider that enterprises introducing Microservices hardly ever start from scratch! There are always some kind of legacy systems or entire IT environments which already exist and might better not be replaced in a Big Bang approach. Usually these legacy systems have to be integrated into the brave new world of Microservices, at least they will have to coexist.

For this reason, it is important to take these systems into consideration when planning a Microservices-based architecture, especially in regards to IT costs. Can the existing hardware infrastructure really be restructured for the Microservices or is there a legacy system which relies exactly on this infrastructure? These are often questions which get caught on the infrastructure or operations team – if there is such an organizational unit in the company. Otherwise it might happen that these questions first arise when a deployment to the system test or production environment is supposed to be done. Especially for being able to recognize these questions early on, I recommend to deal with the deployment pipeline as early as possible in the reorganization project. The deployment pipeline should already be in place before the first business functionality is implemented by the teams. A simple “Hello World” will often be sufficient which then is brought towards production by the combined forces of the entire team. While doing so, the team will almost always encounter open questions which in the worst case will have effects on the design of the systems. However, as not much is implemented at this stage, early on during the project such changes are still comparably cost-efficient to implement.

Conclusion

The organizational changes (resp. “Conway’s Law”), which accompany the introduction of Microservices, are up to now often underestimated. Old habits, prejudices and maybe even trench wars are often deep-rooted. Especially if the new team mates were previously assigned to different departments. However, “one team” has to be more than just a buzzword. If the team manages to bury their prejudices and to put their different experiences to good use, it can advance together. Everyone has to understand that all of them now share the task and responsibility to bring a stable software into production for the customer. Everybody can profit from the experiences of the others when everybody acts on the premise: “Everybody voices their concerns, and we will solve it jointly”.

13.6 Interface to the Customer

To ensure that the development can really be scaled to multiple teams and Microservices, each team needs to have its own Product Owner. In line with Scrum approaches he/she is responsible for the further development of the Microservice. For this purpose he/she defines stories which are implemented in the Microservice. The Product Owner is the source of all requirements and prioritizes them. This is especially easy when a Microservice only comprises features which are within the responsibility of a single department at the business level (Fig. 68). Usually this objective is achieved by adjusting Microservices and teams to the organization of departments. Each department gets “its” Product Owner and therefore “its” team and “its” Microservices.

When the Microservices have a good domain architecture, they can be independently developed. Ultimately, each domain should be implemented in one or many Microservices, and the domain should only be of interest to one department. The architecture has to take the organization of the departments into consideration when distributing the domains into Microservices. This ensures that each department has its own Microservices that are not shared with other domains or departments.

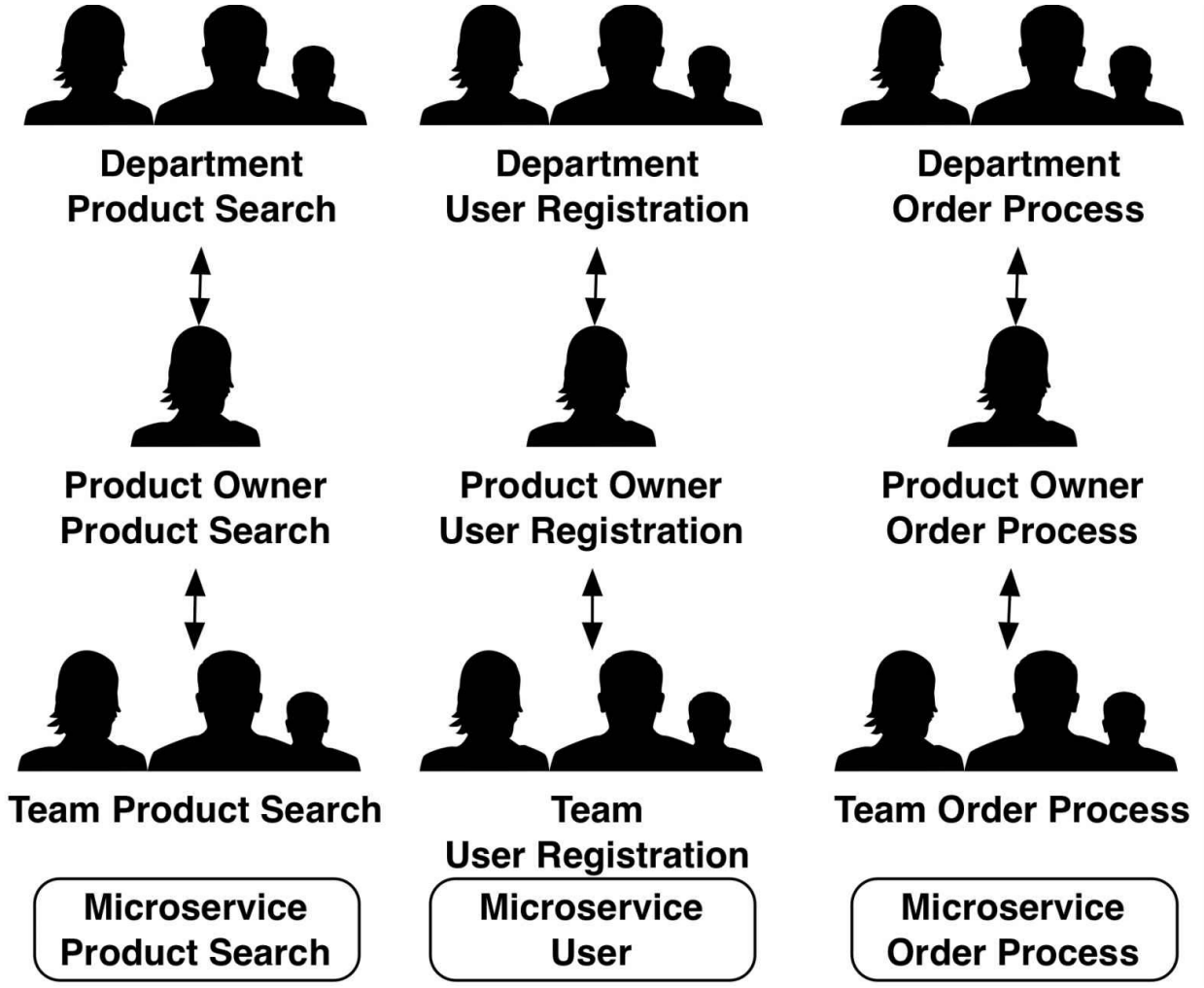


Fig. 68: Department, product owner and Microservices

Unfortunately, the architecture often is not perfect. Besides, Microservices have interfaces – an indication that functionalities might span multiple Microservices. When multiple functionalities concern one Microservice and therefore multiple departments want to influence the development of a Microservice, the Product Owner has to ensure a prioritization which is coordinated with the different departments. This can be a challenge because departments can have different priorities. In that case the Product Owner has to coordinate between the concerned departments.

Let us assume that there is a department which takes care of sales campaigns in an E-commerce shop. It starts a campaign where orders containing a certain item get a rebate on the delivery cost. The required modification concerns the order team: It has to find out whether an order contains such an item. This information has to be transmitted to the delivery Microservice which has to calculate the costs for

the delivery. Accordingly, the Product Owners of these two teams have to prioritize these changes in regards to the changes desired by the departments in charge of delivery and orders. Unfortunately, many of these sales campaigns combine different functionalities so that such a prioritization is often required. The departments for orders and deliveries have their own Microservices, while the department in charge of sales campaigns does not have its own Microservices. Instead it has to introduce its features into the other Microservices.

Architecture Leads to Departments

The Microservice architecture can thus be a direct result of the departmental organization of the company. However, there are also cases where a new department is created around an IT system, which then takes care of this system from the business side. In such a case one can argue that the Microservices architecture directly influences the organization. For instance there might be a new Internet market place which is implemented by an IT system. If it is successful, a department can be created which takes over the further development of this marketplace. This department will continue to develop the IT system from a domain and from a business perspective. In this case the marketplace was developed first and subsequently the department has been created. Therefore the system architecture has defined the departmental structure of the organization.

13.7 Reusable Code

At first sight the reuse of code is a technical problem. [Section 8.3]{#section8-3} already described the challenges which arise when two Microservices use the same library: When the Microservices use the library in such a way that a new release of the library necessitates a new deployment of the Microservices, the result is a deployment dependency. This has to be avoided to allow for an independent deployment of the Microservices. There is additional expenditure because the teams responsible for the Microservices have to coordinate their changes to the library. New features for the different Microservices have to be prioritized and developed. These represent also dependencies between the teams which should rather be avoided.

Client Libraries

Client libraries which encapsulate calls from a Microservice can be acceptable. When the interfaces of the Microservices are downwards compatible, the client library does not have to be replaced in case of a new version of the Microservice. In such a scenario client libraries do not cause problems because a new

deployment of the called Microservices does not lead to an update of the client library or a new deployment of the calling Microservice.

However, when the client library also contains domain objects, problems can occur. When a Microservice wants to change the domain model, the team has to coordinate this change with the other users of the client library and therefore cannot develop independently anymore. The boundaries between a simplified use of the interface which can be sensible and a shared implementation of logic or other deployment dependencies which can be problematic is not clear cut. One option is to entirely forbid shared code.

Reuse Anyhow?

However, obviously, projects can reuse code. Hardly any project nowadays manages without some open source library. Using this code is obviously easy and thus facilitates work. Problems like the ones arising upon reusing code between Microservices are unlikely for a number of reasons:

- Open source projects in general are of high quality. Developers working in different companies use the code and thereby spot errors. Often they even remove the errors so that the quality permanently increases. To publish source code and thereby provide insight into internals is often already motivation enough to increase the quality.
- The documentation allows to immediately start to use the code without a need to directly communicate with the developers. Without a good documentation open source projects hardly find enough users or additional developers since getting started would be too hard.
- There is a coordinated development with a bug tracker and a process for accepting code changes introduced by external developers. Therefore errors and their fixes can be tracked. In addition, it is clear how changes from the outside can be incorporated into the code basis.
- Moreover, in case of a new version of the open source library it is not necessary for all users to use the new version. The dependencies in regards to the library are not so pronounced that a deployment dependency ensues.
- Finally, there are clear rules how one’s own supplements can be incorporated into the open source library.

In the end the difference between a shared library and an open source project is mainly a higher quality in regards to different aspects. Besides there is also an organizational aspect: There is a team which takes care of the open source

project. It directs the project and keeps developing it. This team does not necessarily make all changes, but it coordinates them. Ideally, the team has members from different organizations and projects so that the open source project is developed under different view points and in the context of different use cases.

Reuse as Open Source

With open source projects as role models in mind there are different options for reusable code in a Microservices project:

- The organization around reusable libraries is structured like in an open source project. There are employees responsible for the continued code development, the consolidation of requirements and for incorporating the changes of other employees. The team members ideally come from different Microservice teams.
- The reusable code turns into a real open source project. Developers outside of the organization can use and extend the project.

Both decisions can result into a significant investment since markedly more effort has to go into quality and documentation etc. Besides, the employees working on the project have to get enough freedom to do so in their teams. The teams can control the prioritization in the open source project by only making their members available for certain tasks. Due to the large investment and potential problems with prioritization the decision to establish an open source project should be well considered. The idea itself is not new – experiences in this area have already been collected for quite some time.

If the investment is very high, it means that the code is hardly reusable for the moment and using the code in its current state causes quite some effort. Probably the code is not only hard to reuse, but hard to use at all. The question is why team members would accept such a bad code quality. Investing into code quality in order to make the code reusable can pay off already by reusing it just once.

At first glance it does not appear very sensible to make code available to external developers. This requires that code quality and documentation are of high enough quality for external developers to be able to use the code without direct contact to the developers of the open source project. Only the external developers seem to profit from this approach as they get good code for free.

However, a real open source project has a number of advantages:

- External developers find weak spots by using the code. Besides, they will use the code in different projects so that it gets more generalized. This will improve quality as well as documentation.
- Maybe external developers contribute to the further development of the code. However, this is rather the exception than the norm. But having external feedback via bug reports and requests for new features can already represent a significant advantage.
- Running open source projects is great marketing for technical competence. This can be useful for attracting employees as well as customers. Important is the extent of the project. If it is only a simple supplement of an existing open source project, the investment can be manageable. An entirely new open source framework is a very different topic.

Blueprints, i.e. documentations for certain approaches, represent elements which are fairly easy to reuse. This can be elements of macro architecture like for instance a document detailing the correct approach for logging. Likewise there can be templates which contain all necessary components of a Microservice including a code skeleton, a build script and a Continuous Delivery Pipeline. Such artifacts can rapidly be written and are immediately useful.

Try and Experiment

Maybe you have already previously used your own technical libraries in projects or even developed some yourself. Try to estimate how large the expenditure would be to turn these libraries into real open source libraries. Apart from a good code quality this necessitates also documentation about the use and the extension of the code. Besides, there have to be a bug tracker and forums. How easy would it be to reuse it in the project itself? How high would be the quality of the library?

13.8 Microservices Without Changing the Organization?

Microservices are more than just an approach for software architecture. They have pronounced effects on organization. Changes to the organization are often very difficult. Therefore the question arises whether Microservices can be implemented without changing the organization.

Microservices Without Changing the Organization?

Microservices allow for independent teams. The domain-focused teams are responsible for one or multiple Microservices – this includes ideally their

development as well as operations. Theoretically it is possible to implement Microservices without dividing developers into domain-focused teams. In that case the developers could modify each Microservice – an extension of the ideas presented in section 13.2. It would even be possible that technically focused teams work on Microservices which are split according to domain-based criteria. In this scenario there would be a UI, a middle tier and a database team which work on domain Microservices such as order process or registration. However, a number of advantages usually associated with Microservices cannot be exploited anymore in that case. Firstly, it is not possible anymore to scale the agile processes via Microservices. Secondly, it will be necessary to restrict the technology freedom since the teams will not be able to handle the different Microservices if they all employ different technologies. Besides, each team can modify each Microservice. This entails the danger that though a distributed system is created there are dependencies which prevent the independent development of individual Microservices. The necessity for independent Microservices is obliterated because a team can change multiple Microservices together and therefore can handle also Microservices having numerous dependencies. However, even under these conditions sustainable development, an easier start with Continuous Delivery, independent scaling of individual Microservices or a simple handling of legacy systems can still be implemented because the deployment units are smaller.

Evaluation

To put it clearly: Introducing Microservices without creating domain-focused teams does not lead to the main benefits meant to be derived from Microservices. It is always problematic to implement only some parts of a certain approach as only the synergies between the different parts will generate the overall value. Although implementing Microservices without domain-focused teams is a possible option – it is for sure not recommended.

Departments

As already discussed in section 13.6, the Microservice structure should ideally extend to the departments. However, in reality this is sometimes hard to achieve since the Microservice architecture often deviates too much from the organizational structure of the departments. It is unlikely that the organization of the departments will adapt to the distribution into Microservices. When the distribution of the Microservice cannot be adjusted, the respective Product Owners have to take care of prioritization and coordinate the wishes of the departments, which concern multiple Microservices, in such a way that all

requirements are unambiguously prioritized for the teams. If this is not possible, a Collective Code Ownership approach (section 13.2) can limit the problem. In this case the Product Owner and his/her team can also modify Microservices which do not really belong to their sphere of influence. This can be the better alternative in contrast to a coordination across teams – however both solutions are not optimal.

Operations

In many organizations there is a separate team for operations. The teams responsible for the Microservices should also take care of the operations of their Microservices following the principle of DevOps. However, as discussed in section 13.5, it is not a strict requirement for Microservices to introduce DevOps. If the separation between operations and development is supposed to be maintained, operations has to define the necessary standards for the Microservices in the macro architecture to ensure a smooth operations of the system.

Architecture

Often architecture and development are likewise kept separated. In a Microservices environment there is the area of macro architecture where architects make global decisions for all teams. Alternatively, the architects can be distributed to the different teams and work together with the teams. In addition, they can found an overarching committee which defines topics for macro architecture. In that case it has to be ensured that the architects really have time for this task and are not completely busy with work in their team.

Try and Experiment

What does the organization of a project you know look like?

- Is there a special organizational unit which takes care of architecture? How would they fit into a Microservices-based architecture?
- How is operations organized? How can the organization of operations best support Microservices?
- How well does the domain-based division fit to the departments? How could it be optimized?
- Can a Product Owner with fitting task area be assigned to each team?

13.9 Conclusion

Microservices enable the independence of teams in regards to technical decisions and deployments (section 13.1). This allows the teams to independently

implement requirements. In the end this makes it possible for numerous small teams to work together on a large project. This reduces the communication overhead between the teams. Since the teams can deploy independently, the overall risk of the project is reduced.

Ideally the teams should be put together in a way that allows them to work separately on different domain aspects. If this is not possible or requires too much coordination between the teams, Collective Code Ownership can be an alternative (section 13.2). In that case each developer can change all of the code. Still one team has the responsibility for each Microservice. Changes to this Microservice have to be coordinated with the responsible team.

Section 13.3 described that Microservices have a macro architecture which comprises decisions which concern all Microservices. In addition, there is the micro architecture which can be different for each Microservice. In the areas of technology, operations, domain architecture and testing there are decisions which can either be attributed to micro or macro architecture. Each project has the choice to delegate them to teams (micro architecture) or to centrally define them (macro architecture). Delegating into teams is in line with the objective to achieve a large degree of independence – and is therefore often the better option. A separate architecture team can define the macro architecture – alternatively, the responsible team is assembled from members of the different Microservice teams.

Responsibility for the macro architecture is closely linked to a concept for technical leadership (section 13.4): Less macro architecture means more responsibility for the Microservice teams and less responsibility for the central architecture team.

Though Microservices profit from merging operations and development to DevOps (section 13.5), it is not strictly required to introduce DevOps to do Microservices. If DevOps is not possible or desired, operations can define guidelines in the context of macro architecture to unify certain aspects in order to ensure a smooth operations of the Microservice-based system.

Microservices should always implement their own separate requirements. Therefore it is best when each Microservice can be assigned to a certain department on the business side (section 13.6). If this is not possible, the Product Owners have to coordinate the requirements coming from different departments in such a way that each Microservice has clearly prioritized requirements. When

Collective Code Ownership is used, a Product Owner and his/her team can also change Microservices of other teams – which can limit the communication overhead. Instead of coordinating priorities, a team will introduce the changes which are necessary for a new feature by itself – even if they concern different Microservices. The team responsible for the modified Microservice can review the introduced changes and adjust them if necessary.

Code can be reused in a Microservices project if the code is treated like an open source project (section 13.7). An internal project can be handled like an internal open source project – or can in fact be turned into a public open source project. It has to be considered that the effort for a real open source project is high. Therefore, it can be more efficient not to reuse code. Besides, the developers of the open source project have to prioritize domain requirements versus changes to the open source project which can be a difficult decision at times.

Section 13.8 discussed that an introduction of Microservices without changes to the organizational structure at the development level does not work in real life. When there are no domain-focused teams which can develop certain domain aspects independently of other teams, it is practically impossible to develop multiple features in parallel and thus to bring more features to the market within the same time. However, this is just what Microservices were meant to achieve. Sustainable development, an easy introduction of Continuous Delivery, independent scaling of individual Microservices or a simple handling of legacy systems are still possible. Operations and an architecture team can define the macro architecture so that in this area changes to the organizational structure are not strictly required. Ideally, the requirements of the departments are always reflected by one Microservice. If that is not possible, the Product Owners have to coordinate and prioritize the required changes.

Essential Points

- Microservices have significant effects on the organization. Independent small teams which together work on a large project are an important advantage of Microservices.
- Viewing the organization as part of the architecture is an essential innovation of Microservices.
- A combination of DevOps and Microservices is advantageous, but not obligatory.

Part IV: Technologies

This part of the book shows how Microservices can be implemented with concrete technologies. Chapter 14 contains a complete example for a Microservices-architecture based on Java, Spring, Spring Boot, Spring Cloud, the Netflix stack and Docker. The example is a good starting point for your own implementation or experiments. Many of the technological challenges discussed in Part 3 are solved in this part with the aid of concrete technologies â€“ for instance build, deployment, services discovery, communication, load balancing and tests.

Even smaller than Microservices are the Nanoservices from chapter 15. They require special technologies and a number of compromises. Therefore the chapter introduces technologies which can implement very small services i.e. Amazon Lambda for JavaScript and Java, OSGi for Java, Java EE, Vert.x on the JVM (Java Virtual Machine) with support for languages like Java, Scala, Clojure, Groovy, Ceylon, JavaScript, Ruby or Python. The programming language Erlang likewise enables very small services, but is also able to integrate other systems. Seneca is a JavaScript framework specialized in the implementation of Nanoservices.

At the close of the book chapter 16 shows what can be achieved with Microservices.

14 Example for a Microservices-based Architecture

This chapter provides an example for an implementation of a Microservices-based architecture. It aims at demonstrating concrete technologies in order to lay the foundation for experiments. The example application has a very simple domain architecture containing a few compromises. Section 14.1 deals with this topic in detail.

For a real system with a comparable low complexity as the presented example application an approach without Microservices would be better suited. However, the low complexity makes the example application easy to understand and simple to extend. Some aspects of a Microservice environment, such as security, documentation, monitoring or logging, are not illustrated in the example application – but these aspects can be relatively easily addressed with some experiments.

Section 14.2 explains the technology stack of the example application. The build tools are described in section 14.3. Section 14.4 deals with Docker as technology for the deployment. Docker needs to run in a Linux environment. Section 14.5 describes Vagrant as a tool for generating such environments. Section 14.6 introduces Docker Machine as alternative tool for the generation of a Docker environment, which can be combined with Docker Compose for the coordination of several Docker Containers (section 14.7). The implementation of Service Discovery is discussed in section 14.8. The communication between the Microservices and the user interface is the main topic of section 14.9. Thanks to Resilience other Microservices are not affected if a single Microservice fails. In the example application resilience is implemented with Hystrix (section 14.10). Load Balancing (section 14.11), which can distribute the load onto several instances of a Microservice, is closely related to that. Possibilities for the integration of Non-Java-technologies are detailed in section 14.12, and testing is discussed in section 14.13.

The code of the example application can be found at https://github.com/ewolff/microservice. It is Apache-licensed, and can,

accordingly, be used and extended freely for any purpose.

14.1 Domain Architecture

The example application has a simple web interface, with which users can submit orders. There are three Microservices (Fig. 69):

- **Catalog** keeps track of products. Items can be added or deleted.
- **Customer** performs the same task in regards to customers: It can register new customers or delete existing ones.
- **Order** cannot only show orders, but also create new orders.

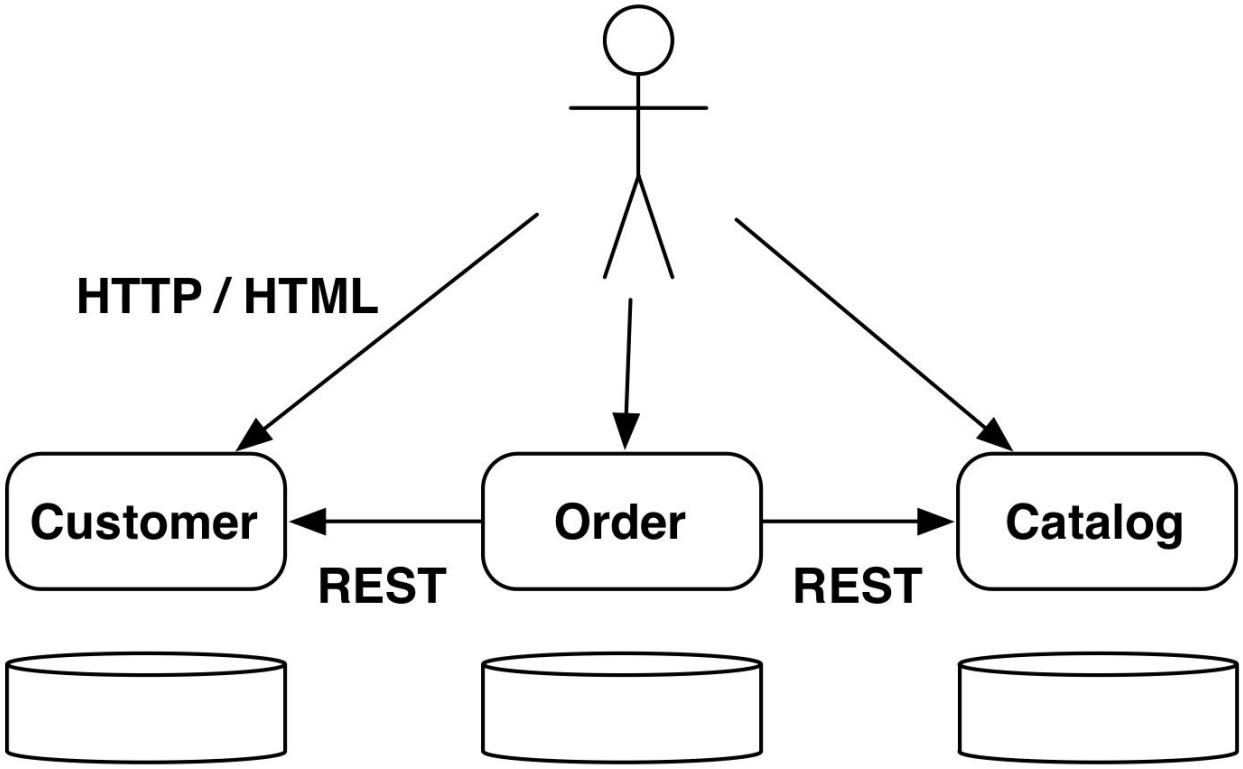


Fig. 69: Architecture of the example application

For the orders the Microservice “Order” needs access to the two other Microservices, “Customer” and “Catalog”. The communication is achieved via REST. However, this interface is only meant for the internal communication between the Microservices. The user can interact with all three Microservices via the HTML-/HTTP-interface.

Separate Data Storages

The data storages of the three Microservices are completely separate. Only the respective Microservice knows the information about the business objects. The Microservice “Order” saves only the primary keys of the items and customers, which are necessary for the access via the REST interface. A real system should rather use artificial keys as the internal primary keys otherwise get visible to the outside. These are internal details of the data storage that should be hidden. To expose the primary keys, the class **SpringRestDataConfig** within the Microservices configures Spring Data Rest accordingly.

Lots of Communication

Whenever an order needs to be shown, the Microservice “Customer” is called for the customer data and the Microservice “Catalog” for each line of the order in order to determine the price of the item. This can have a negative influence on the response times of the application as the display of the order cannot take place before all requests have been answered by the other Microservices. As the requests to the other services take place synchronously and sequentially, latencies will add up. This problem can be solved by using asynchronous parallel requests.

In addition a lot of computing power is needed to marshal the data for sending and receiving. This is acceptable in case of such a small example application. When such an application is supposed to run in production, alternatives have to be considered.

This problem can for instance be solved by caching. This is relatively easy as customer data will not change frequently. Items can change more often – still, by far not so fast that caching would pose a problem. Only the amount of data can interfere with this approach. The use of Microservices has the advantage that such a cache can be implemented relatively simply at the interface of the Microservices, or even at the level of HTTP, if this protocol is used. An HTTP Cache, like the one used for websites, can be added to REST Services in a transparent manner and without much programming effort.

Bounded Context

Caching will solve the problem of too long response times technically. However, very long response times can also be a sign for a fundamental problem. Section 4.3 argued that a Microservice should contain a *Bounded Context*. A specific domain model is only valid in a *Bounded Context*. The modularization into Microservices in this example contradicts this idea: The domain model is used to modularize the system into the Microservices “Order” for orders, “Catalog” for

items and “Customer” for customers. In principle the data of these entities should be modularized in different *Bounded Context*s.

The described modularization implements in spite of low domain complexity a system consisting of three Microservices. In this manner the example application is easy to understand while still having several Microservices and demonstrating the communication between Microservices. In a real system the Microservice “Order” can also handle information about the items that is relevant for the order process such as the price. If necessary, the service can replicate the data from another Microservice into its own database in order to access it efficiently. This is an alternative to the aforementioned caching. There are different possibilities how the domain models can be modularized into the different *Bounded Context*s “Order” and “Customer” resp. “Catalog”.

This design can cause errors: When an order has been put into the system and afterwards the price of the item is changed, the price of the order changes as well – that should not happen. In case the item is deleted, there is even an error when displaying the order. In principle the information concerning the item and the customer should become part of the order. In that case the historical data of the orders including customer and item data would be transferred into the service “Order”.

Don’t Modularize Microservices by Data!

It is important to understand the problem inherent in architecting a Microservices system by domain model. Often the task of a global architecture is misunderstood: The team designs a domain model, which comprises for instance objects such as customers, order and items. Based on this model Microservices are defined. That is how the modularization into Microservices could have come about in the example application, resulting in a huge amount of communication. A modularization based on processes such as ordering, customer registration and product search might be more advantageous. Each process could be a *Bounded Context* that has its own domain model for the most important domain objects. For product search the categories of items might be the most relevant while for the ordering process data like weight and size might matter more.

The modularization by data can also be advantageous in a real system. When the Microservice “Order” gets too big in combination with the handling of customer and product data, it is sensible to modularize data handling. In addition the data can be used by other Microservices. When devising the architecture for a system,

there is rarely a single right way of doing things. The best approach depends on the system and the properties the system should have.

14.2 Basic Technologies

Microservices in the example application are implemented with Java. Basic functionalities for the example application are provided by the Spring Framework. This framework offers not only Dependency Injection, but also a Web-Framework, which allows for the implementation of REST-based services.

HSQL Database

The database HSQLDB handles and stores data. It is an In-Memory database, which is written in Java. The database stores the data only in RAM so that all data are lost upon restarting the application. In line with this, this database is not really suited for production use, even if it can write data to a hard disk. On the other hand it is not necessary to install an additional database server, which keeps the example application easy. The database runs in the respective Java application.

Spring Data REST

The Microservices use Spring Data REST in order to provide the domain objects with little effort via REST and to write them into the database. Handing objects out directly means that the internal data representation leaks into the interface between the services. Changing the data structures is very difficult as the clients need to be adjusted as well. However, Spring Data REST can hide certain data elements and can be configured flexibly so that the tight coupling between the internal model and the interface can be decoupled if necessary.

Spring Boot

Spring Boot facilitates Spring further. Spring Boot renders the generation of a Spring system very easy: With Spring Boot Starters predefined packages are available, which contain everything that is necessary for a certain type of application. Spring Boot can generate WAR files, which can be installed on a Java application or web server. In addition it is possible to run the application without an application or web server. The result of the build is a JAR file in that case, which can be run with a Java Runtime Environment (JRE). The JAR file contains everything for running the application and also the necessary code to deal with HTTP requests. This approach is by far less demanding and simpler than the use of an application server https://jaxenter.com/java-application-servers-dead-112186.html.

A simple example for a Spring Boot application is shown in Listing 1. The main program **main** hands the control over to Spring Boot. The class is passed in as a parameter so that the application can be called. The annotation **@SpringBootApplication** makes sure that Spring Boot generates a suitable environment. For example a web server is started, and an environment for a Spring web application is generated as the application is a web application. Because of **@RestController** the Spring Framework instantiates the class and calls methods for the processing of REST requests. **@RequestMapping** shows which method is supposed to handle which request. Upon requests of the URL "/" the method **hello()** is called, which returns as result the sign chain "hello" in the HTTP body. In a **@RequestMapping** annotation URL templates such as "/customer/{id}" can be used. Then a URL like "/customer/42" can be cut into separate parts and the 42 bound to a parameter annotated with **@PathVariable**. As dependency the application uses only spring-boot-starter-web pulling all necessary libraries for the application along, for instance the web server, the Spring Framework and additional dependent classes. Section 14.3 will discuss this topic in more detail.

Listing 1: A simple Spring Boot REST service

```
1 @RestController
2 @SpringBootApplication
3 public class ControllerAndMain {
4
5  @RequestMapping("/")
6  public String hello() {
7   return "hello";
8  }
9
10  public static void main(String[] args) {
11   SpringApplication.run(ControllerAndMain.class,
12   args);
13  }
14
15 }
```

Spring Cloud

Finally the example application uses Spring Cloud to gain easy access to the Netflix Stack. Fig70 shows an overview.

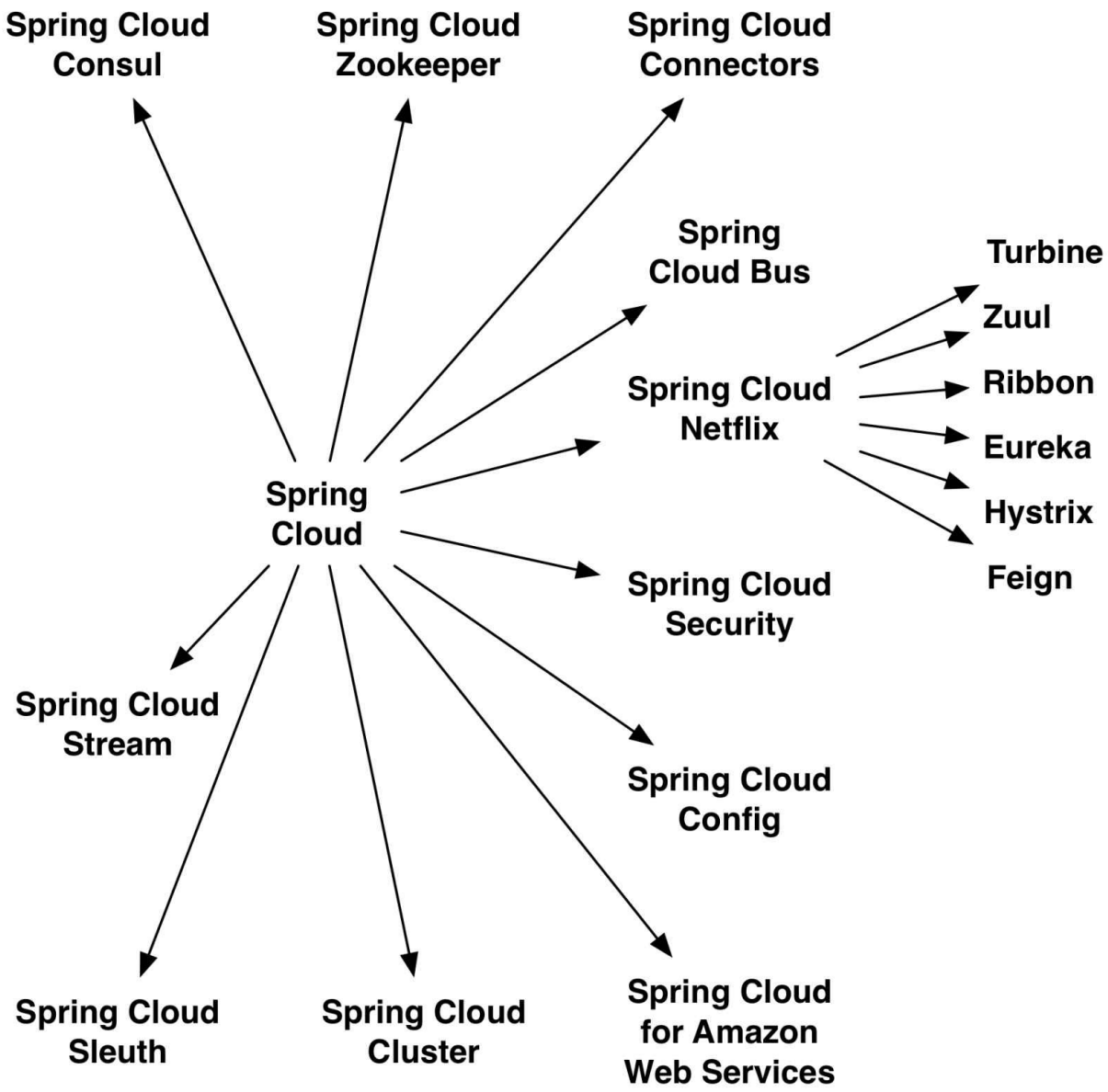


Fig. 70: Overview of Spring Cloud

Spring Cloud offers via the Spring Cloud Connectors access to the PaaS (Platform as a Service) Heroku and Cloud Foundry. Spring Cloud for Amazon Web Services offers an interface for services from the Amazon Cloud. This part of Spring Cloud is responsible for the name of the project, but not helpful for the implementation of Microservices.

However, the other sub projects of Spring Cloud provide a very good basis for the implementation of Microservices:

- **Spring Cloud Security** supports the implementation of security mechanisms as typically required for Microservices, among those Single Sign On into a Microservices environment. That way a user can use each of the Microservices without having to log in anew every time. In addition the user token is transferred automatically for all calls to other REST services to ensure that those calls can also work with the correct user rights.
- **Spring Cloud Config** can be used to centralize and dynamically adjust the configuration of Microservices. Section 12.4 already presented technologies, which configure Microservices during deployment. To be able to reproduce the state of a server at any time, a new server should be started with a new Microservice instance in case of a configuration change instead of dynamically adjusting an existing server. If a server is dynamically adjusted, there is no guarantee that new servers are generated with the right configuration as they are configured via a different way. Because of these disadvantages the example application refrains from using this technology.
- **Spring Cloud Bus** can send dynamic configuration changes for Spring Cloud Config. Moreover, the Microservices can communicate via Spring Cloud Bus. However, the example application does not use this technology because Spring Cloud Config is not used and the Microservices communicate via REST.
- **Spring Cloud Sleuth** enables distributed tracing with tools like Zipkin or Htrace. It can also use a central log storage with ELK (see section 12.2).
- **Spring Cloud Zookeeper** support Apache Zookeeper (see section 8.8). This technology can be used to coordinate and configure distributed services.
- **Spring Cloud Consult** facilitates Services Discovery using Consul (see section 8.9).
- **Spring Cloud Cluster** implements leader election and stateful patterns using technologies like Zookeeper or Consul. It can also used the NoSQL datastore Redis or the Hazelcast cache.
- Finally **Spring Cloud Stream** supports messaging using Redis, Rabbit or Kafka.

Spring Cloud Netflix

Spring Cloud Netflix offers simple access to Netflix Stack, which has been especially designed for the implementation of Microservices. The following technologies are part of this stack:

- **Zuul** can implement routing of requests to different services.
- **Ribbon** serves as Load Balancer.

- **Hystrix** assists with implementing resilience in Microservices.
- **Turbine** can consolidate monitoring data from different Hystrix servers.
- **Feign** is an option for an easier implementation of REST clients. It is not limited to Microservices. It is not used in the example application.
- **Eureka** can be used for Service Discovery.

These technologies are the ones that influence the implementation of the example application most.

Try and Experiment

For an introduction into Spring it is worthwhile to check out the Spring Guides at https://spring.io/guides/. They show in detail how Spring can be used to implement REST services or to realize messaging solutions via JMS. An introduction into Spring Boot can be found at https://spring.io/guides/gs/spring-boot/. Working your way through these guides provides you with the necessary know-how for understanding the additional examples in this chapter.

14.3 Build

The example project is built with the tool Maven. The installation of the tool is described at https://maven.apache.org/download.cgi. The command **mvn package** in the directory **microservice/microservice-demo** can be used to download all dependent libraries from the Internet and to compile the application.

The configuration of the projects for Maven is saved in files named **pom.xml**. The example project has a Parent-POM in the directory **microservice-demo**. It contains the universal settings for all modules and in addition a list of the example project modules. Each Microservice is such a module, and some infrastructure servers are modules as well. The individual modules have their own **pom.xml**, which contains the module name among other information. In addition they contain the dependencies, i.e. the Java libraries they use.

Listing 2: Part of pom.xml including dependencies

```
...
<dependencies>

 <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
  </dependency>
```

```
 8
 9    <dependency>
10      <groupId>org.springframework.boot</groupId>
11      <artifactId>
12          spring-boot-starter-data-jpa
13        </artifactId>
14    </dependency>
15 ...
16 </dependencies>
17 ...
```

Listing 2 shows a part of a **pom.xml**, which lists the dependencies of the module. Depending on the nature of the Spring Cloud features the project is using, additional entries have to be added in this part of the **pom.xml** usually with the **groupId org.springframework.cloud**.

The build process results in one JAR file per Microservice, which contains the compiled code, the configuration and all necessary libraries. Java can directly start such JAR files. Although the Microservices can be accessed via HTTP, they do not have to be deployed on an application or web server. This part of the infrastructure is also contained in the JAR file.

As the projects are built with Maven, they can be imported into all usual Java IDEs (Integrated Development Environment) for further development. IDEs simplify code changes tremendously.

Try and Experiment

Download and Compile the Example

Download the example provided at https://github.com/ewolff/microservice. Install Maven, see https://maven.apache.org/download.cgi. In the sub directory **microservices-demo** execute the command **mvn package**. This will build the complete project.

Create a Continuous Integration Server for the Project

https://github.com/ewolff/user-registration is an example project for a Continuous Delivery project. This contains in sub directory **ci-setup** a setup for a Continuous Integration Server (Jenkins) with static code analysis (Sonarqube) and Artifactory for the handling of binary artefacts. Integrate the Microservices project into this infrastructure so that a new built is triggered upon each change.

The next section (14.4) will discuss Vagrant in more detail. This tool is used for the Continuous Integration Servers. It simplifies the generation of test environments greatly.

14.4 Deployment Using Docker

Deploying Microservices is very easy:

- Java has to be installed on the server.
- The JAR file, which resulted from the build, has to be copied to the server.
- A separate configuration file **application.properties** can be created for further configurations. It is automatically read out by Spring Boot and can be used for additional configurations. An **application.properties** containing default values is comprised in the JAR file.
- Finally a Java process has to start the application out of the JAR file.

Each Microservice starts within its own Docker Container. As discussed in section 12.6, Docker uses Linux Containers. In this manner the Microservice cannot interfere with processes in other Docker Containers and has a completely independent file system. The Docker Image is the basis for this file system. However, all Docker Containers share the Linux Kernel. This saves resources. In comparison to an operating system process a Docker Container has virtually no additional overhead.

Listing 3: Dockerfile for a Microservice used in the example application

```
FROM java
CMD /usr/bin/java -Xmx400m -Xms400m \
  -jar /microservice-demo/microservice-demo-catalog\
  /target/microservice-demo-catalog-0.0.1-SNAPSHOT.jar
EXPOSE 8080
```

A file called **Dockerfile** defines the composition of a Docker Container. Listing 3 shows a Dockerfile for a Microservice used in the example application:

- **FROM** determines the base image used by the Docker Container. A Dockerfile for the image java is contained in the example project. It generates a minimal Docker image with only a JVM installed.
- **CMD** defines the command executed at the start of the Docker Container. In the case of this example it is a simple command line. This line starts a Java application out of the JAR file generated by the build.
- Docker Containers are able to communicate with the outside via network ports. **EXPOSE** determines which ports are accessible from outside. In the example the application receives HTTP requests via port 8080.

14.5 Vagrant

Docker runs exclusively under Linux as it uses Linux Containers. However, there are solutions for other operating systems, which start a virtual Linux machine and thus allow the use of Docker. This is largely transparent so that the use is practically identical to the use under Linux. But in addition all Docker Containers need to be built and started.

To make installing and handling Docker as easy as possible, the example application uses Vagrant. Fig. 71 shows how Vagrant works:

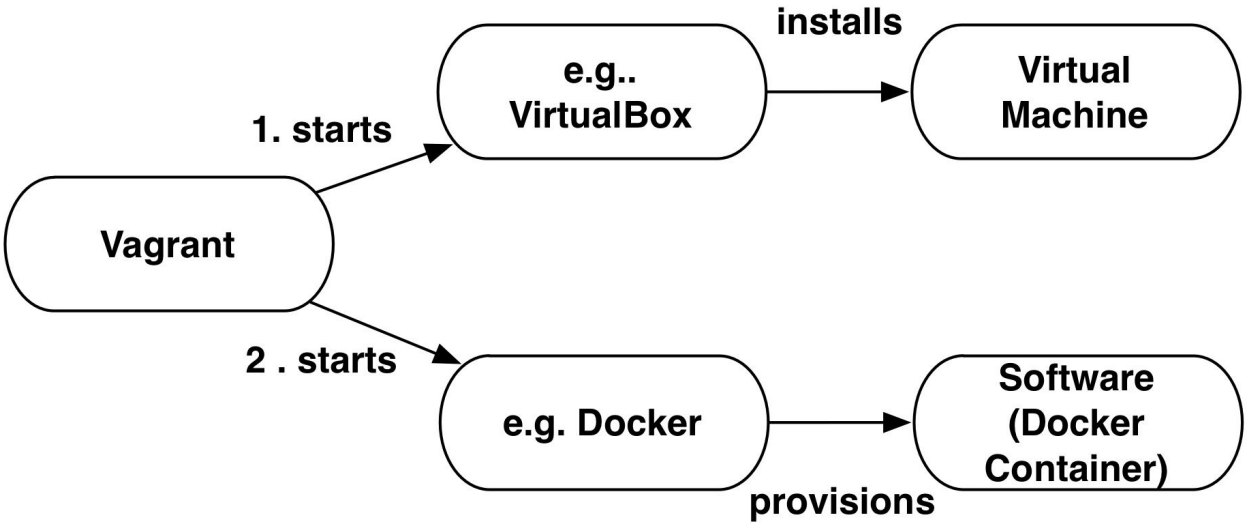


Fig. 71: How Vagrant works

To configure Vagrant a single file is necessary, the Vagrantfile. Listing 4 shows the Vagrantfile of the example application:

Listing 4: Vagrantfile from the example application

```
Vagrant.configure("2") do |config|
  config.vm.box = " ubuntu/trusty64"
  config.vm.synced_folder "../microservice-demo",
    "/microservice-demo", create: true
   config.vm.network "forwarded_port",
     guest: 8080, host: 18080
   config.vm.network "forwarded_port",
     guest: 8761, host: 18761
   config.vm.network "forwarded_port",
       guest: 8989, host: 18989

  config.vm.provision "docker" do |d|
    d.build_image "--tag=java /vagrant/java"
    d.build_image "--tag=eureka /vagrant/eureka"
    d.build_image
        "--tag=customer-app /vagrant/customer-app"
    d.build_image
        "--tag=catalog-app /vagrant/catalog-app"
    d.build_image "--tag=order-app /vagrant/order-app"
    d.build_image "--tag=turbine /vagrant/turbine"
    d.build_image "--tag=zuul /vagrant/zuul"
  end
  config.vm.provision "docker", run: "always" do |d|
    d.run "eureka",
      args: "-p 8761:8761"+
         " -v /microservice-demo:/microservice-demo"
    d.run "customer-app",
```

```
28        args: "-v /microservice-demo:/microservice-demo"+
29            " --link eureka:eureka"
30      d.run "catalog-app",
31        args: "-v /microservice-demo:/microservice-demo"+
32            " --link eureka:eureka"
33      d.run "order-app",
34        args: "-v /microservice-demo:/microservice-demo"+
35            " --link eureka:eureka"
36      d.run "zuul",
37        args: "-v /microservice-demo:/microservice-demo"+
38            " -p 8080:8080  --link eureka:eureka"
39      d.run "turbine",
40        args: "-v /microservice-demo:/microservice-demo"+
41            " --link eureka:eureka"
42    end
43
44  end
```

- **config.vm.box** selects a base image – in this case a Ubuntu-14.04 Linux installation (Trusty Tahr).
- **config.vm.synced_folder** mounts the directory containing the results of the Maven build into the virtual machine. In this manner the Docker Containers can directly make use of the build results.
- The ports of the virtual machine can be linked to the ports of the computer running the virtual machine. The **config.vm.network** settings can be used for that. In this manner applications in the Vagrant virtual machine become accessible as if running directly on the computer.
- **config.vm.provision** starts the part of the configuration, which deals with the software provisioning within the virtual machine. Docker serves as provisioning tool and is automatically installed within the virtual machine.
- Finally **d.build_image** generates the Docker images using Dockerfiles. First the base image java is created. Images for the three Microservices customer-app, catalog-app and order-app follow. The images for the Netflix technologies servers belong to the infrastructure: Eureka for Service Discovery, Turbine for monitoring and Zuul for routing of client requests.
- Vagrant starts the individual images using **d.run**. This step is not only performed when provisioning the virtual machine, but also when the system is started anew (**run: “always”**). The option **–v** mounts the directory **/microservice-demo** into each Docker Container so that the Docker Container can directly execute the compiled code. **-p** links a port of the Docker Container to a port of virtual machine. This link allows to access the Docker Container Eureka under the host name eureka from within the other Docker Containers.

In the Vagrant setup the JAR files containing the application code are not contained in the Docker image. The directory **/microservice-demo** does not belong to the Docker Container. It resides on the host running the Docker Containers i.e. the Vagrant VM. It would also be possible to copy these files into the Docker image. Afterwards the resulting image could be copied on a repository server and downloaded from there. Then the Docker Container would contain all necessary files to run the Microservice. A deployment in production then only needs to start the Docker images on a production server. This approach is used in the Docker Machine setup (see section 14.6).

Networking in the Example Application

Fig. 72 shows how the individual Microservices of the example application communicate via the network. All Docker Containers are accessible in the network via IP addresses from the 172.17.0.0/16 range. Docker generates such a network automatically and connects all Docker Containers to the network. Within the network all ports are accessible that are defined in the Dockerfiles using EXPOSE. The Vagrant virtual machine is also connected to this network. Via the Docker links (compare Listing 4) all Docker Containers know the Eureka container and can access it under the host name **eureka**. The other Microservices have to be looked up via Eureka. All further communication takes place via the IP address.

In addition the -p-Option in the **d.run** entries for the Docker Containers in Listing 4 has connected the ports to the Vagrant virtual machine. These Containers can be accessed via these ports of the Vagrant virtual machine. To reach them also from the computer running the Vagrant virtual machine there is a port mapping, which links the ports to the local computer. This is accomplished via the **config.vm.network** entries in Vagrantfile. The port 8080 of the Docker Container "zuul" can for instance be accessed via the port 8080 in the Vagrant virtual machine. This port can be reached from the local computer via the port 18080. So the URL http://localhost:18080/ accesses this Docker Container.

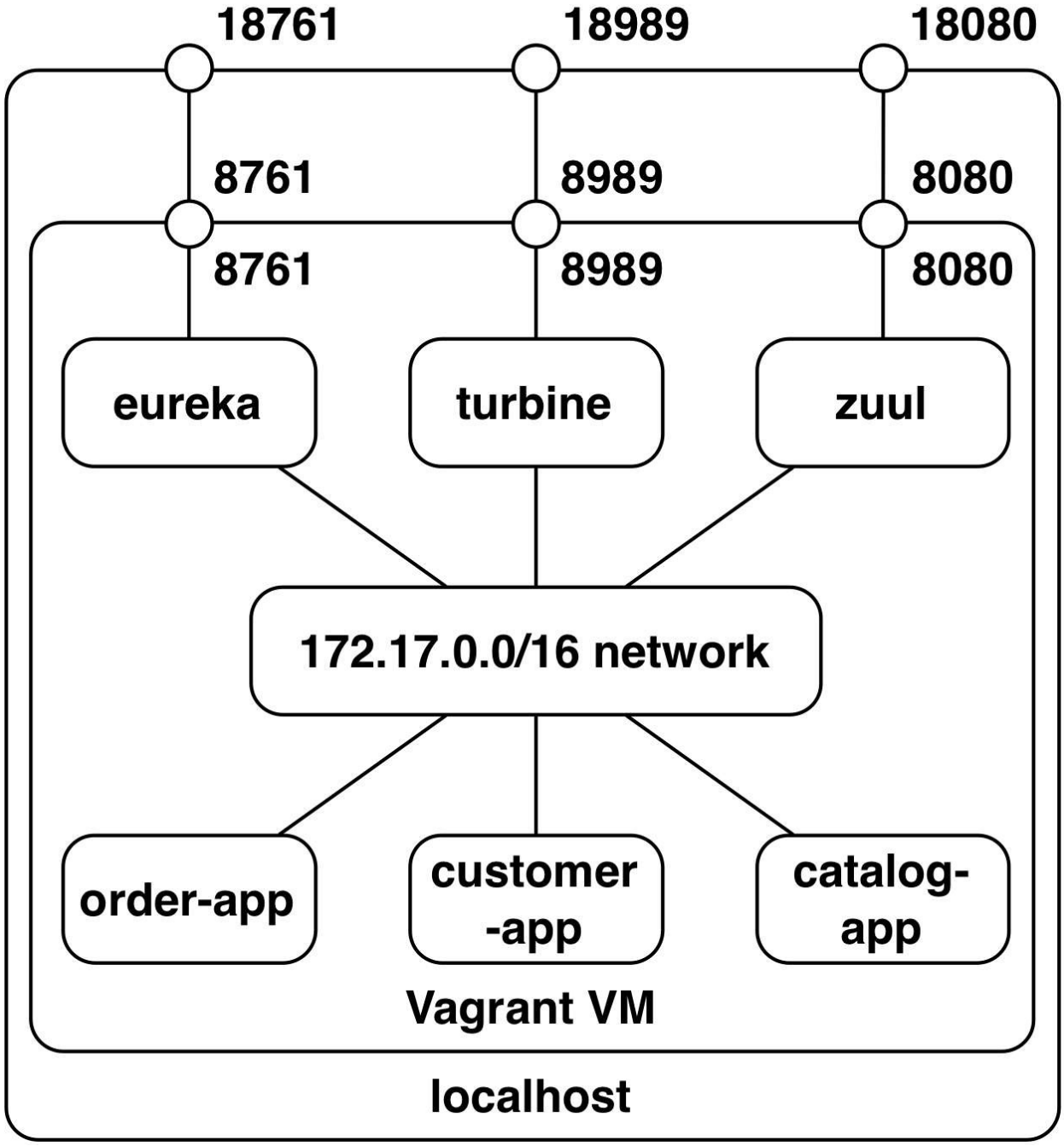


Fig. 72: Network and ports of the example application

Try and Experiment

Run the Example Application

The example application does not need much effort to make it run. A running example application lays the foundation for the experiments described later in this chapter.

One remark: The **Vagrantfile** defines how much RAM and how many CPUs the virtual machines gets. The settings **v.memory** and **v.cpus**, which are not shown in the Listing, deal with this. Depending on the used computer, the values should be increased if a lot of RAM or many CPUs are present. Whenever the values can be increased, they should be elevated in order to speed the application up.

The installation of Vagrant is described in http://docs.vagrantup.com/v2/installation/index.html. Vagrant needs a virtualization solution like VirtualBox. The installation of VirtualBox is explained at https://www.virtualbox.org/wiki/Downloads. Both tools are free.

The example can only be started once the code has been compiled. Instructions how to compile the code can be found in the experiment described in section 14.3. Afterwards you can change into the directory **docker-vagrant** and start the example demo using the command **vagrant up**.

To interact with the different Docker Containers you have to log into the virtual machine via the command **vagrant ssh**. This command has to be executed within the sub directory **docker-vagrant**. For this to be possible a ssh client has to be installed on the computer. On Linux and Mac OS X such a client is usually already present. In Windows installing git will bring an ssh client along as described at http://git-scm.com/download/win. Afterwards **vagrant ssh** should work.

Investigate Docker Containers

Docker contains several useful commands:

- **docker ps** provides an overview of the running Docker Containers.
- The command **docker log "name of Docker Container"** shows the logs.
- **docker log -f "name of Docker Container"** provides incessantly the up-to-date log information of the Container.
- **docker kill "name of the Docke Container"** terminates a Docker Container.
- **docker rm "name of the Docker Container"** deletes all data. For that all Containers first needs to be stopped.

After starting the application the log files of the individual Docker Containers can be looked at.

Update Docker Containers

A Docker Container can be terminated (**docker kill**) and the data of the Container deleted (**docker rm**). The commands have to be executed inside the Vagrant virtual machine. **vagrant provision** starts the missing Docker Containers again. This command has to be executed on the host running Vagrant. If you want to change the Docker Container, simply delete it, compile the code again and generate the system anew using **vagrant provision**.

Additional Vagrant commands:

- **vagrant halt** terminates the virtual machine.
- **vagrant up** starts it again.
- **vagrant destroy** destroys the virtual machine and all saved data.

Store Data on Disk

Right now the Docker Container does not save the data so that it is lost upon restarting. The used HSQLDB database can also save the data into a file. To achieve that a suitable HSQLDB URL has to be used, see http://hsqldb.org/doc/guide/dbproperties-chapt.html#dpc_connection_url . Spring Boot can read the JDBC URL out of the **application.properties** file, see http://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-sql.html#boot-features-connect-to-production-database . Now the Container can be restarted without data loss. But what happens if the Docker Container has to be generated again? Docker can save data also outside of the Container itself, compare https://docs.docker.com/userguide/dockervolumes/. These options provide a good basis for further experiments. Also another database than HSQLDB can be used such as MySQL. For that purpose another Docker Container has to be installed, which contains the database. In addition to adjusting the JDBC URL a JDBC driver has to be added to the project.

How is the Java Docker Image Built?

The Docker file is more complex than the ones discussed here. https://docs.docker.com/reference/builder/ demonstrates which commands are available in Dockerfiles. Try to understand the structure of the Dockerfiles.

14.6 Docker Machine

Vagrant serves to install environments on a developer laptop. In addition to Docker Vagrant can use e.g. simple shell scripts for deployment. However, for production environments this solution is unsuitable. Docker Machine is

specialized in Docker. It supports many more virtualization solutions as well as some Cloud providers.

Fig. 73 demonstrates how Docker Machine builds a Docker environment: First, using a virtualization solution like VirtualBox a virtual machine is installed. This virtual machine is based on boot2docker, a very lightweight Linux designed specifically as a running environment for Docker Containers. On that Docker Machine installs a current version of Docker. A command like **docker-machine create –driver virtualbox dev** generates for instance a new environment with the name dev running on a VirtualBox computer.

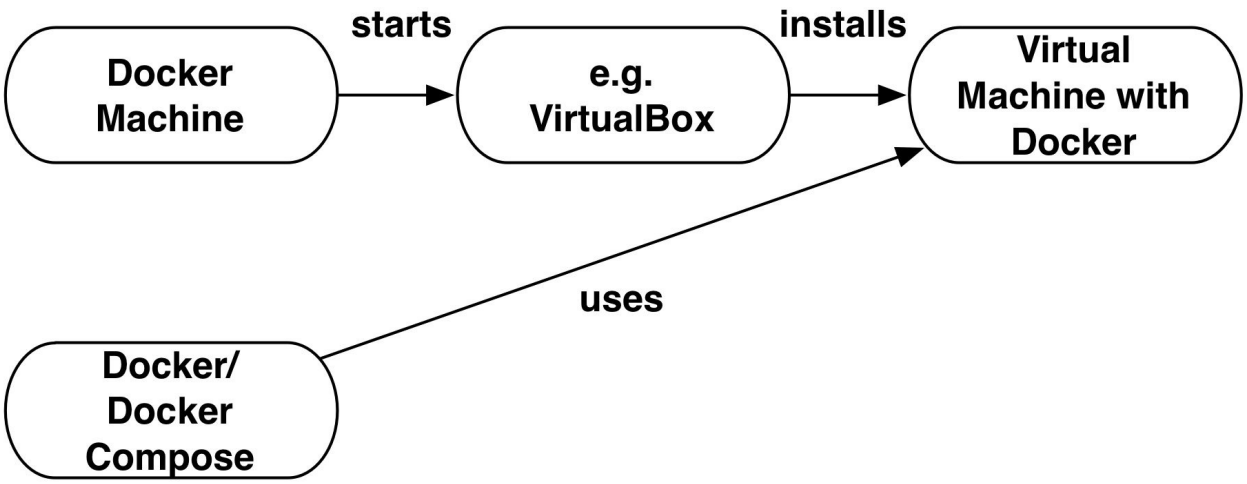


Fig. 73: Docker Machine

The Docker tool now can communicate with this environment. The Docker command line tools use a REST interface to communicate with the Docker server. Accordingly, the command line tool just has to be configured in a way that allows it to communicate with the server in a suitable manner. In Linux or Mac OS X the command **eval "$(docker-machine env dev)"** is sufficient to configure the Docker appropriately. For Windows PowerShell the command **docker-machine.exe env –shell powershell dev** must be used and in Windows cmd **docker-machine.exe env –shell cmd dev** .

Docker Machine renders it thus very easy to install one or several Docker environments. All the environments can be handled by Docker Machine and accessed by the Docker command line tool. As Docker Machine also supports technologies like Amazon Cloud or VMware vSphere, it can be used to generate production environments.

Try and Experiment

The example application can also run in an environment created by Docker Machine.

The installation of Docker Machine is described at https://docs.docker.com/machine/#installation. Docker Machine requires a virtualization solution like VirtualBox. How to install VirtualBox can be found at https://www.virtualbox.org/wiki/Downloads. Using **docker-machine create –virtualbox-memory "4096" –driver virtualbox dev** a Docker environment called **dev** can now be created on a Virtual Box. Without any further configuration the storage space is set to 1 GB, which is not sufficient for a larger number of Java Virtual Machines.

docker-machine without parameters displays a help text, and **docker-machine create** shows the options for the generation of a new environment. https://docs.docker.com/machine/get-started-cloud/ demonstrates how Docker Machine can be used in a Cloud. This means that the example application can also easily be started in a Cloud environment.

At the end of your experiments **docker-machine rm** deletes the environment

14.7 Docker Compose

A Microservice-based system comprises typically several Docker Containers. These have to be generated together and need to be put into production simultaneously.

This can be achieved with Docker Compose. It enables the definition of Docker Containers, which each house one service. YAML serves as format.

Listing 5: Docker compose configuration for the example application

```
1  eureka:
2     build: ../microservice-demo/microservice-demo-eureka-server
3     ports:
4        - "8761:8761"
5  customer:
6     build: ../microservice-demo/microservice-demo-customer
7     links:
8       - eureka
9  catalog:
10    build: ../microservice-demo/microservice-demo-catalog
11    links:
12      - eureka
13 order:
14    build: ../microservice-demo/microservice-demo-order
15    links:
16      - eureka
17 zuul:
18    build: ../microservice-demo/microservice-demo-zuul-server
```

```
19    links:
20      - eureka
21    ports:
22       - "8080:8080"
23 turbine:
24    build: ../microservice-demo/microservice-demo-turbine-server
25    links:
26      - eureka
27    ports:
28       - "8989:8989"
```

Listing 5 shows the configuration of the example application. It consists of the different services. **build** references the directory containing the Dockerfile. The Dockerfile is used to generate the image for the service. **links** defines which additional Docker Containers the respective Container should be able to access. All Containers can access the Eureka Container under the name **eureka**. In contrast to the Vagrant configuration there is no Java base image, which contains only a Java installation. This is because, Docker Compose supports only containers which really offer a service. Therefore this base image has to be downloaded from the Internet. Besides, in case of the Docker Compose containers the JAR files are copied into the Docker images so that the images contain everything for starting the Microservices.

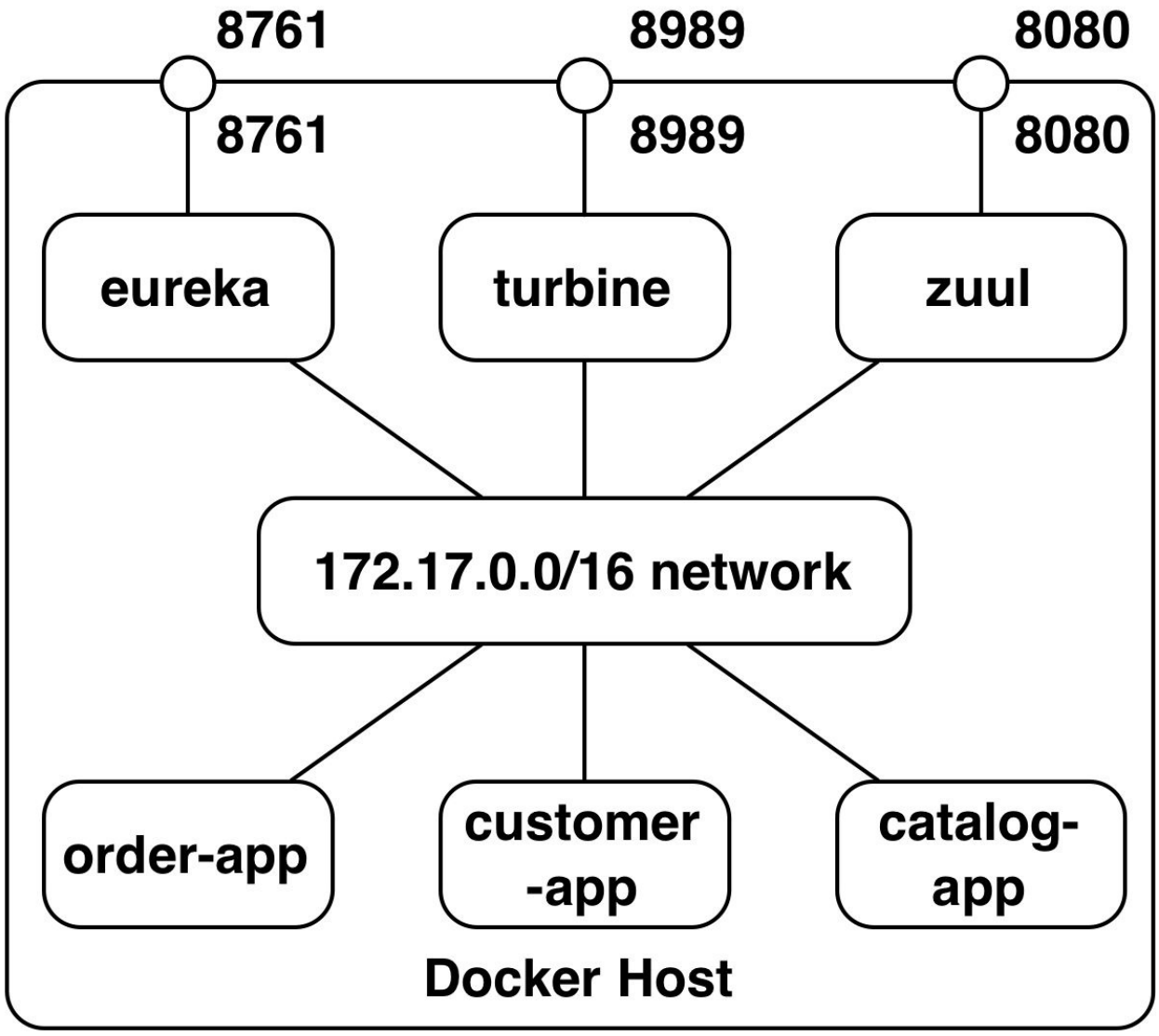


Fig. 74: Network for Docker Compose

The resulting system is very similar to the Vagrant system (Fig. 74). The Docker containers are linked via their own private network. From the outside only Zuul can be accessed for the processing of requests and Eureka for the dashboard. The are running directly on a host that then can be accessed from the outside.

Using **docker-compose build** the system is created based on the Docker Compose configuration. Thus the suitable Images are generated. **docker-compose up** then starts the system. Docker Compose uses the same settings as the Docker command line tool. So it can also work together with Docker Machine. Thus it is transparent whether the system is generated on a local virtual machine or somewhere in the Cloud.

Try and Experiment

Run the Example with Docker Compose

The example application possesses a suitable Docker Compose configuration. Upon the generation of an environment with Docker Machine Docker Compose can be used to create the Docker containers. README.md in the directory docker describes the necessary procedure.

Scale the Application

Have a look at the **docker-compose scale** command. It can scale the environment. Services can be restarted, logs can be analyzed and finally stopped. Once you have started the application, you can test these functionalities.

Cluster Environments for Docker

Mesos (http://mesos.apache.org/) together with Mesosphere (http://mesosphere.com/), Kubernetes (http://kubernetes.io/) or CoreOS (http://coreos.com/) offers similar options as Docker Compose and Docker Machine, however they are meant for servers and server clusters. The Docker Compose and Docker Machine configurations can provide a good basis for running the application on these platforms.

14.8 Service Discovery

Section 8.9 introduced the general principles of Service Discovery. The example application uses Eureka for Service Discovery.

Eureka is a REST-based server, which allows services to register themselves so that other services can request their location in the network. In essence, each service can register a URL under its name. Other services can find the URL by the name of the service. Using this URL other services can then send REST messages to this service.

Eureka supports replication onto several servers and caches on the client. This makes the system fail-safe against the failure of individual Eureka servers and allows to answer requests rapidly. Changes to data have to be replicated to all

servers. Accordingly, it can take some time till they are really updated everywhere. During this time the data is inconsistent: Each server has a different version of the data.

In addition Eureka supports Amazon Web Services because Netflix uses it in this environment. Eureka can for instance quite easily be combined with Amazon's scaling.

Eureka monitors the registered services and removes them from the server list if they cannot be reached anymore by the Eureka server.

Eureka is the basis for many other services of the Netflix Stack and for Spring Cloud. Through a uniform Service Discovery other aspects such as routing can easily be implemented.

Eureka Client

For a Spring Boot application to be able to register with a Eureka server and to find other Microservices, the application has to be annotated with **@EnableDiscoveryClient** or **@EnableEurekaClient**. In addition a dependency from **spring-cloud-starter-eureka** has to be included in the **pom.xml**. The application registers automatically with the Eureka server and can access other Microservices. The example application accesses other Microservices via a load balancer. This is described in detail in section 14.11.

Configuration

Configuring the application is necessary to define for instance the Eureka server to be used. The file **application.properties** (Listing 6) is used for that. Spring Boot reads it out automatically in order to configure the application. This mechanism can also be used to configure one's own code. In the example application the values serve to configure the Eureka client:

- The first line defines the Eureka-Server. The example application uses the Docker link, which provides the Eureka server under the host name "eureka".
- **leaseRenewalIntervalInSeconds** determines how often data are updated between client and server. As the data have to be held locally in a cache on each client, a new service first needs to create its own cache and replicate it onto the server. Afterwards the data are replicated onto the clients. Within a test environment it is important to track system changes rapidly so that the example application uses five seconds instead of the preset value of 30

seconds. In production with many clients this value should be increased. Otherwise the updates of information will use a lot of resources, even though the information remains essentially unchanged.

- **spring.application.name** serves as name for the service during the registration at Eureka. During registration the name is converted into capital letters. This service would thus be known by Eureka under the name “CUSTOMER”.
- There can be several instances of each service to achieve fail over and load balancing. The **instanceId** has to be unique for each instance of a service. Because of that it contains a random number, which ensures unambiguousness.
- **preferIpAddress** makes sure that Microservices register with their IP address and not with their host name. In a Docker environment host names are unfortunately not easily resolvable by other hosts. This problem is circumvented by the use of IP addresses.

Listing 6: Part of **application.properties** with Eureka configuration

```
1 eureka.client.serviceUrl.defaultZone=http://eureka:8761/eureka/
2 eureka.instance.leaseRenewalIntervalInSeconds=5
3 spring.application.name=catalog
4 eureka.instance.metadataMap.instanceId=catalog:${random.value}
5 eureka.instance.preferIpAddress=true
```

Eureka Server

The Eureka server (Listing 7) is a simple Spring Boot application, which turns into a Eureka server via the **@EnableEurekaServer** annotation. In addition the server needs a dependency on **spring-cloud-starter-eureka-server**.

Listing 7: Eureka Server

```
1 @EnableEurekaServer
2 @EnableAutoConfiguration
3 public class EurekaApplication {
4   public static void main(String[] args) {
5     SpringApplication.run(EurekaApplication.class,
6       args);
7   }
8 }
```

The Eureka server offers a dashboard, which shows the registered services. In the example application this can be found at http://localhost:18761/ (Vagrant) or on Docker host under port 8761 (Docker Compose). Fig. 75 shows a screenshot of the Eureka Dashboards for the example application. The three Microservices and

the Zuul-Proxy, which is discussed in the next section, are present on the dashboard.

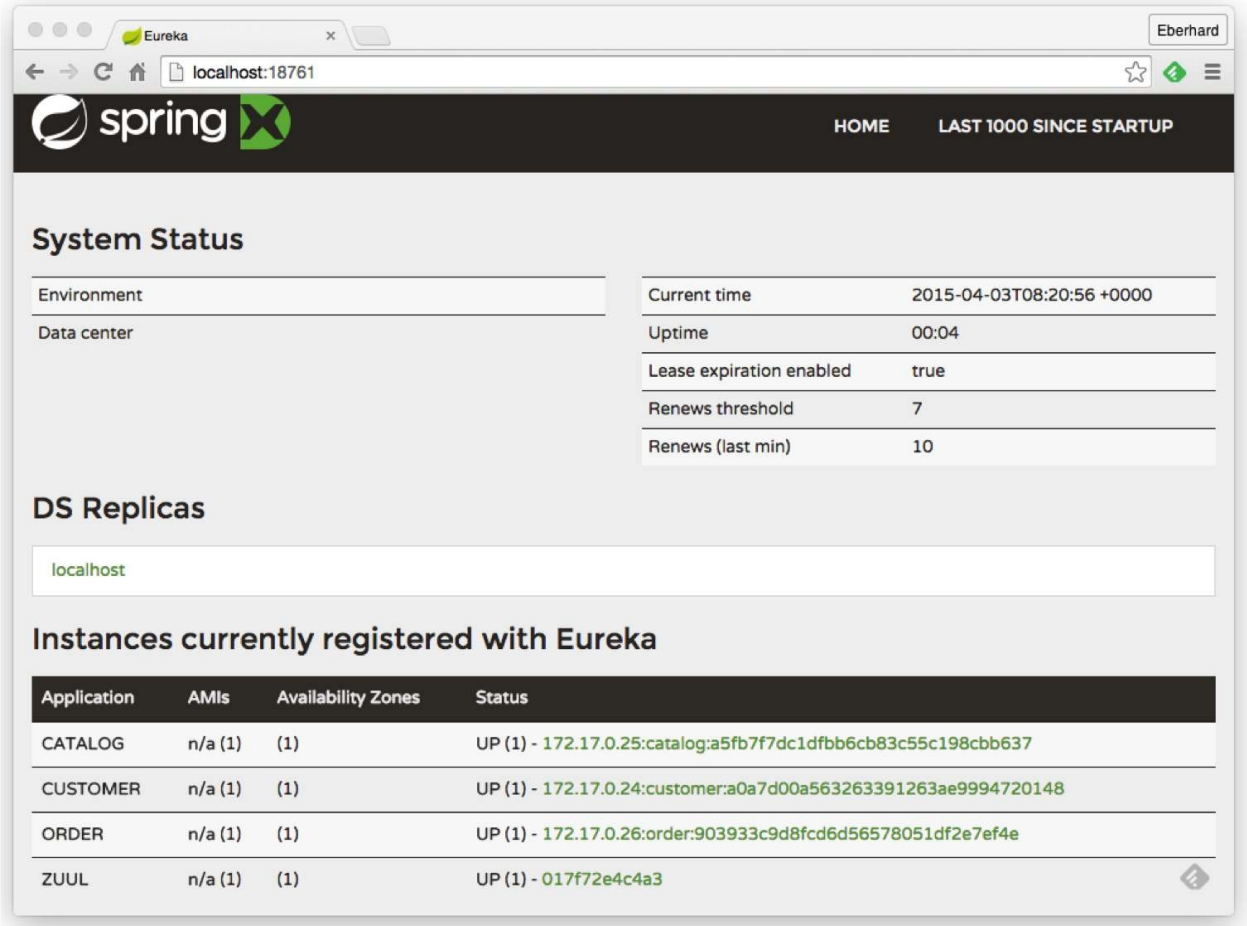


Fig. 75: Eureka Dashboard

14.9 Communication

Chapter 9 explained how Microservices communicate with each other and can be integrated. The example application uses REST for internal communication. The REST end points can be contacted from outside, however the web interface the system offers is of far greater importance. The REST implementation uses HATEOAS. The list containing all orders for instance contains links to the individual orders. This is automatically implemented by Spring Data REST. However, there are no links to the customer and the items of the order.

Using HATEOAS can go further: The JSON can contain a link to an HTML document for each order – and vice versa. In this way a JSON-REST-based service can generate links to HTML pages to display or modify data. Such HTML code can for instance present an item in an order. As the catalog team provides the HTML code for the item, the catalog team itself can introduce changes to the presentation – even if the items are displayed in another module.

REST is also of use here: HTML and JSON are really only representations of the same resource that can be addressed by a URL. Via Content Negotiation the right resource representation as JSON or HTML can be selected (compare section 9.2).

Zuul: Routing

The Zuul Proxy transfers incoming requests to the respective Microservices. The Zuul Proxy is a separate Java process. To the outside only one URL is visible, however internally the calls are processed by different Microservices. This allows the system to internally change the structure of the Microservices, while still offering a URL to the outside. In addition Zuul can provide web resources. In the example Zuul provides the first HTML page viewed by the user.

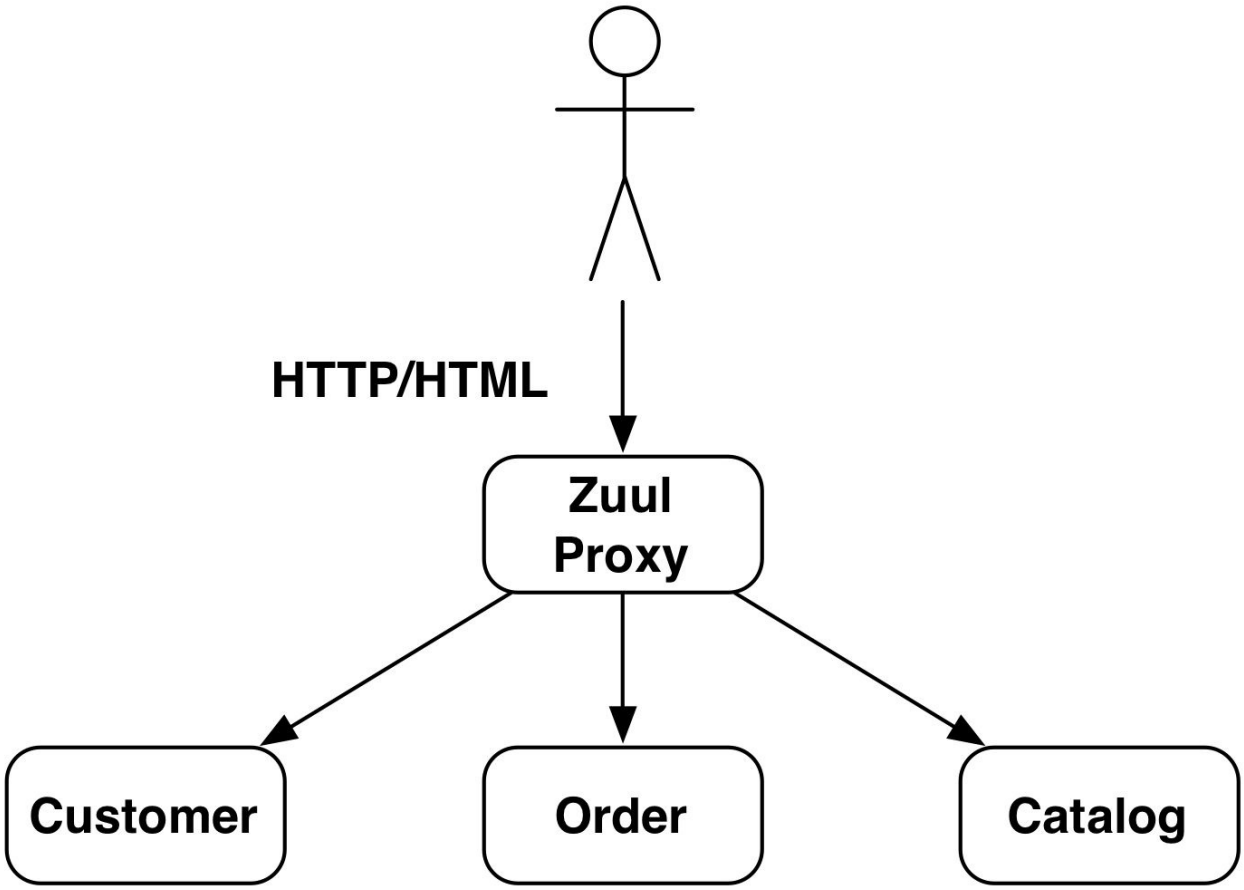


Fig. 76: Zuul-Proxy in the example application

Zuul needs to know which requests to transfer to which Microservice. Without additional configuration Eureka will forward a request to a URL starting with "/customer" to the Microservice called CUSTOMER. This renders the internal Microservice names visible to the outside. But this routing can also be configured differently. Moreover Zuul filters can change the requests in order to implement general aspects in the system. There is for instance an integration with Spring Cloud Security to pass on security tokens to the Microservices. Such filters can also be used to pass on certain requests to specific servers. This allows for instance to transfer requests to servers having additional analysis options for investigating error situations. In addition a part of a Microservice functionality can be replaced by another Microservice.

Implementing the Zuul-Proxy server with Spring Cloud is very easy and analogous to the Eureka server presented in Listing 7. Instead of **@EnableEurekaServer** it is **@EnableZuulProxy**, which activates the Zuul-Proxy. As additional dependency **spring-cloud-starter-zuul** has to be added to the application, for

instance within the Maven build configuration, which then integrates the remaining dependencies of Zuul into the application.

A Zuul server represents an alternative to a Zuul Proxy. It does not have routing built-in, but uses filters instead. A Zuul server is activated by **@EnableZuulServer**.

Try and Experiment

Add Links to Customer and Items

Extend the application so that an order contains also links to the customer and to the items and thus implements HATEOAS better. Supplement the JSON documents for customer, items and orders with links to the forms.

Use the Catalog Service to Show Items in Orders

Change the order presentation so that HTML from the Catalog service is used for items. To do so, you have to insert suitable JavaScript code into the order component, which loads HTML code from the Catalog.

Implement Zuul Filters

Implement your own Zuul filter (compare https://github.com/Netflix/zuul/wiki/Writing-Filters). The filter can for instance only release the requests. Introduce an additional routing to an external URL. For instance /google could redirect to http://google.com/. Compare the

Spring Cloud reference documentation .

Authentication and Authorization

Insert an authentication and authorization with Spring Cloud Security. Compare http://cloud.spring.io/spring-cloud-security/.

14.10 Resilience

Resilience means that Microservices can deal with the failure of other Microservices. Even if a called Microservice is not available, they will still work. Section10.5 presented this topic.

The example application implements Resilience with Hystrix. This library protects calls so that no problems arise if a system fails. When a call is protected by Hystrix, it is executed in a different thread than the call itself. This thread is taken from a distinct thread pool. This makes it comparatively easy to implement a timeout during a call.

Circuit Breaker

In addition Hystrix implements a Circuit Breaker. If a call causes an error, the Circuit Breaker will open after a certain number of errors. In that case subsequent calls are not directed to the called system anymore, but generate an error immediately. After a sleep window the Circuit Breaker closes so that calls are directed to the actual system again. The exact behavior can be configured. In the configuration the error threshold percentage can be determined. That is the percentage of calls which have to cause an error within the time window for the Circuit Breaker to open. Also the sleep window can be defined, in which the Circuit Breaker is open and not sending calls to the system.

Hystrix with Annotations

Spring Cloud uses Java Annotations from the project hystrix-javanica for the configuration of Hystrix. This project is part of hystrix-contrib . The annotated methods are protected according to the setting in the Annotation. Without this approach Hystrix commands would have to be written, which is a lot more effort than just adding some Annotations to a Java method.

To be able to use Hystrix within a Spring Cloud application, the application has to be annotated with **@EnableCircuitBreaker** resp. **@EnableHystrix**. Moreover, the project needs to contain a dependency to **spring-cloud-starter-hystrix**.

Listing 8 shows a section from the class **CatalogClient** of the Order Microservice from the example application. The method **findAll()** is annotated with **@HystrixCommand**. This activates the processing in a different thread and the Circuit Breaker. The Circuit Breaker can be configured – in the example the number of calls, which have to cause an error in order to open the Circuit Breaker,

is set to 2. In addition the example defines a **fallbackMethod**. Hystrix calls this method if the original method generates an error. The logic in **findAll()** saves the last result in a cache, which is returned by the **fallbackMethod** without calling the real system. In this way a reply can still be returned when the called Microservice fails, however this reply might no longer be up-to-date.

Listing 8: Example for a method protected by Hystrix

```
@HystrixCommand(
 fallbackMethod = "getItemsCache",
 commandProperties = {
 @HystrixProperty(
  name = "circuitBreaker.requestVolumeThreshold",
  value = "2") })
public Collection<Item> findAll() {
  this.itemsCache = ...
  ...
  return pagedResources.getContent();
}

private Collection<Item> getItemsCache() {
  return itemsCache;
}
```

Monitoring with the Hystrix Dashboard

Whether a Circuit Breaker is currently open or closed, gives an indication of how well a system is running. Hystrix offers data to monitor this. A Hystrix system provides such data as a stream of JSON documents via HTTP. The Hystrix Dashboard can visualize the data in a web interface. The dashboard presents all Circuit Breakers along with the number of requests and their state (open/closed) (Fig. 77). In addition it displays the state of the thread pools.

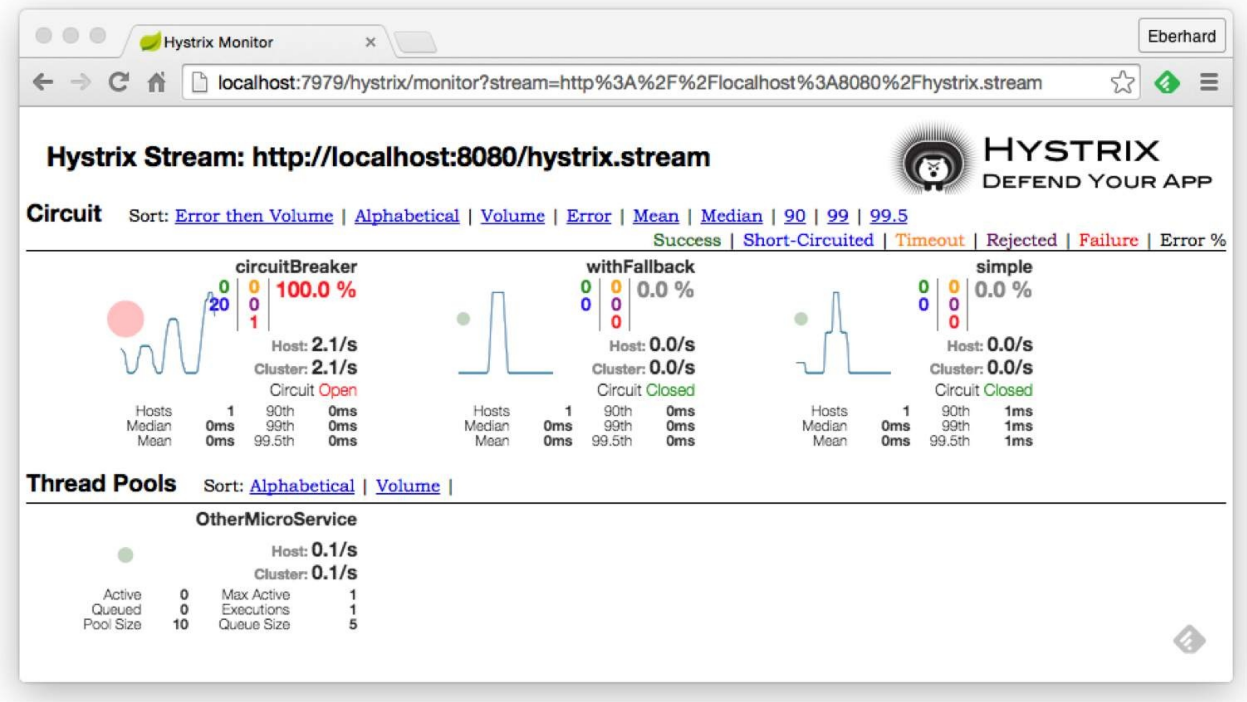


Fig. 77: Example for a Hystrix Dashboard

A Spring Boot Application needs to have the annotation **@EnableHystrixDashboard** and a dependency to **spring-cloud-starter-hystrix-dashboard** to be able to display a Hystrix Dashboard. That way any Spring Boot application might in addition show a Hystrix Dashboard or the dashboard can be implemented in an application by itself.

Turbine

In a complex Microservices environment it is not useful that each instance of a Microservice visualizes the information concerning the state of its Hystrix Circuit Breaker. The state of all Circuit Breakers in the entire system should be summarized on a single dashboard. To visualize the data of the different Hystrix systems on one dashboard there is the Turbine project. Fig. 78 illustrates the approach Turbine takes: The different streams of the Hystrix enabled Microservices are provided at URLs like http://<host:port>/hystrix.stream. The Turbine server requests them and provides them in a consolidated manner at the URL http://<host:port>/turbine.stream. This URL can be used by the dashboard in order to display the information of all Circuit Breakers of the different Microservice instances.

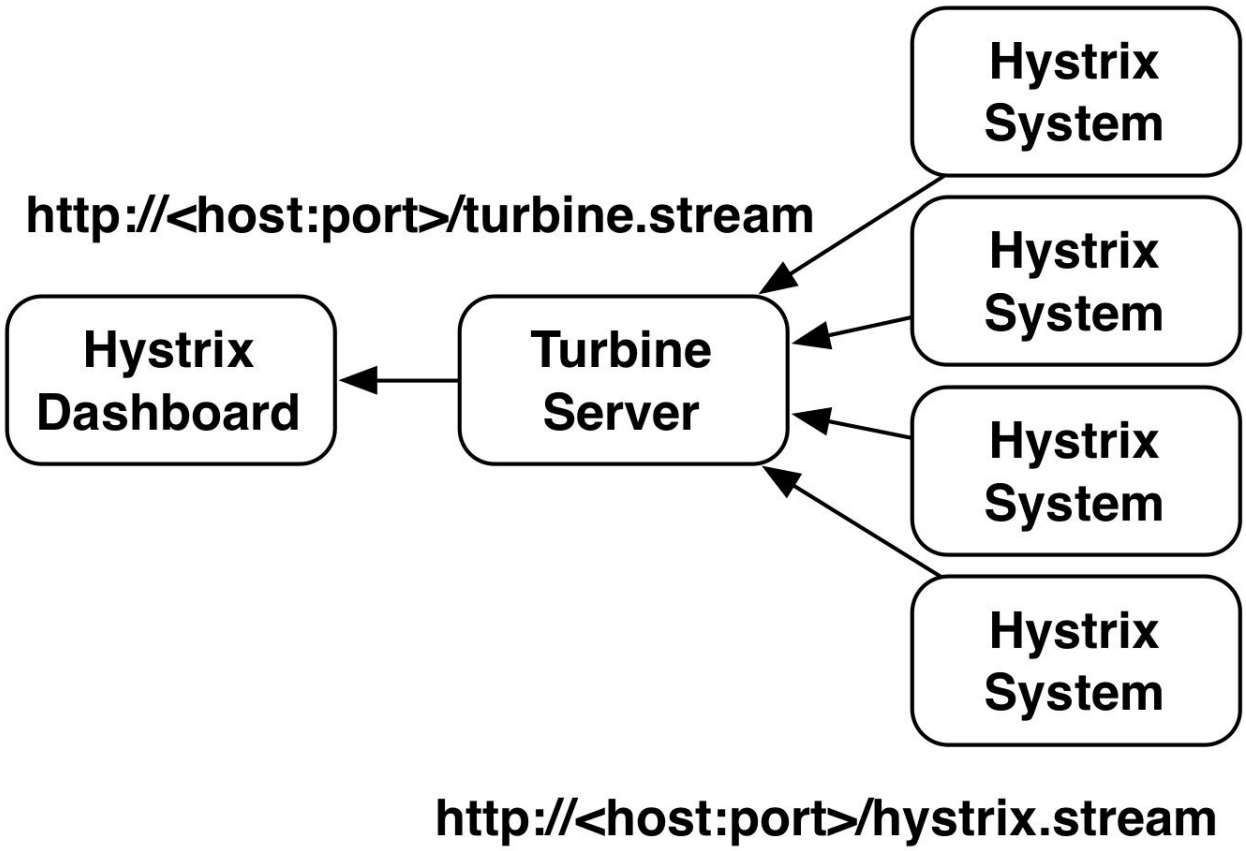


Fig. 78: Turbine consolidates Hystrix monitoring data.

Turbine runs in a separate process. With Spring Boot the Turbine server is a simple application, which is annotated with **@EnableTurbine** and **@EnableEurekaClient**. In the example application it has the additional annotation **@EnableHystrixDashboard** so that it also displays the Hystrix Dashboard. It also needs a dependency on **spring-cloud-starter-turbine**.

Which data are consolidated by the Turbine server is determined by the configuration of the application. Listing 9 shows the configuration of the Turbine servers of the example project. It serves as a configuration for a Spring Boot application just like **application.properties** files, but is written in YAML. The configuration sets the value "ORDER" for **turbine.aggregator.clusterConfig**. This is the application name in Eureka. **turbine.aggregator.appConfig** is the name of the data stream in the Turbine server. In the Hystrix Dashboard a URL like http://172.17.0.10:8989/turbine.stream?cluster=ORDER has to be used in visualize the data stream. Part of the URL is the IP-Adresse of the Turbine server, which can be found in the Eureka Dashboard. The dashboard accesses the Turbine server via the network between the Docker containers.

Listing 9: Configuration application.yml

```
turbine:
 aggregator:
  clusterConfig: ORDER
 appConfig: order
```

Try and Experiment

Terminate Microservices

Using the example application generate a number of orders. Find the name of the Catalog Docker Container using **docke ps**. Stop the Catalog Docker Container with **docker kill**. This use is protected by Hystrix. What happens?

What happens if the Customer Docker Container is terminated as well? The use of this Microservice is not protected by Hystrix.

Add Hystrix to Customer Microservice

Protect the use of the Customer Docker Container also with Hystrix. In order to do so change the class **CustomerClient** from the Order project. **CatalogClient** can serve as a template.

Change Hystrix Configuration

Change the configuration of Hystrix for the Catalog Microservice. There are several configuration options. Listing 8 (CatalogClient from the Order-Project) shows the use of the Hystrix annotations. Other time intervals for opening and closing of the Circuit Breakers are for instance a possible change.

14.11 Load Balancing

For Load Balancing the example application uses Ribbon. Many Load Balancers are proxy-based. In this model the clients send all calls to a Load Balancer. The Load Balancer runs as a distinct server and forwards the request to a web server – often depending on the current load of the web servers.

Ribbon implements a different model called Client Side Load Balancing: The client has all the information to communicate with the right server. The client calls the server directly and distributes the load by itself to different servers. In the architecture there is no bottle neck as there is no central server all calls would have to pass. In conjunction with data replication by Eureka Ribbon is quite resilient: As long as the client runs, it can send requests. The failure of a Proxy Load Balancer would stop all calls to the server.

Dynamic scaling is very simple within this system: A new instance is started, enlists itself at Eureka and then the Ribbon Clients redirect load to the instance.

As already discussed in the section dealing with Eureka (Section 14.8), data can be inconsistent over the different servers. Because data are not up-to-date, servers can be contacted, which really should be left out by the Load Balancing.

Ribbon with Spring Cloud

Spring Cloud simplifies the use of Ribbon. The application has to be annotated with **@RibbonClient**. While doing so, a name for the application can be defined. In addition the application needs to have a dependency on **spring-cloud-starter-ribbon**. In that case an instance of a Microserve can be accessed using code like in Listing 10. For that purpose the code uses the Eureka name of the Microservice.

Listing 10: Determining a server with Ribbon Load Balancing

```
ServiceInstance instance
  = loadBalancer.choose("CATALOG");
String url = "http://" +
  instance.getHost() + ":" +
  instance.getPort() +
  "/catalog/";
```

The use can also be transparent to a large extent. To illustrate this Listing 11 shows the use of **RestTemplates** with Ribbon. This is a Spring class, which can be used to call REST services. In the Listing the **RestTemplate** of Spring is injected into the object as it is annotated with **@Autowired**. The call in **callMicroservice()** looks like it is contacting a server called “stores”. In reality this name is used to search a server at Eureka, and to this server the REST call is sent. This is done via Ribbon so that the load is also distributed across the available servers.

Listing 11: Using Ribbon with RestTemplate

```
@RibbonClient(name = "ribbonApp")
  … // Left out other Spring Cloud / Boot Annotations
```

```
3 public class RibbonApp {
4
5  @Autowired
6    private RestTemplate restTemplate;
7
8    public void callMicroservice() {
9       Store store = restTemplate.
10       getForObject("http://stores/store/1", Store.class);
11   }
12
13 }
```

Try and Experiment

Load Balance to an Additional Service Instance

The Order Microservice distributes the load onto several instances of the Customer and Catalog Microservice – if several instances exist. Without further measures, only a single instance is started. The Order Microservice shows in the log which Catalog or Customer Microservice it contacts. Initiate an order and observe which Services are contacted.

Afterwards start an additional Catalog Microservice. You can do that using the command: **docker run -v /microservice-demo:/microservice-demo –link eureka:eureka catalog-app** in Vagrant. For Docker Compose **docker-compose scale catalog=2** should be enough. Verify whether the container is running and observe the log output.

For reference: Try and Experiment in section 14.4 shows the main commands for using Docker. Section 14.7 shows how to use Docker Compose.

Create Data

Create a new dataset with a new item. Is the item always displayed in the selection of items? Hint: The database runs within the process of the Microservice – i.e. each Microservice instance possesses its own database.

14.12 Integrating Other Technologies

Spring Cloud and the entire Netflix Stack are based on Java. Thus, it seems impossible for other programming languages and platforms to use this infrastructure. However, there is a solution: The application can be supplied with a sidecar. The sidecar is written in Java and uses Java libraries to integrate into a Netflix-based infrastructure. The sidecar for instance takes care of registration

and finding other Microservices in Eureka. Netflix itself offers for this purpose the Prana project. The Spring Cloud solution is explained in the documentation. The sidecar runs in a distinct process and serves as an interface between the Microservice itself and the Microservice infrastructure. In this manner other programming languages and platforms can be easily integrated into a Netflix or Spring Cloud environment.

14.13 Tests

The example application contains test applications for the developers of Microservices. These do not need a Microservice infrastructure or additional Microservices – in contrast to the production system. This allows developers to run each Microservice without a complex infrastructure.

The class **OrderTestApp** in the Order project contains such a test application. The applications contain their own configuration file **application-test.properties** with specific settings within the directory **src/test/resources**. The settings prevent that the applications register with the Service Discovery Eureka. Besides they contain different URLs for the dependent Microservices. This configuration is automatically used by the test application as it uses a Spring profile called “test”. All JUnit tests use these settings as well so that they can run without dependent services.

Stubs

The URLs for the dependent Microservices in the test application and the JUnit tests point to Stubs. These are simplified Microservices, which only offer a part of the functionalities. They run within the same Java process as the real Microservices or JUnit tests. So only a single Java process has to be started for the development of a Microservice, analogous to the usual way of developing with Java. The Stubs can be implemented differently – for instance using a different programming language or even a web server, which returns certain static documents representing the test data (compare section 11.6). Such approaches might be better suited for real-life applications.

Stubs facilitate development. If each developer needs to use a complete environment including all Microservices during development, a tremendous amount of hardware resources and a lot of effort to keep the environment continuously up-to-date would be necessary. The Stubs circumvent this problem as no dependent Microservices are needed during development. Due to the stubs

the effort to start a Microservice is hardly bigger than the one for a regular Java application.

In a real project the teams can implement Stubs together with the real Microservices. The Customer team can implement in addition to the real service a stub for the Customer Microservice, which is used by the other Microservices for development. This ensures that the stub largely resembles the Microservice and is updated if the original service is changed. The Stub can be taken care of in a different Maven projects, which can be used by the other teams.

Consumer-driven Contract Test

It has to be ensured that the Stubs behave like the Microservices they simulate. In addition a Microservice has to define the expectations regarding the interface of a different Microservice. This is achieved by Consumer-driven Contract Tests (compare section 11.7). These are written by the team, which uses the Microservices. In the example this is the team, which is responsible for the Order Microservice. In the Order Microservice the Consumer-driven Contract Tests are found in the classes **CatalogConsumerDrivenContractTest** and **CustomerConsumerDrivenContractTest**. They run there to test the stubs of the Customer and Catalog Microservice for correctness.

Even more important than the correct functioning of the stubs is the correct functioning of the Microservices themselves. For that reason the Consumer-driven Contract Tests are also contained in the Customer and Catalog project. There they run against the implemented Microservices. This ensures that the Stubs as well as the real Microservices are in line with this specification. In case the interface is supposed to be changed, these tests can be used to confirm that the change does not break the calling Microservice. It is up to the used Microservices – Customer and Catalog in the example –, to comply with these tests. In this manner the requirements of the Order Microservice in regards to the Customer and Catalog Microservice can be formally defined and tested. The Consumer-driven Contract Tests serve in the end as formal definition of the agreed interface.

In the example application the Consumer-driven Contract Tests are part of the Customer and Catalog projects in order to verify that the interface is correctly implemented. Besides they are part of the Order project for verifying the correct functioning of the stubs. In a real project copying the tests should be prevented. The Consumer-driven Contract Tests can be located in one project together with the tested Microservices. Then all teams need to have access to the Microservice

projects to be able to alter the tests. Alternatively, they are located within the projects of the different teams, which are using the Microservice. In that case the tested Microservice has to collect the tests from the other projects and execute them.

In a real project it is not really necessary to protect stubs by Consumer-driven Contract Tests, especially as it is the purpose of the stubs to offer an easier implementation than the real Microservices. Thus the functionalities will be different and conflict with Consumer-driven Contract Tests.

Try and Experiment

Insert a field into Catalog or Customer data. Is the system still working? Why?

Delete a field in the implementation of the server for Catalog or Customer. Where is the problem noticed? Why?

Replace the home grown stubs with stubs, which use a tool from Section 11.6.

Replace the Consumer-driven Contract Tests with tests, which use a tool from Section 11.7.

Experiences with JVM-based Microservices in the Amazon Cloud (Sascha Möllering)

by Sascha Möllering, zanox AG

During the last months zanox has implemented a light-weight Microservices architecture in Amazon Web Services (AWS), which runs in several AWS regions. Regions divide the Amazon Cloud into sections like US-East or EU-West, which each have their own data centers. They work completely independently of each

other and do not exchange any data directly. Different AWS regions are used because latency is very important for this type of application and is minimized by latency-based routing. In addition it was a fundamental aim to design the architecture in an event-driven manner. Furthermore, the individual services were intended not to communicate directly, but rather to be separated by message queues resp. bus systems. An Apache Kafka cluster as message bus in the zanox data center serves as central point of synchronization for the different regions. Each service is implemented as a stateless application. The state is stored in external systems like the bus systems, Amazon ElastiCache (based on the NoSQL database Redis), the data stream processing technology Amazon Kinesis and the NoSQL database Amazon DynamoDB. The JVM serves as basis for the implementation of the individual services. We chose Vert.x and the embedded web server Jetty as frameworks. We developed all applications as self-contained services so that a Fat JAR, which can easily be started via **java –jar**, is generated at the end of the build process.

There is no need to install any additional components or an application server. Vert.x serves as basis framework for the HTTP part of the architecture. Within the application work is performed almost completely asynchronously to achieve high performance. For the remaining components we use Jetty as framework: These act either as Kafka/Kinesis consumer or update the Redis cache for the HTTP layer. All called applications are delivered in Docker Containers. This allows the use of a uniform deployment mechanism independent of the utilized technology. To be able to deliver the services independently in the different regions, an individual Docker Registry storing the Docker images in a S3 bucket was implemented in each region. S3 is a service that allows the storage of large file on Amazon server.

If you intend to use Cloud Services, you have to address the question whether you want to use the managed services of a Cloud provider or develop and run the infrastructure yourself. zanox decided to use the managed services of a Cloud provider because building and administrating proprietary infrastructure modules does not provide any business value. The EC2 computers of the Amazon portfolio are pure infrastructure. IAM on the other hand offers comprehensive security mechanisms. In the deployed services the AWS Java SDK is used, which allows in combination with IAM roles for EC2 to generate applications, which are able to access the managed services of AWS without using explicit credentials. During initial bootstrapping an IAM role containing the necessary permissions is assigned to an EC2 instance. Via the Metadata Service the AWS SDK is given the necessary credentials. This enables the application to access the managed

services defined in the role. Thus, an application can be implemented, which sends metrics to the monitoring system Amazon Cloud Watch and events to the data streaming processing solution Amazon Kinesis without having to role out explicit credentials together with the application.

All applications are equipped with REST interfaces for heartbeats and healthchecks so that the application itself as well as the infrastructure necessary for the availability of the application can be monitored at all times: Each application uses healthchecks to monitor the infrastructure components it uses. Application scaling is implemented via Elastic Load Balancing (ELB) and AutoScaling to be able to achieve a fine-grained application depending on the concrete load. AutoScaling starts additional EC2 instances if needed. ELB distributes the load between the instances. The AWS ELB service is not only suitable for web applications working with HTTP protocols, but for all types of applications. A healthcheck can also be implemented based on a TCP protocol without HTTP. This is even simpler than an HTTP healthcheck.

Still the developer team decided to implement the ELB healthchecks via HTTP for all services to achieve that they all behave exactly the same, independent of the implemented logic, the used frameworks and the language. It is well possible that in the future also applications, which do not run on JVM and for instance use Go or Python as programming languages, are deployed in AWS.

For the ELB healthcheck zanox uses the application heartbeat URL. As a result, traffic is only directed to the application resp. potentially necessary infrastructure scaling operations are only performed once the EC2 instance with the application has properly been started and the heartbeat was successfully monitored.

For application monitoring Amazon CloudWatch is a good choice as CloudWatch alarms can be used to define scaling events for the AutoScaling Policies, i.e. the infrastructure scales automatically based on metrics. For this purpose EC2 basis metrics like CPU can for instance be used. Alternatively, it is possible to send your own metrics to CloudWatch. For this purpose this project uses a fork of the project jmxtrans-agent, which uses the CloudWatch API to send JMX metrics to the monitoring system. JMX (Java Management Extension) is the standard for monitoring and metrics in the Java world. Besides metrics are sent from within the application (i.e. from within the business logic) using the library Coda Hale Metrics and a module for the CloudWatch integration by Blacklocus.

A slightly different approach is chosen for the logging: In a Cloud environment it is never possible to rule out that a server instance is abruptly terminated. This causes often the sudden loss of data, which are stored on the server. Log files are an example for that. For this reason a logstash-forwarder runs in parallel to the core application on the server for sending the log entries to our ELK-Service running in our own data center. This stack consists of Elasticsearch for storage, Logstash for parsing the log data and Kibana for UI-based analysis. ELK is an acronym for Elasticsearch, Logstash und Kibana. In addition a UUID is calculated for each request resp. each event in our HTTP layer so that log entries can still be assigned to events after EC2 instances have ceased to exist.

Conclusion

The pattern of Microservices architectures fits well to the dynamic approach of Amazon Cloud if the architecture is well designed and implemented. The clear advantage over implementing in your own data center is the infrastructure flexibility. This allows to implement a nearly endlessly scalable architecture, which is in addition very cost-efficient.

14.14 Conclusion

The technologies used in the example provide a very good foundation for implementing a Microservices architecture with Java. Essentially, the example is based on the Netflix Stack, which has demonstrated its efficacy for years already in one of the largest websites.

The example demonstrates the interplay of different technologies for Service Discovery, Load Balancing and Resilience – as well as an approach for testing Microservices und for their execution in Docker Containers. The example is not meant to be directly useable in a production context, but is first of all designed to be very easy to set up and get running. This entails a number of compromises. However, the example serves very well as foundation for further experiments and the testing of ideas.

In addition the example demonstrates a Docker-based application deployment, which is a good foundation for Microservices.

Essential Points

- Spring, Spring Boot, Spring Cloud and the Netflix Stack offer a well integrated stack for Java-based Microservices. These technologies solve all

typical challenges posed during the development of Microservices.

- Docker based deployment is easy to implement and in conjunction with Docker Machine and Docker Compose can be used for deployment in the Cloud, too.
- The example application shows how to test Microservices using Consumer-Driven Contract Tests and Stubs without special tools. However, for real life projects tools might be more useful.

Try and Experiment

Add Log Analysis

The log analysis of all log files is important for running a Microservice system. At https://github.com/ewolff/user-registration an example project is provided. In the sub directory **log-analysis** it contains a setup for an ELK (Elasticsearch, Logstash und Kibana) stack-based log analysis. Use this approach to also add a log analysis to the Microservice example.

Add Monitoring

In addition the example project from the Continuous Delivery book contains in the sub directory graphite an installation of Graphite for monitoring. Adapt this installataion for the Microservice example.

Rewrite a Service

Rewrite one of the services in a different programming language. Use the Consumer-driven Contract Tests (compare Section 14.13 and 11.7 to protect the implementation. Make use of a sidecar for the integration into the technology stack (compare Section 14.12).

15 Technologies for Nanoservices

Section 15.1 discusses the advantages of Nanoservices and why Nanoservices can be useful. Section 15.2 defines Nanoservices and distinguishes them from Microservices. Section 15.3 focuses on Amazon Lambda: a Cloud technology which can be used with Python, JavaScript or Java. Here each function call is billed instead of renting virtual machines or application servers. OSGi (section 15.4) modularizes Java applications and also provides services. Another Java technology for Nanoservices is Java EE (section 15.5), if used correctly. Vert.x, another option, (section 15.6) also runs on the JVM, but supports in addition to Java a broad variety of programming languages. Section 15.7 focuses on the programming language Erlang which is quite old. The architecture of Erlang allows the implementation of Nanoservices. Seneca (section 15.8) has a similar approach as Erlang, but is based on JavaScript and has been specially designed for the development of Nanoservices.

The term Microservice is not uniformly defined. Some people believe Microservices should be extremely small services – i.e. ten to a hundred lines of code (LoC). This book calls such services Nanoservices. The distinction between Microservices and Nanoservices is the focus of this chapter. A suitable technology is an essential prerequisite for the implementation of small services. If the technology for instance combines several services into one operating system process, the resource utilization per service can be decreased and the service rollout in production facilitated. This decreases the expenditure per service which allows to support a large number of small Nanoservices.

15.1 Why Nanoservices?

Nanoservices are well in line with the already discussed size limits of Microservices: Their size is below the maximum size, which was defined in section 4.1and depends for instance on the number of team members. In addition, a Microservice should be small enough to still be understood by a developer. With suitable technologies the technical limits for the minimal size of a Microservice, which were discussed in section 4.1, can be further reduced.

Very small modules are easier to understand and therefore easier to maintain and change. Besides smaller Microservices can more easily be replaced by new implementations or a rewrite. Accordingly, systems consisting of minimally sized Nanoservices can more easily be developed further.

There are systems which successfully employ Nanoservices. In fact, in practice it is rather the too large modules that are the source of problems and prevent the successful further development of a system. Each functionality could be implemented in its own Microservice – each class or function could become a separate Microservice. Section 10.2 demonstrated that it can be sensible for CQRS to implement a Microservice which only reads data of a certain type. Writing the same type of data can already be implemented in another Microservice. So Microservices can really have a pretty small scope.

Minimum Size of Microservices is Limited

What are reasons against very tiny Microservices? Section 4.1 identified factors which render Microservices below a certain size not practicable:

- The expenditure for infrastructure increases. When each Microservice is a separate process and requires infrastructure such as an application server and monitoring, the expenditure necessary for running hundreds or even thousands of Microservices becomes too large. Therefore, Nanoservices require technologies which allow to keep the expenditure for infrastructure per individual service as small as possible. In addition, a low resource utilization is desirable. The individual services should consume as little memory and CPU as possible.
- In case of very small services a lot of communication via the network is required. That has a negative influence on system performance. Consequently, when working with Nanoservices communication between the services should not occur via the network. This might result in less technological freedom. When all Nanoservices run in a single process, they are usually required to employ the same technology. Such an approach also affects system robustness. When several services run in the same process, it is much more difficult to isolate them from each other. A Nanoservice can use up so many resources that other Nanoservices do not operate error-free anymore. When two Nanoservices run in the same process, the operating system cannot intervene in such situations. In addition, a crash of a Nanoservice can result in the failure of additional Nanoservices. If the

processes crashes, it will affect all Nanoservices running in the same process.

The technical compromises can have a negative effect on the properties of Nanoservices. In any case the essential feature of Microservices has to be maintained – namely, the independent deployment of the individual services.

Compromises

In the end the main task is to identify technologies which minimize the overhead per Nanoservice and at the same time preserve as many advantages of Microservices as possible.

In detail the following points need to be achieved:

- The expenditure for infrastructure such as monitoring and deployment has to be kept low. It has to be possible to bring a new Nanoservice into production without much effort and to have it immediately displayed in monitoring.
- Resource utilization for instance in regards to memory should be as low as possible to allow a large number of Nanoservices also with little hardware. This does not only make the production environment cheaper, but also facilitates the generation of test environments.
- Communication should be possible without network. This does not only improve latency and performance, but increases the reliability of the communication between Nanoservices because it is not influenced by network failures.
- Concerning isolation a compromise has to be found. The Nanoservices should be isolated from each other so that one Nanoservice cannot cause another Nanoservice to fail. Otherwise a single Nanoservice might cause the entire system to break down. However, achieving a perfect isolation might be less important than having a lower expenditure for infrastructure, a low resource utilization and the other advantages of Nanoservices.
- Using Nanoservices can limit the choice of programming languages, platforms and frameworks. Microservices on the other hand allow in principle a free choice of technology.

Desktop Applications

Nanoservices enable the use of Microservice approaches in areas in which Microservices themselves are hardly useable. One example is the possibility to divide a desktop application in Nanoservices. OSGi (section 15.4) is for instance

used for desktop and even for embedded applications. A desktop application consisting of Microservices is on the other hand probably too difficult to deploy to really use it for desktop applications. Each Microservice has to be deployed by itself and that is hardly possible for a large number of desktop clients - some of which might even be located in other companies. Moreover the integration of several Microservices into a coherent desktop application is hard - in particular if they are implemented as completely separated processes.

15.2 Nanoservices: Definition

A Nanoservice differs from a Microservice: It compromises in certain areas. One of these areas is isolation: Multiple Nanoservices run on a single virtual machine or in a single process. Another area is technology freedom: Nanoservices use a shared platform or programming language. Only with these limitations does the use of Nanoservices become feasible. The infrastructure can be so efficient that a much larger number of services is possible. This allows the individual services to be smaller. A Nanoservice might comprise only a few lines of code.

However, by no means may the technology require a joint deployment of Nanoservices since independent deployment is the central characteristic of Microservices and also Nanoservices. Independent deployment constitutes the basis for the essential advantages of Microservices: Teams which can work independently, a strong modularization and as consequence a sustainable development.

Therefore, Nanoservices can be defined as follows:

- Nanoservices *compromise* in regards to some Microservice properties such as isolation and technology freedom. However, Nanoservices still have to be independently deployable.
- The compromises allow for a *larger number* of services and therefore for *smaller services*. Nanoservices can contain just a few lines of code.
- To achieve this, Nanoservices use *highly efficient runtime environments*. These exploit the restrictions of Nanoservices in order to allow for more and smaller services.

Thus Nanoservices depend a lot on the employed technologies. The technology enables certain compromises in Nanoservices and therefore Nanoservices of a certain size. Therefore, this chapter is geared to different technologies to explain the possible varieties of Nanoservices.

The objective of Nanoservices is to amplify a number of advantages of Microservices. Having even smaller deployment units decreases the deployment risk further, facilitates deployment even more and achieves better understandable and replaceable services. In addition, the domain architecture will change: A *Bounded Context* which might consist of one or a few Microservices will now comprise a multitude of Nanoservices which each implement a very narrowly defined functionality.

The difference between Microservices and Nanoservices is not strictly defined: If two Microservices are deployed in the same virtual machine, efficiency increases and isolation is compromised. The two Microservices now share an operating system instance and a virtual machine. When one of the Microservices uses up the resources of the virtual machine, the other Microservice running on the same virtual machine will also fail. This is the compromise in terms of isolation. So in a sense these Microservices are already Nanoservices.

By the way, the term “Nanoservice” is not used very much. This book uses the term “Nanoservice” to make it plain that there are modularizations which are similar to Microservices, but differ when it comes to detail thereby allowing for even smaller services. To distinguish these technologies with their compromises clearly from “real” Microservices the term “Nanoservice” is useful.

15.3 Amazon Lambda

Amazon Lambda is a service in the Amazon Cloud. It is available worldwide in all Amazon computing centers.

Amazon Lambda can execute individual functions which are written in Python, JavaScript with Node.js or Java 8 with OpenJDK. The code of these functions does not have dependencies on Amazon Lambda. Access to the operating system is possible. The computers the code is executed on contain the Amazon Web Services SDK as well as ImageMagick for image manipulations. These functionalities can be used by Amazon Lambda applications. Besides additional libraries can be installed.

Amazon Lambda functions have to start quickly because it can happen that they are started for each request. Therefore, the functions may also not hold a state.

Thus there are no costs when there are no requests that cause an execution of the functions. Each request is billed individually. Currently the first million requests

is for free and a further million costs 0,20 $.

Calling Lambda Functions

Lambda functions can be called directly via a command line tool. The processing occurs asynchronously. The functions can return results via different Amazon functionalities. For this purpose, the Amazon Cloud contains messaging solutions such as SNS (Simple Notification Service) or SQS (Simple Queuing Service).

The following events can trigger a call of a Lambda function:

- In S3 (Simple Storage Service) large files can be stored and downloaded. Such actions trigger events to which an Amazon Lambda function can react.
- Amazon Kinesis can be used to administrate and distribute data streams. This technology is meant for the real time processing of large data amounts. Lambda can be called as reaction to new data in these streams.
- With Amazon Cognito it is possible to use Amazon Lambda to provide simple backends for mobile applications.
- The API Gateway provides a way to implement REST APIs using Amazon Lambda.
- Furthermore it is possible to have Amazon Lambda functions be called at regular intervals.
- As a reaction to a notification in SNS (Simple Notification Service) an Amazon Lambda function can be executed. As there are many services which can provide such notifications, this makes Amazon Lambda useable in many scenarios.
- DynamoDB is a database within the Amazon Cloud. In case of changes to the database it can call Lambda functions. So Lambda functions essentially become database triggers.

Evaluation for Nanoservices

Amazon Lambda allows the independent deployment of different functions without problems. They can also bring their own libraries along.

The technological expenditure for infrastructure is minimal when using this technology: A new version of an Amazon Lambda function can easily be deployed with a command line tool. Monitoring is also simple: The functions are immediately integrated into Cloud Watch. Cloud Watch is offered by Amazon to create metrics of Cloud applications and to consolidate and monitor log files. In addition, alarms can be defined based on these data which can be forwarded by

SMS or email. Since all Amazon services can be contacted via an API, monitoring or deployment can be automated and integrated into their own infrastructures.

Amazon Lambda provides integration with the different Amazon services e.g. S3, Kinesis and DynamoDB. It is also easily possible to contact an Amazon Lambda function via REST using the API Gateway. However, Amazon Lambda exacts that Node.js, Python or Java are used. This profoundly limits the technology freedom.

Amazon Lambda offers an excellent isolation of functions. This is also necessary since the platform is used by many different users. It would not be acceptable for a Lambda function of one user to negatively influence the Lambda functions of other users.

Conclusion

Amazon Lambda allows to implement extremely small services. The overhead for the individual services is very small. Independent deployment is easily possible. A Python, JavaScript or Java function are the smallest deployment units supported by Amazon Lambda – it is hardly possible to make them any smaller. Even if there is a multitude of Python, Java or JavaScript functions, the expenditure for the deployments remains relatively low.

Amazon Lambda is a part of the Amazon ecosystem. Therefore, it can be supplemented by technologies like Amazon Elastic Beanstalk. There Microservices can run which can be larger and written in other languages. In addition, a combination with EC2 (Elastic Computing Cloud) is possible. EC2 offers virtual machines on which any software can be installed. Moreover, there is a broad choice in regards to databases and other services which can be used with little additional effort. Amazon Lambda defines itself as a supplement of this tool kit. In the end one of the crucial advantages of the Amazon Cloud is that nearly every possible infrastructure is available and can easily be used. Thus developers can concentrate on the development of specific functionalities while most standard components can just be rented.

Try and Experiment

There is a comprehensive tutorial which illustrates how to use Amazon Lambda. It does not only demonstrate simple scenarios, but also shows how to use complex mechanisms such as different Node.js libraries, implementing REST services or how to react to different events in the Amazon system. Amazon offers cost free quotas of most services to new customers. In case of Lambda each customer gets such a large free quota that it is fully sufficient for tests and a first getting to know the technology. Also note that the first million calls during a month are free. However, you should check the current pricing.

15.4 OSGi

OSGi is a standard with many different implementations. Embedded systems often use OSGi. Also the development environment Eclipse is based on OSGi, and many Java desktop applications use the Eclipse framework. OSGi defines a modularization within the JVM (Java Virtual Machine). Even though Java allows for a division of code into classes or packages, there is no modular concept for larger units.

The OSGi Module System

OSGi supplements Java by such a module system. To do so OSGi introduces bundles into the Java world. Bundles are based on Java's JAR files which comprise code of multiple classes. Bundles have a number of additional entries in the file **META-INF/MANIFEST.MF**, which each JAR file should contain. These entries define which classes and interfaces the bundle exports. Other bundles can import these classes and interfaces. Thereby OSGi extends Java with a quite sophisticated module concept without inventing entirely new concepts.

Listing 12: OSGi MANIFEST.MF

```
1 Bundle-Name: A service
2 Bundle-SymbolicName: com.ewolff.service
3 Bundle-Description: A small service
4 Bundle-ManifestVersion: 2
5 Bundle-Version: 1.0.0
6 Bundle-Acltivator: com.ewolff.service.Activator
7 Export-Package: com.ewolff.service.interfaces;version="1.0.0"
8 Import-Package: com.ewolff.otherservice.interfaces;version="1.3.0"
```

Listing 12 shows an example of a **MANIFEST.MF** file. It contains the description and name of the bundle and the bundle activator. This Java class is executed upon the start of the bundle and can initialize the bundle. **Export-Package** indicates

which Java packages are provided by this bundle. All classes and interfaces of these packages are available to other bundles. **Import-Package** serves to import packages from another bundle. The packages can also be versioned.

In addition to interfaces and classes bundles can also export services. However, an entry in **MANIFEST.MF** is not sufficient for this. Code has to be written. Services are in the end only Java objects. Other bundles can import and use the services. Also calling the services happens in the code.

Bundles can be installed, started, stopped and uninstalled at runtime. So bundles are easy to update: Stop and uninstall the old version, then install a new version and start. However, if a bundle exports classes or interfaces and another bundle uses these, an update is not so simple anymore. All bundles which use classes or interfaces of the old bundle and now want to use the newly installed bundle have to be restarted.

Handling Bundles in Practice

Sharing code is by far not as important for Microservices as the use of services. Nevertheless at least the interface of the services has to be offered to other bundles.

In practice a procedure has been established where a bundle only exports the interface code of the service as classes and Java interfaces. Another bundle contains the implementation of the service. The classes of the implementation are not exported. The service implementation is exported as OSGi service. To use the service a bundle has to import the interface code from the one bundle and the service from the other bundle (compare Fig. 79).

OSGi allows to restart services. With the described approach the implementation of the service can be exchanged without having to restart other bundles. These bundles only import the Java interfaces and classes of the interface code. That code does not change for a new service implementation so that restarting is not necessary anymore. That way the access to services can be implemented in such a manner that the new version of the service is in fact used.

With the aid of OSGi blueprints or OSGi declarative services these details can be abstracted away when dealing with the OSGi service model. This facilitates the handling of OSGi. These technologies for instance render it much easier to handle the restart of a service or its temporary failure during the restart of a bundle.

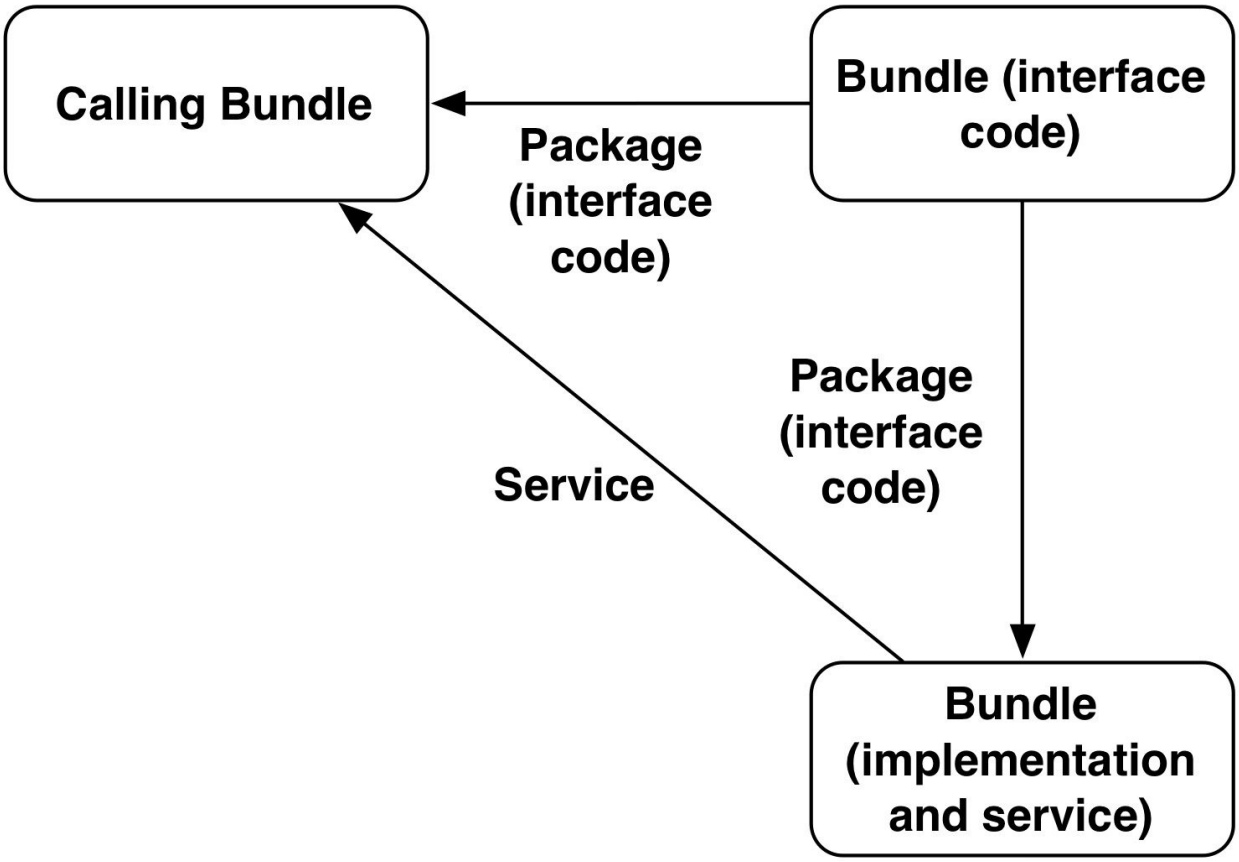


Fig. 79: OSGi service, implementation and interface code

An independent deployment of services is possible, but also laborious since interface code and service implementation have to be contained in different bundles. This model allows only changes to the implementation. Modifications of the interface code are more complex. In such a case the bundles using a service have to be restarted because they have to reload the interface.

In reality OSGi systems are often completely reinstalled for these reasons instead of modifying individual bundles. An Eclipse update for instance often entails a restart. A complete reinstallation facilitates also the reproduction of the environment. When an OSGi system is dynamically changed, at some point it will be in a state which nobody is able to reproduce. However, modifying individual bundles is an essential prerequisite for implementing the Nanoservice approach with OSGi. Independent deployment is an essential property of a Nanoservice. So OSGi compromises this essential property.

Evaluation for Nanoservices

OSGi has a positive effect on Java projects in regards to architecture. The bundles are usually relatively small so that the individual bundles are easy to understand.

In addition, the split into bundles forces the developers and architects to think about the relationships between the bundles and to define them in the configurations of the bundles. Other dependencies between bundles are not possible within the system. Normally this leads to a very clean architecture with clear and intended dependencies.

However, OSGi does not offer technological freedom: It is based on the JVM and therefore can only be used with Java or JVM-based languages. For example, it is nearly impossible that an OSGi bundle brings along its own database because databases are normally not written in Java. For such cases additional solutions alongside the OSGi infrastructure have to be found.

For some Java technologies an integration with OSGi is difficult since loading Java classes works differently without OSGi. Moreover, many popular Java application servers do not support OSGi for deployed applications so that changing code at runtime is not supported in such environments. The infrastructure has to be specially adapted for OSGi.

Furthermore, the bundles are not fully isolated: When a bundle uses a lot of CPU or causes the JVM to crash, the other bundles in the same JVM will be affected. Failures can occur for instance due to memory leak which causes more and more memory to be allocated due to an error until the system breaks down. Such errors easily arise due to blunders.

On the other hand, the bundles can locally communicate due to OSGi. Distributed communication is also possible with different protocols. Moreover, the bundles share a JVM which reduces for instance the memory utilization.

Solutions for monitoring are likewise present in the different OSGi implementations.

Conclusion

OSGi leads first of all to restrictions in regards to technological freedom. It restricts the project to Java technologies. In practice the independent deployment of the bundles is hard to implement. Especially interface changes are poorly supported. Besides bundles are not well isolated from each other. On the other hand, bundles can easily interact via local calls.

Try and experiment

- Get familiar with OSGi for instance with the aid of a tutorial.

- Create a concept for the distribution into bundles and services for a part of a system you know.
- If you had to implement the system with OSGi: Which additional technologies (databases etc.) would you have to use? How would you handle this?

15.5 Java EE

Java EE is a standard from the Java field. It comprises different APIs such as for instance JSF (Java ServerFaces), Servlet and JSP (Java Server Pages) for web applications, JPA (Java Persistence API) for persistence or JTA for transactions. Besides Java EE defines a deployment model. Web applications can be packaged into WAR files (Web ARchive), JAR files (Java ARchive) can contain logic components like Enterprise Java Beans (EJBs), and EARs (Enterprise ARchives) can comprise a collection of JARs and WARs. All these components are deployed in one application server. The application server implements the Java EE APIs and offers for instance support for HTTP, threads and network connections and also support for accessing databases.

This section deals with WARs and the deployment model of Java EE application servers. Chapter 14 already described in detail a Java system that does not require an application server. Instead it directly starts a Java application on the Java Virtual Machine (JVM). The application is packaged in a JAR file and contains the entire infrastructure. This deployment is called Fat JAR deployment, because the application including the entire infrastructure is contained in one single JAR. The example from chapter 14 uses Spring Boot which also supports a number of Java EE APIs such as JAX-RS for REST. Dropwizard also offers such a JAR model. It is actually focused on JAX RS-based REST web services, however, it can also support other applications. Wildfly Swarm is a variant of the Java EE server Wildfly which also supports such a deployment model.

Nanoservices with Java EE

A Fat JAR deployment utilizes too many resources for Nanoservices. In a Java EE application server multiple WARs can be deployed thereby saving resources. Each WAR can be accessed via its own URL. Furthermore, each WAR can be individually deployed. This allows to bring each Nanoservice individually into production.

However, the separation between WARs is not optimal:

- Memory and CPU are collectively used by all Nanoservices. When a Nanoservice uses a lot of CPU or memory, this can interfere with other Nanoservices. A crash of one Nanoservice propagates to all other Nanoservices.
- In practice, redeployment of a WAR causes memory leaks if it is not possible to remove the entire application from memory. Therefore, in practice the independent deployment of individual Nanoservices is hard to achieve.
- In contrast to OSGi the ClassLoaders of the WARs are completely separate. There is no possibility for accessing the code of other Nanoservices.
- Because of the separation of the code WARs can only communicate via HTTP or REST. Local method calls are not possible.

Since multiple Nanoservices share an application server and a JVM, this solution is more efficient than the Fat JAR Deployment of individual Microservices in their own JVM as described in chapter 14. The Nanoservices use a shared heap and thereby use less memory. However, scaling works only by starting more application servers. Each of the application servers contains all Nanoservices. All Nanoservices have to be scaled collectively. It is not possible to scale individual Nanoservices.

The technology choice is restricted to JVM technologies. Besides all technologies are excluded which do not work with the servlet model such as Vert.x (section 15.6) or Play.

Microservices with Java EE?

For Microservices Java EE can also be an option: Theoretically it would be possible to run each Microservice in its own application server. In this case an application server has to be installed and configured in addition to the application. The version of the application server and its configuration have to fit to the version of the application. For Fat JAR deployment there is no need for a specific configuration of the application server because it is part of the Fat JAR

and therefore configured just like the application. This additional complexity of the application server is not counterbalanced by any advantage. Since deployment and monitoring of the application server only work for Java applications, these features can only be used in a Microservices-based architecture when the technology choice is restricted to Java technologies. In general, application servers have hardly any advantages – especially for Microservices.

An example

The application from chapter 14 is also available with the Java EE deployment model. Fig. 80 provides an overview of the example: There are three WARs, which comprise order, customer and catalog. They communicate with each other via REST. When customer fails, order would also fail in the host since order communicates only with this single customer instance. To achieve better availability, the access would have to be rerouted to other customer instances.

A customer can use the UI of the Nanoservices from the outside via HTML/HTTP. The code contains only small modifications compared to the solution from chapter 14. The Netflix libraries have been removed. On the other hand, the application has been extended with support for servlet containers.

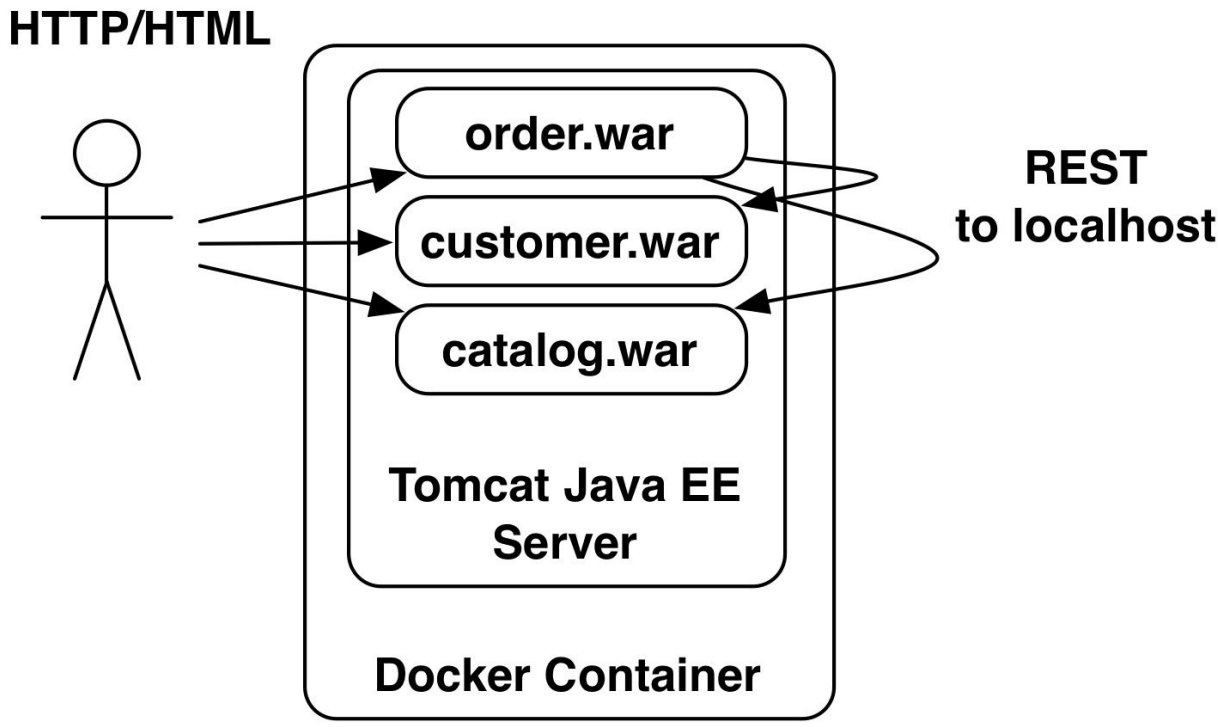


Fig. 80: Example application with Java EE Nanoservices

Try and Experiment

The application as Java EE Nanoservices can be found on GitHub.

The application does not use the Netflix technologies.

Hystrix offers Resilience (compare section 14.10).

- Does it make sense to integrate Hystrix into the application?
- How are the Nanoservices isolated from each other?
- Is Hystrix always helpful?
- Compare also section 10.5 concerning stability and resilience. How can these patterns be implemented in this application?

- Eureka is helpful for service discovery. How would it fit into the Java EE Nanoservices?
- How can other service discovery technologies be integrated (compare section 8.9)?

- Ribbon for load balancing between REST services could likewise be integrated. Which advantages would that have? Would it also be possible to use Ribbon without Eureka?

15.6 Vert.x

Vert.x is a framework containing numerous interesting approaches. Although it runs on the (Java Virtual Machine), it supports many different programming languages – such as Java, Scala, Clojure, Groovy, Ceylon as well as JavaScript, Ruby or Python. A Vert.x system is built from Verticles. They receive events and can return messages.

Listing 13 shows a simple Vert.x Verticle, which only returns the incoming messages. The code creates a server. When a client connects to the server, a callback is called, and the server creates a pump. The pump serves to transfer data from a source to a target. In the example source and target are identical.

The application becomes only active when a client connects and the callback is called. Likewise, the pump becomes only active when new data are available

from the client. Such events are processed by the event loop which calls the Verticles. The Verticles then have to process the events. An event loop is a thread. Usually one event loop is started per CPU core so that the event loops are processed in parallel. An event loop and thus a thread resp. a CPU core can support an arbitrary number of network connections. Events of all connections can be processed in a single event loop. Therefore, Vert.x is also suitable for applications which have to handle a large number of network connections.

Listing 13: Simple Java Vert.x Echo Verticle

```
1 public class EchoServer extends Verticle {
2
3   public void start() {
4     vertx.createNetServer().connectHandler(new Handler<NetSocket>() {
5       public void handle(final NetSocket socket) {
6         Pump.createPump(socket, socket).start();
7       }
8     }).listen(1234);
9   }
10 }
```

As described Vert.x supports different programming languages. Listing 14 shows the same Echo Verticle in JavaScript. The code adheres to JavaScript conventions and uses for instance a JavaScript function for callback. Vert.x has a layer for each programming language that adapts the basic functionality in such a way that it seems like a native library for the respective programming language.

Listing 14: Simple JavaScript Vert.x Echo Verticle

```
1 var vertx = require('vertx')
2
3 vertx.createNetServer().connectHandler(function(sock) {
4   new vertx.Pump(sock, sock).start();
5 }).listen(1234);
```

Vert.x modules can contain multiple Verticles in different languages. Verticles and modules can communicate with each other via an event bus. The messages on the event bus use JSON as data format. The event bus can be distributed onto multiple servers. In this manner Vert.x supports distribution and can implement high availability by starting modules on other servers. Besides the Verticles and modules are loosely coupled since they only exchange messages. Vert.x also offers support for other messaging systems and can also communicate with HTTP and REST. So it is relatively easy to integrate Vert.x systems into Microservice-based systems.

Modules can be individually deployed and also removed again. Since the modules communicate with each other via events, modules can be easily replaced by new modules at runtime. They only have to process the same messages. A module can implement a Nanoservice. Modules can be started in new nodes so that the failure of a JVM can be compensated.

Vert.x supports also Fat JARs where the application brings all necessary libraries along. This is useful for Microservices since this means that the application brings all dependencies along and is easier to deploy. For Nanoservices this approach is not so useful because the approach consumes too many resource - deploying multiple Vert.x modules in one JVM is a better option for Nanoservices.

Conclusion

Via the independent module deployment and the loose coupling by the event bus Vert.x supports multiple Nanoservices within a JVM. However, a crash of the JVM, a memory leak or blocking the event loop would affect all modules and Verticles in the JVM. On the other hand, Vert.x supports many different programming languages – in spite of the restriction to JVM. This is not only a theoretical option. In fact Vert.x aims at being easily useable in all supported languages. Vert.x presumes that the entire application is written in a non blocking manner. However, there is the possibility to execute blocking tasks in Worker Verticles. They use separate thread pools so that they do not influence the non blocking Verticles. So even code that does not support the Vert.x non blocking approach can still be used in a Vert.x system. This allows for even greater technological freedom.

Try and Experiment

The Vert.x homepage offers an easy start to developing with Vert.x. It demonstrates how a web server can be implemented and executed with different programming languages. The modules in the example use Java and Maven. There are also complex examples in other programming languages.

15.7 Erlang

Erlang is a functional programming language which is first of all used in combination with the OTP (Open Telecom Platform) framework. Originally, Erlang has been developed for telecommunication. In this field applications have to be very reliable. Meanwhile Erlang is employed in all areas which profit from

its strengths. Erlang uses a virtual machine similar to Java as runtime environment which is called BEAM (Bogdan/ Björn's Erlang Abstract Machine).

Erlang's strengths are first of all its resilience against failures and the possibility to let systems run for years. This is only possible via dynamic software updates. At the same time Erlang has a light-weight concept for parallelism. Erlang uses the concept of processes for parallel computing. These processes are not related to operating system processes and are even more light-weight than operating system threads. In an Erlang system millions of processes can run which are all isolated from each other.

Another factor contributing to the isolation is the asynchronous communication. Processes in an Erlang system communicate with each other via messages. Messages are sent to the mailbox of a process (see Fig. 81). In one process only one message is processed at a time. This facilitates the handling of parallelism: There is parallel execution because many messages can be handled at the same time. But each process takes care of only one message at a time. Parallelism is achieved because there are multiple processes. The functional approach of the language, which attempts to get by without a state, fits well to this model. This approach corresponds to the Verticles in Vert.x and their communication via the event bus.

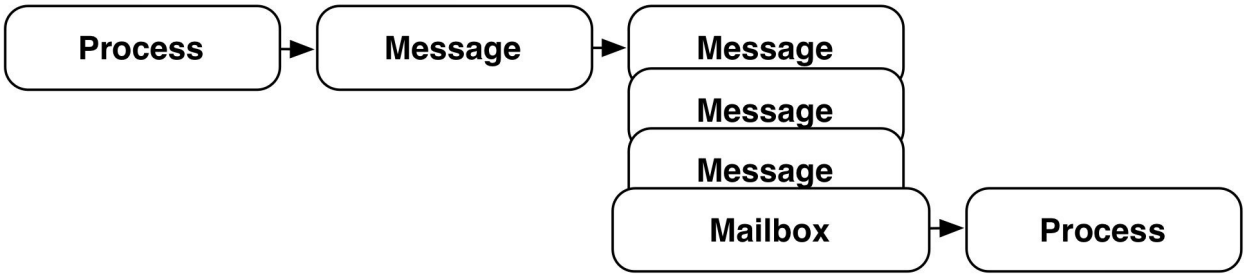


Fig. 81: Communication between Erlang processes

Listing 15 shows a simple Erlang server which returns the received message. It is defined in its own module. The module exports the function **loop**, which does not have any parameters. The function receives a message **Msg** from a node **From** and then returns the same message to this node. The operator "!" serves for sending the message. Afterwards the function is called again and waits for the next message. Exactly the same code can also be used for being called by another computer via the network. Local messages and messages via the network are processed by the same mechanisms.

Listing 15: An Erlang echo server

```
1 -module(server).
2 -export([loop/0]).
3 loop() ->
4      receive
5        {From, Msg} ->
6          From ! Msg,
7          loop()
8 end.
```

Due to the sending of messages Erlang systems are especially robust. Erlang makes use of “Let It Crash”. An individual process is just restarted when problems occur. This is the responsibility of the supervisor: A process which is specifically dedicated to monitoring other processes and restarting them if necessary. The supervisor itself is also monitored and restarted in case of problems. Thereby a tree is created in Erlang which in the end prepares the system for the case that processes should fail (see Fig. 82).

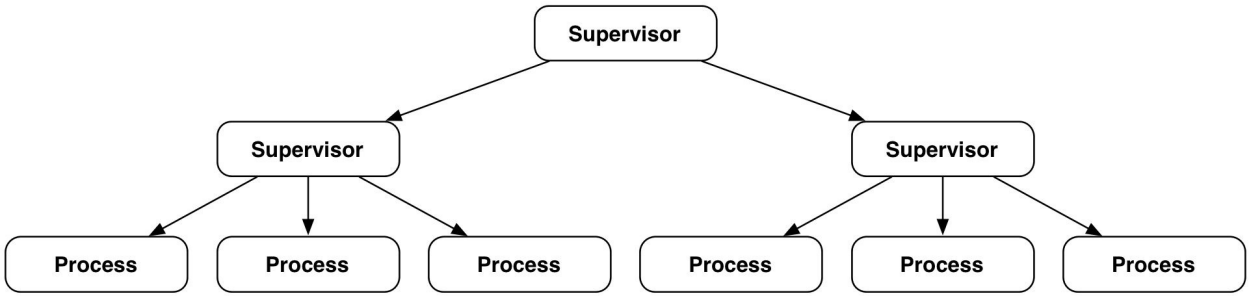


Fig. 82: Monitoring in Erlang systems

Since the Erlang process model is so light-weight, restarting a process is rapidly done. When the state is stored in other components, there will also be no information loss. The remainder of the system is not affected by the failure of the process: As the communication is asynchronous, the other processes can handle the higher latency caused by the restart. In practice this approach has proven very reliable. Erlang systems are very robust and still easy to develop.

This approach is based on the actor model: Actors communicate with each other via asynchronous messages. As a response they can themselves send messages, start new actors or change their behavior for the next messages. Erlang’s processes correspond to actors.

In addition, there are easy possibilities to monitor Erlang systems. Erlang itself has built-in functions which can monitor memory utilization or the state of the

mailboxes. OTP offers for this purpose the Operations and Maintenance Support (OAM), which can for instance also be integrated into SNMP systems.

Since Erlang solves typical problems arising upon the implementation of Microservices like resilience, it supports the implementation of Microservices quite well. In that case a Microservice is a system written in Erlang which internally consists of multiple processes.

However, the services can also get smaller: Each process in an Erlang system could be considered as Nanoservice. It can be deployed independently of the others, even during runtime. Furthermore, Erlang supports operating system processes. In that case they are also integrated into the supervisor hierarchy and restarted in case of a break down. This means that any operating system process written in any language might become a part of an Erlang system and its architecture.

Evaluation for Nanoservices

As discussed an individual process in Erlang can be viewed as Nanoservice. The expenditure for the infrastructure is relatively small in that case: Monitoring is possible with built-in Erlang functions. The same is true for deployment. Since the processes share a BEAM instance, the overhead for a single process is not very high. In addition, it is possible for the processes to exchange messages without having to communicate via the network and therefore with little overhead. The isolation of processes is also implemented.

Finally, even processes in other languages can be added to an Erlang system. For this purpose an operating system process which can be implemented in an arbitrary language is put under the control of Erlang. The operating system process can for instance be safeguarded by “Let It Crash”. This allows to integrate practically all technologies into Erlang – even if they run in a separate process.

On the other hand, Erlang is not very common. The consequent functional approach also needs getting used to. Finally, the Erlang syntax is not very intuitive for many developers.

Try and Experiment

A very simple example is based on the code from this section and demonstrates how communication between nodes is possible. You can use it to get a basic understanding of Erlang.

There is a very nice tutorial for Erlang, which also treats deployment and operation. With the aid of the information from the tutorial the example can be supplemented by a supervisor.

An alternative language out of the Erlang ecosystem is Elixir. Elixir has a different syntax, but also profits from the concepts of OTP. Elixir is much simpler to learn than Erlang and thus lends itself to a first start.

There are many other implementations of the actor model. It is worthwhile to look more closely before deciding whether such technologies are also useful for the implementation of Microservices or Nanoservices and which advantages might be associated. Akka from the Scala / Java area might be of interest here.

15.8 Seneca

Seneca is based on Node.js and accordingly uses JavaScript on the server. Node.js has a programming model where one operating system process can take care of many tasks in parallel. To achieve this there is an event loop which handles the events. When a message enters the system via a network connection, the system will first wait until the event loop is free. Then the event loop processes the message. The processing has to be fast since the loop is blocked otherwise resulting in long waiting times for all other messages. For this reason, the response of other servers may in no case be waited for in the event loop. That would block the system for too long. The interaction with other systems has to be implemented in such a way that the interaction is only initiated. Then the event loop is freed to handle other events. Only when the response of the other system arrives, it is processed by the event loop. Then the event loop calls a callback which has been registered upon the initiation of the interaction. This model is similar to the approaches used by Vert.x and Erlang.

Seneca introduces a mechanism in Node.js which allows to process commands. Patterns of commands are defined which cause certain code to be executed.

Communicating via such commands is also easy to do via the network. Listing 16 shows a server which calls **seneca.add()**. Thereby a new pattern and code for handling events with this pattern are defined. To the command with the component **cmd: "echo"** a function reacts. It reads out the **value** from the command and puts it into the **value** parameter of the function **callback**. Then the function **callback** is called. With **seneca.listen()** the server is started and listens to commands from the network.

Listing 16: Seneca Server

```
1 var seneca = require("seneca")()
2
3 seneca.add( {cmd: "echo"}, function(args,callback){
4      callback(null,{value:args.value})
5 })
6
7 seneca.listen()
```

The client in Listing 17 sends all commands which cannot be processed locally via the network to the server. **seneca.client(). seneca.act()** creates the commands that are sent to the server. It contains **cmd: "echo"** – therefore the function of the server in Listing 16 is called. **"echo this"** is used as value. The server returns this string to the function which was passed in as a callback – and in this way it is finally printed on the console. The example code can be found on GitHub.

Listing 16: Seneca Client

```
1 var seneca=require("seneca")()
2
3 seneca.client()
4
5 seneca.act('cmd: "echo",value:"echo this", function(err,result){
6      console.log( result.value )
7 })
```

Therefore, it is very easy to implement a distributed system with Seneca. However, the services do not use a standard protocol like REST for communicating. Nevertheless, also REST systems can be implemented with Seneca. Besides the Seneca protocol is based on JSON and therefore can also be used by other languages.

A Nanoservice can be a function which reacts with Seneca to calls from the network – and therefore it can be very small. As already described, a Node.js system as implemented with Seneca is fragile when a function blocks the event loop. Therefore, the isolation is not very good.

For the monitoring of a Seneca application there is an admin console which at least offers a simple monitoring. However, it is in each case only available for one Node.js process. Monitoring across all servers has to be achieved by different means.

An independent deployment of a single Seneca function is only possible if there is a single Node.js process for the Seneca function. This represents a profound limitation for independent deployment since the expenditure of a Node.js process is hardly acceptable for a single JavaScript function. In addition, it is not easy to integrate other technologies into a Seneca system. In the end the entire Seneca system has to be implemented in JavaScript.

Evaluation for Nanoservices

Seneca has been especially developed for the implementation of Microservices with JavaScript. In fact, it enables a very simple implementation for services which can also be contacted via the network. The basic architecture is similar to Erlang: In both approaches services send messages resp. commands to each other to which functions react. In regards to the independent deployment of individual services, the isolation of services from each other and the integration of other technologies Erlang is clearly superior. Besides Erlang has a much longer history and has long been employed in different very demanding applications.

Try and Experiment

The code example can be a first step to get familiar with Seneca. You can also use the basic tutorial. In addition, it is worthwhile to look at other examples. The Nanoservice example can be enlarged to a comprehensive application or can be distributed to a larger number of Node.js processes.

15.9 Conclusion

The technologies presented in this chapter show how Microservices can also be implemented very differently. Since the difference is so large, the use of the separate term “Nanoservice” appears justified. Nanoservices are not necessarily

independent processes anymore which can only be contacted via the network, but might run together in one process and use local communication mechanisms to contact each other. Thereby not only the use of extremely small services is possible, but also the adoption of Microservice approaches in areas such as embedded or desktop applications.

An overview of the advantages and disadvantages of different technologies in regards to Nanoservices is provided in Tab. 3. Erlang is the most interesting technology since it also allows the integration of other technologies and is able to isolate the individual Nanoservices quite well from each other so that a problem in one Nanoservice will not trigger the failure of the other services. In addition, Erlang has been the basis of many important systems for a long time already so that the technology as such has proven its reliability beyond doubt.

Seneca follows a similar approach, but cannot compete with other technologies in terms of isolation and the integration of other technologies than JavaScript. Vert.x has a similar approach on the JVM and supports numerous languages. However, it does not isolate Nanoservices as well as Erlang. Java EE does not allow for communication without network, and individual deployment is difficult in Java EE. In practice memory leaks occur frequently during the deployment of WARs. So during a deployment the application server is usually restarted to avoid memory leaks. Then all Nanoservices are unavailable for some time. Therefore a Nanoservice cannot be deployed without influencing the other Nanoservices. OSGi allows in contrast to Java EE the shared use of code between Nanoservices. In addition, OSGi uses methods calls for communication between services and not commands resp. messages like Erlang and Seneca. Commands or messages have the advantage of being more flexible. Parts of a message which a certain service does not understand are not a problem- they can just be ignored.

Tab.3: Technology evaluation for Nanoservices

	Lambda	**OSGi**	**Java EE**	**Vert.x**	**Erlang**	**Seneca**
Effort for infrastructure per service	++	+	+	+	++	++
Resource consumption	++	++	++	++	++	++
Communication with network	-	++	- -	+	++	-
Isolation of services	++	- -	- -	-	++	-
Use of different	-	- -	- -	+	+	- -

technologies

Amazon Lambda is especially interesting since it is integrated into the Amazon ecosystem. This makes handling the infrastructure very easy. The infrastructure can be a challenging problem in case of small Nanoservices because so many more environments are needed due to the high number of services. With Amazon a database server is only an API call or a click away – alternatively, an API can be used to store data instead of a server. Servers become invisible for storing data – and this is also the case with Amazon Lambda for executing code. There is no infrastructure for an individual service, but only code which is executed and can be used by other services. Because of the prepared infrastructure monitoring is also no challenge anymore.

Essential Points

- Nanoservices divide systems into even smaller services. To achieve this, they compromise in certain areas such as technology freedom or isolation.
- Nanoservices require efficient infrastructures which can handle a large number of small Nanoservices.

16 How to Start with Microservices

As conclusion of the book this chapter shows what the start with Microservices can look like. Section 16.1 enumerates the different advantages of Microservices once more to illustrate that there is not only a single reason to introduce Microservices, but several. Section 16.2 describes several ways for introducing Microservices – depending on the use context and the expected advantages. Section 16.3 finally follows up on the question whether Microservices are more than just a hype.

16.1 Why Microservices?

Microservices entail a number of advantages such as (compare also chapter 5):

- Microservices make it easier to implement agility for large projects since teams can work independently.
- Microservices can help to supplement and replace legacy applications stepwise.
- Microservice-based architectures allow for sustainable development since they are less susceptible to architecture decay and because individual Microservices can be easily replaced. This increases the long-term maintainability of the system.
- In addition, there are technical reasons for Microservices such as robustness and scalability.

To prioritize these advantages and the additional ones mentioned in chapter 5 should be the first step when considering the adaptation of a Microservice-based architecture. Likewise the challenges discussed in chapter 6 have to be evaluated and, where necessary, strategies for dealing with these challenges have to be devised.

Continuous Delivery and infrastructure play a prominent role in this context. If the deployment processes are still manual, the expenditure for operating a large number of Microservices is so high that their introduction is hardly feasible. Unfortunately, many organizations still have profound weaknesses especially in the area of Continuous Delivery and infrastructure. In such a case Continuous

Delivery should be introduced alongside Microservices. Since Microservices are much smaller than Deployment Monoliths, Continuous Delivery is also easier with Microservices. Therefore, both approaches have synergies.

In addition the organizational level (chapter 13) has to be taken into account. When the scalability of agile processes constitutes an important reason for introducing Microservices, the agile processes should already be well established. For example, there has to be a Product Owner per team, who also decides about all features, as well as agile planning. The teams should also be already largely self-reliant – otherwise in the end they might not make use of the independence Microservices offer.

Introducing Microservices can solve more than just one problem. The specific motivation for Microservices will differ between projects. The large number of advantages can on its own be a good reason for introducing Microservices. In the end the strategy for introducing Microservices has to be adapted to the advantages that are most important in the context of a specific project.

16.2 Roads towards Microservices

There are different approaches which pave the way towards Microservices:

- The most typical scenario is to start out with a monolith which is converted stepwise into a multitude of Microservices. Usually, different functionalities are transferred one by one into Microservices. A driving force behind this conversion is often the wish for an easier deployment. However, independent scaling and achieving a more sustainable architecture can also be important reasons.
- However, migrating from a monolith to Microservices can also occur in a different manner. When for instance resilience is the main reason for switching to Microservices, the migration can be started by first adding technologies like Hystrix to the monolith. Afterwards the system can be split into Microservices.
- Starting a Microservice-based system from scratch is by far the rarer scenario. Even in such a case a project can start by building a monolith. However, it is more sensible to devise a first coarse-grained domain architecture which leads to the first Microservices. Thereby an infrastructure is created which supports more than just one Microservice. This approach also allows teams to already work independently on features. However, a

fine-granular division into Microservices right from the start often does not make sense because it will probably have to be revised again later on. Introducing the necessary profound changes into an already existing Microservices architecture can be highly complex.

Microservices are easy to combine with existing systems which facilitates their introduction. A small Microservice as supplement to an existing Deployment Monolith is rapidly written. If problems arise, such a Microservice can also be rapidly removed again from the system. Other technical elements can then be introduced in a stepwise manner.

The easy combination of Microservices with legacy systems is an essential reason for the fact that the introduction of Microservices is quite simple and can immediately result in advantages.

16.3 Microservice: Hype or Reality?

Without doubt Microservices are an approach which is in the focus of attention right now. This does not have to be bad – yet, such approaches often are at second glance only a fashion and do not solve any real problems.

However, the interest in Microservices is more than just a fashion or hype:

- As described in the introduction, Amazon has been employing Microservices for many years. Likewise, many internet companies have been following this approach for a long time. Therefore, Microservices are not just a new fashion, but have already been used for a long time behind the scenes in many companies before they became fashionable.
- For the Microservice pioneers the advantages associated with Microservices were so profound that they were willing to invest a lot of money into creating the not yet existing necessary infrastructures. These infrastructures are nowadays available free of cost as Open Source – Netflix is a prominent example. Therefore, it is much easier nowadays to introduce Microservices.
- The trend towards agility and Cloud infrastructures is suitably complemented by Microservices-based architectures: They enable the scaling of agility and fulfill the demands of the Cloud in regards to robustness and scalability.
- Likewise Microservices as small deployment units support Continuous Delivery which is employed by many enterprises to increase software quality and to bring software more rapidly into production.

- There is more than one reason for Microservices. Therefore, Microservices represent an improvement for many areas. Since there is not a single reason for the introduction of Microservices, but a number of them, it is more likely that even very diverse projects will in the end really benefit from switching to Microservices.

Presumably, everybody has already seen large, complex systems. Maybe it is now the time to develop smaller systems and to profit from the associated advantages. In any case there seem to be only very few reasons arguing for monoliths – except for their lower technical complexity.

16.4 Conclusion

Introducing Microservices makes sense for a number of reasons:

- There is a plethora of advantages (discussed in section 16.1 and chapter 5).
- The way to Microservices is evolutionary. It is not necessary to start adopting Microservices for the whole system from the beginning. Quite in contrast: A stepwise migration is the usual way (section 16.2). Many different approaches can be chosen in order to profit as quickly as possible from the advantages Microservices offer.
- The start is reversible: If Microservices prove not to be suitable for a certain project, they can easily be replaced again.
- Microservices are clearly more than a hype (section 16.3). For being just a hype they have been in use for too long and have been too broadly adapted. Therefore, one should at least experiment with Microservices – and this books invites the reader in many places to do just that.

Try and Experiment

Answer the following questions for an architecture/system you are familiar with:

- Which are the most important advantages of Microservices in this context?
- How could a migration to Microservices be achieved? Possible approaches:
  - Implement new functionalities in Microservices
  - Enable certain properties (e.g. robustness or rapid deployment) via suitable technologies
- What could a project look like which tests the introduction of Microservices with as little expenditure as possible?
  - In which case would this project be a success and the introduction of Microservices therefore sensible?