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 2016-08-12

Tweet This Book!

Please help Eberhard Wolff by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought the #Microservices book by @ewolff http://microservices-book.com/

Contents

Introduction to the Sample

Thank you for your interest in the Microservices book and for downloading this sample!

The sample contains chapter 1 (Preface), chapter 2 (Introduction) and parts of chapter 4 (What are Microservices?) of the complete book. Also it contains the overview for each of the four parts of the book.

The book is a translation from German and offers a comprehensive introduction to Microservices.

Table of Contents of the Complete Book

Part III: Implementing Microservices

1 Preface

Although Microservices is a new term, the concepts that it represents have been around for 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 spoke about small teams which develop and run services with their own databases. Today this structure is called DevOps, and the architecture is known as Microservices.

Later I was asked to develop a strategy for a client that would allow them to integrate modern technologies into their 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.

In 2009, I worked with another client who had divided his complete infrastructure into REST services, each being developed by individual teams. This would also be called Microservices today. Many other companies with a web-based business model had already implemented similar architectures at that time. Lately, I have also realized how Continuous Delivery influences software architecture. This is another area where Microservices offer a number of advantages.

This is the reason for writing this book: A number of people have been pursuing a Microservices approach for a long time, among them some very experienced architects. Like every other approach to architecture, Microservices 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 is 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 the 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 fulfill only one task, but it should perform this task 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 a 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.
- 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 pass, in its entirety, through all phases of the Continuous Delivery pipeline such as development, 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. Internally, Deployment Monoliths can have a modular structure – however all modules have to be brought into production simultaneously.

1.2 Why Microservices

Microservices allow software to be divided into modules making it easier to change the software.

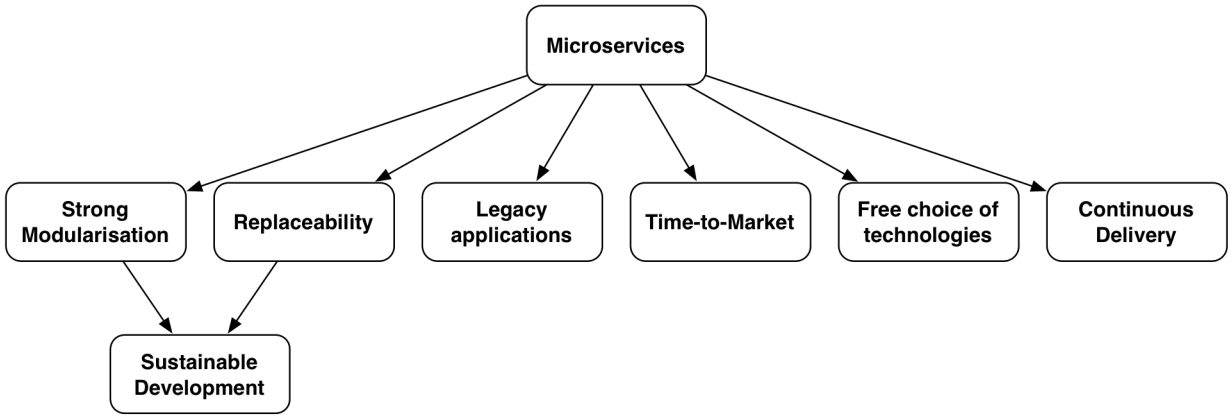

Fig. 1: Advantages of Microservices

Microservices offer a number of important advantages:

Strong Modularization

Microservices offer 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, undesirable 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 such as messages or REST. This makes the technical hurdles for the use of Microservices higher and thus unwanted dependencies are less likely to arise. In principle, it should be possible to achieve a high level of modularization in Deployment Monoliths. However, practical experience teaches us that the architecture of Deployment Monoliths progressively deteriorates over time.

Easy Replaceability

Microservices can be replaced more easily. Other components utilize a Microservice via an explicit interface. If a service offers the same interface, it can replace the Microservice. The new Microservice can use a different code base and even different technologies as long as it presents the same interface. This can often be impossible or difficult to achieve in legacy systems.

Small Microservices further facilitate replacement. The need to replace code in the future is often neglected during the development of software systems. Who wants to consider how a newly built system can be replaced in the future? In addition, the easy replaceability of Microservices reduces 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 the need arises.

Sustainable Development

Strong modularization and easy replaceability allow for sustainable software development. Most of the time working on a new project is straightforward but over longer projects productivity decreases.

One of the reasons is the erosion of architecture. Microservices counteract this erosion by enforcing 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, can by replaced one by one to 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 also modify requests prior to their processing by the legacy system. With this approach it is not necessary to completely replace the legacy system. 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 shorter time-to-market. As mentioned previously, Microservices can be brought into production on a one-by-one basis. If teams working on a large system are responsible for one or more Microservices and if features require changes only to these Microservices, each team can develop and bring features into production without time consuming coordination with other teams. This allows many teams to work on numerous features in parallel and bring more features into production in less time than would have been possible with a Deployment Monolith. Microservices help with 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 removes the need to scale the entire system when it is only a few pieces of functionality that are used intensely. This will often be a significant simplification.

Free Choice of Technologies

When developing Microservices there are no restrictions with regards to the usage of technologies. This gives the ability to test a new technology within a single Microservice without affecting other services. 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 costs low. In addition, it is possible to use specific technologies for specific functions, 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 a small number of Microservices. This reduces the potential risk and enables independent technology decisions for different Microservices. Importantly, it makes the decision to try out and evaluate new, highly innovative technologies easier. This increases the productivity of developers and prevents the technology platform from becoming outdated. In addition, the use of modern technologies will attract well 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 ensure 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 points are strong arguments 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 also deals with prioritization.

Challenges

However, there is no light without shadow. 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 can make working on the architecture challenging.

Refactoring is Difficult.

The strong modularization leads to some disadvantages: Refactoring, where functionality moves between Microservices, is difficult to perform. And, once introduced, it is hard to change the Microservices-based modularization of a system. However, these problems can be reduced with smart approaches.

Domain Architecture is Important.

The modularization into Microservices for different domains is important as it determines how teams are divided. Problems at this level also affect the organization. Only a solid domain architecture can ensure the independent development of a Microservice. As it is difficult to change the modularization once established, 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 require that operations are automated to make sure that operating the platform does not become laborious.

Distributed Systems are Complex.

Developers face increased complexity: 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 explains what Microservices are, why they are interesting and where they are useful. Practical examples demonstrate the impact of Microservices in different scenarios. Chapter 2 explains the structure of the book. To illustrate the importance of Microservices chapter 3 contains detailed scenarios of 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. Finally, Section 2.6 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 the necessary information in order for 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 work best when a business is organized to support a Microservices architecture. In the introduction managers get to understand the basic ideas behind Microservices. Afterwards they can focus on the organizational impact of using 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, help to convey the basic concepts.

Architects

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

The book highlights possible areas for experimentation and additional information sources. These will help the interested reader to test their new knowledge practically and delve deeper into subjects that are of relevance to them.

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) has presented the basic properties as well as the 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 section gives some initial insights into Microservices and points out contexts for applications.

Part II

Part II not only explains Microservices in detail, but 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 not only have technical, but also organizational advantages, and there are good reasons for turning to Microservices from a business perspective.
- 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 plenty 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, technical challenges are discussed.

- Chapter 9 presents the different approaches to the **integration** of and the **communication** between Microservices. This includes not only communication via REST or messaging, but also the 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. However, the tests need to not only 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 implemented technically:

- 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 Nanoservices, which are presented in chapter 15. Nanoservices require specific technologies and a number of compromises. The chapter discusses different technologies and their related advantages and disadvantages.
- Chapter 16 demonstrates how Microservices can be adopted.

2.4 Essays

The book contains essays, which were written by Microservices experts. The experts were asked to record their main findings about Microservices on approximately two pages. Sometimes these essays 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 the chapters that are primarily meant for people with a different type of job. However, the chapters focussed on topics that are most relevant for a certain audience are 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 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, they are encouraged to complete the related exercises to get a better grasp of the topics presented in that chapter.

2.6 Acknowledgment

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.

As a native speaker Matt Duckhouse has added some significant improvements to the text and improved its readability.

My employer innoQ has also played an important role throughout the writing process. Many of the 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 also go 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.

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

This part of the book discusses the different facets 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 a number of different ways to define Microservices. The different definitions are based on different aspects of Microservices. They also show 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 explains the relationship between Microservices, architecture and organization by using the Law of Conway.
- Finally section 4.3 presents a domain architecture 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” conveys the fact that the size of the service matters, obviously Microservices are supposed to be small.

One way to define the size of a Microservice is to count the Lines of Code (LoC[1]). However, such an approach has a number of problems:

- It depends on the programming language used. Some languages require more code than others to express the same functionality – and Microservices are explicitly not supposed to predetermine the technology stack. Therefore, 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 may argue for larger or smaller Microservices.

[1] http://yobriefca.se/blog/2013/04/28/micro-service-architecture/

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 they are 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. For this type of system the "First Rule of Distributed Object Design[2]" 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 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[3] teaches us 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 is an argument for larger Microservices. When a Microservice contains a UI and fully implements a specific part of the domain, it can operate without calling on other Microservices in most cases because all components of this part of the domain are implemented within one Microservice. The desire to prevent distributed communication is another reason to build systems according to the domain.

Sustainable Architecture

Microservices also use distribution 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,

[2] http://martinfowler.com/bliki/FirstLaw.html

[3] http://martinfowler.com/articles/distributed-objects-microservices.html

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 also create challenges, for instance during refactoring. When it becomes apparent that a piece of functionality does not fit 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 inevitably leads to a new implementation. Such problems do not arise when functionalities are moved within a Microservice. This consideration may argue for larger Microservices and 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 and deploy those features into production independently of other teams. By ensuring this, the architecture allows for the scaling of development without requiring too much coordination effort between 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 never grow so large that three to nine people cannot develop it further by themselves. In addition to the sheer size, the number of features to be implemented in an individual Microservice plays an important role. Whenever a large volume 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 infrastructure. In the case of larger Microservices the system overall contains fewer Microservices and consequently requires less infrastructure.

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 bad quality that it cannot be developed any further. The replaceability of Microservices is an advantage when compared to monolithic applications, which can hardly 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. The smaller a Microservice is, the easier it is to replace it with a new implementation. Above a certain size a Microservice may be difficult to replace as it then poses the same challenges as 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 is 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 and 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 the delivery of the decision that changes have to be reversed has to be guaranteed. However, communication within networks is unreliable. Until it is decided whether a change may take place, further changes to the data are barred. If additional changes have taken place, it might not longer be possible 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 (see Section 9.4). With this 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, also has 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 lead to the use of mechanisms such as compensation transactions for Microservices.

Summary

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

- The team size sets an upper limit: A Microservice should never be so 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 reduces 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 size of each Microservice will be larger.
- Similarly, distributed communication increases with the number of Microservices. 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 should not be so small that consistency and transactions comprise several Microservices.

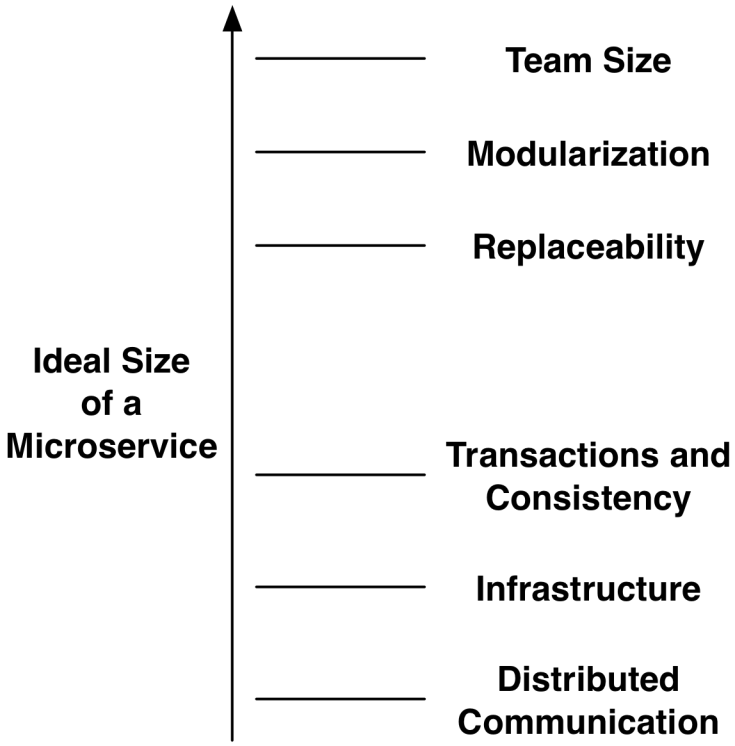

Fig. 5: Factors Influencing the Size of a Microservice

These factors 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 these criteria. This also depends on the technology being used. Especially the effort necessary for providing infrastructure for a Microservice and 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 to 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 great is the effort required 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[4] 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 to 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 architectural 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 architectural parts.

From the Law of Conway it can also be deduced that design modularization is sensible. Via such a design it is possible to ensure that not every team member has to constantly coordinate with every

[4] http://www.melconway.com/research/committees.html

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 architectural 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 increasingly 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 create this type of organization. Moreover, this distribution also appears logical. Team members can easily support each other, and technical exchange is also facilitated.

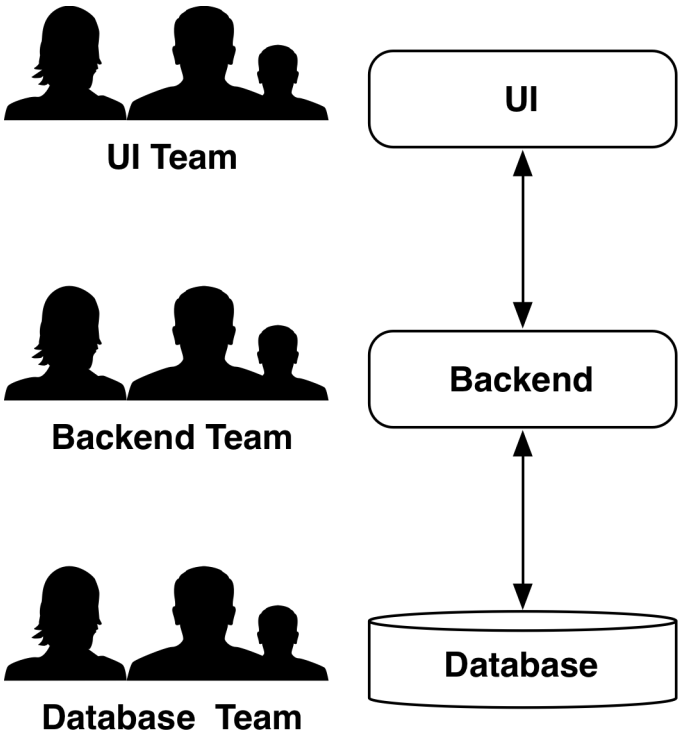

Fig. 6: Technical Project Distribution

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 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. Therefore it takes three sprints to implement a single feature.

In the end this approach creates a large number of dependencies as well as a high communication and coordination overhead. 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 instead on the notion that developers with similar skills should be grouped together within the organization. This organization becomes an obstacle to a design driven by the domain like Microservices whose development is not compatible with 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 the order process. These teams work on their 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 to the team names. Components and teams are synonymous. This approach corresponds to the idea of so-called cross functional teams, as proposed by methods such as Scrum. These teams should encompass different roles so that they can cover a large range of tasks. Only a team designed along such principles can be in charge of a component – from engineering requirements via implementation through 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 them 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.

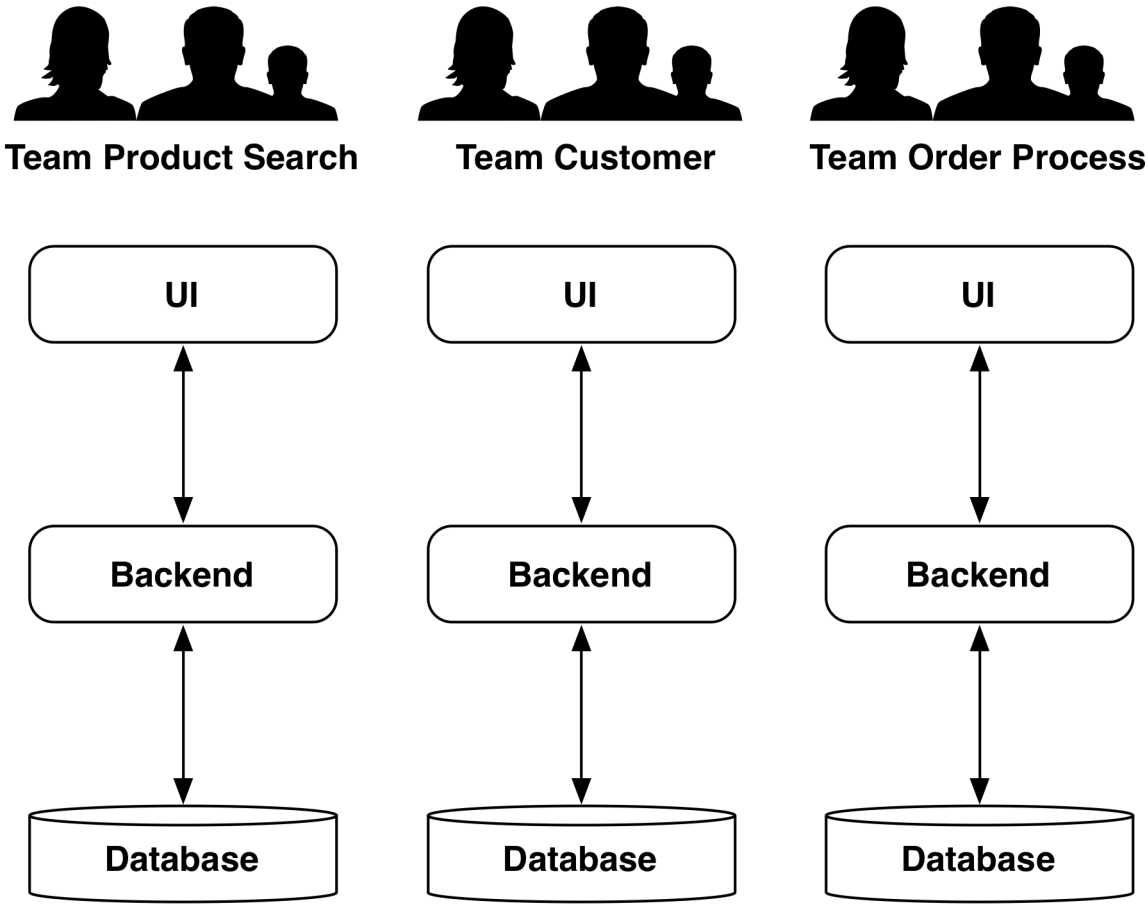

Fig. 7: Project by domains

Ultimately, the key message to be taken 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 separation of functionality into components also facilitates maintenance. Since an individual team is responsible for 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 people who 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 a 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 makes architectural changes between Microservices more difficult and makes the overall process less flexible. Whenever a piece of 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 change renders 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?

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.

Part IV: Technologies

This part of the book moves away from the theoretical to show the technologies involved in actual implementations of Microservices. Chapter 14 contains a complete example of 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, service discovery, communication, load balancing and tests.

Even smaller than Microservices are the Nanoservices discussed in chapter 15. They require special technologies and a number of compromises. The chapter introduces technologies that can implement very small services - Amazon Lambda for JavaScript, Python 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 and Python. The programming language Erlang can also be used for very small services and it is able to integrate with other systems. Seneca is a specialized JavaScript framework for the implementation of Nanoservices.

At the close of the book chapter 16 concludes by re-iterating the benefits of using Microservices and discusses how you might go about starting to use them.