Microservices

FLEXIBLE SOFTWARE ARCHITECTURE

EBERHARD WOLFF

Microservices

Microservices

Flexible Software Architecture

Eberhard Wolff

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco
Amsterdam • Cape Town • Dubai • London • Madrid • Milan
Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Control Number: 2016952028

ISBN-13: 978-0-134-60241-7
ISBN-10: 0-134-60241-2

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing: October 2016

Editor-in-Chief
Mark Taub

Acquisitions Editor
Chris Guzikowski

Development Editor
Chris Zahn

Managing Editor
Sandra Schroeder

Project Editor
Lori Lyons

Production Manager
Dhayanidhi

Copy Editor
Warren Hapke

Indexer
Erika Millen

Proofreader
Sudhakaran

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
codeMantra

To my family and friends for their support.

And to the computing community for all the fun it has provided to me.

Contents at a Glance

Contents

Preface

Although “microservices” is a new term, the concepts that it represents have been around for long time. In 2006, Werner Vogels (CTO at 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 that develop and run services with their own databases. Today this structure is called DevOps, and the architecture is known as micro services.

Later I was asked to develop a strategy for a client that would enable 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.

Overview of the Book

This book provides a detailed introduction to microservices. Architecture and organization are the main topics. However, technical implementation strategies are not neglected. A complete example of a microservice-based system demonstrates a concrete technical implementation. The discussion of technologies for nanoservices illustrates that modularization does not stop with microservices. The book provides all the necessary information for readers to start using microservices.

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-based architecture. In the introduction, managers 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, helps 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.

Structure and Coverage

The book is organized into four parts.

Part I: Motivation and Basics

The first part of the book explains the motivation for using microservices and the foundation of the microservices architecture. Chapter 1, "Preliminaries," presents the basic properties as well as the advantages and disadvantages of microservices. Chapter 2, “Microservice Scenarios,” presents two scenarios for the use of microservices: an e-commerce application and a system for signal processing. This section provides some initial insights into microservices and points out contexts for applications.

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

Part II not only explains microservices in detail but also deals with their advantages and disadvantages:

- Chapter 3, “What Are Microservices,” 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 a technical perspective based on domain-driven Design and Bounded Context.
- The reasons for using microservices are detailed in Chapter 4, “Reasons for Using Microservices.” Microservices have not only 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 5, “Challenges.” Among these are technical challenges as well as problems related to architecture, infrastructure, and operation.
- Chapter 6, “Microservices and SOA,” 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: Implementing Microservices

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 7, “Architecture of Microservice-Based Systems,” describes the architecture of microservices-based systems. In addition to domain architecture, technical challenges are discussed.
- Chapter 8, “Integration and Communication,” 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 9, "Architecture of Individual Microservices," 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 10, "Testing Microservices and Microservice-Based Systems." Tests have to be as independent as possible to enable 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 11, "Operations and Continuous Delivery of Microservices." Microservices generate a huge number of deployable artifacts and thus increase the demands on the infrastructure. This is a substantial challenge when introducing microservices.
- Chapter 12, "Organizational Effects of a Microservices-Based Architecture," illustrates how microservices also influence the organization. After all, microservices are an architecture, which is supposed to influence and improve the organization.

Part IV: Technologies

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

- Chapter 13, "Example of a Microservices-Based Architecture," 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, that 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 14, "Technologies for Nanoservices." Nanoservices require specific technologies and a number of compromises. The chapter discusses different technologies and their related advantages and disadvantages.
- Chapter 15, "Getting Started with Microservices," demonstrates how microservices can be adopted.

Essays

The book contains essays that were written by experts of various aspects of microservices. 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 correct 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 viewpoints in order to subsequently develop an opinion.

Paths through the Book

The book offers content suitable 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 focused on topics that are most relevant for a certain audience are indicated in Table P.1.

Table P.1 *Paths through the Book*

Chapter	Developer	Architect	Manager
1 - Preliminaries	X	X	X
2 - Microservice Scenarios	X	X	X
3 - What Are Microservices?	X	X	X
4 - Reasons for Using Microservices	X	X	X
5 - Challenges	X	X	X
6 - Microservices and SOA		X	X
7 - Architecture of Microservice-Based Systems		X	
8 - Integration and Communication	X	X	
9 - Architecture of Individual Microservices	X	X	
10 - Testing Microservices and Microservice-Based Systems	X	X	
11 - Operations and Continuous Delivery of Microservices	X	X	

(*Continued*)

Table P.1 *Continued*

Chapter	Developer	Architect	Manager
12 - Organizational Effects of a Microservices-Based Architecture			X
13 - Example of a Microservice-Based Architecture	X		
14 - Technologies for Nanoservices	X	X	
15 - Getting Started 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 13 and 14, which deal with concrete technologies and code.

The instructions for experiments, which are given in the sections "Try and Experiment," help deepen your understanding by providing practical exercises. Whenever a chapter is of particular interest to you, you are encouraged to complete the related exercises to get a better grasp of the topics presented in that chapter.

Supplementary Materials

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/.

Register your copy of *Microservices* at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN 9780134602417 and click Submit. Once the process is complete, you will find any available bonus content under "Registered Products."

Acknowledgments

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—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 due 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.

Addison-Wesley enabled me to take the English translation to the next level. Chris Zahn, Chris Guzikowski, Lori Lyons and Dhayanidhi Karunanidhi provided excellent support for that process.

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.

This page intentionally left blank

About the Author

Eberhard Wolff, a Fellow at innoQ in Germany, has more than 15 years of experience as an architect and consultant working at the intersection of business and technology. He has given talks and keynote addresses at several international conferences, served on multiple conference program committees, and written more than 100 articles and books. His technological focus is on modern architectures—often involving cloud, continuous delivery, DevOps, microservices, or NoSQL.

This page intentionally left blank

Part II

Microservices: What, Why, and Why Not?

Part II 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 3, "What Are Microservices," 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 issues are the size of a microservice, Conway's Law as organizational influence, and domain-driven design particularly with respect to *Bounded Context* from a domain perspective. Furthermore, the chapter addresses the question of whether a microservice should contain a UI.

Chapter 4, "Reasons for Using Microservices," focuses on the advantages of microservices, taking alternatingly technical, organizational, and business perspectives.

Chapter 5, "Challenges," deals with the associated challenges in the areas of technology, architecture, infrastructure, and operation.

Chapter 6, "Microservices and SOA," distinguishes microservices from service-oriented architecture (SOA). 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.

This page intentionally left blank

Chapter 3

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 or her own definition of the term microservice—depending on the individual project scenario.

The chapter discusses the term microservice from different perspectives:

- Section 3.1 focuses on the size of microservices.
- Section 3.2 explains the relationship between microservices, architecture, and organization by using the Conway's Law.
- Section 3.3 presents a domain architecture of microservices based on domain-driven design (DDD) and bounded context.
- Section 3.5 explains why microservices should contain a user interface (UI).

3.1 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.

Modularization

One factor is modularization. Teams develop software in modules to be better able to deal with its complexity; instead of having to understand the entire software package, developers only need to understand the module(s) 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, because changes require an understanding of the entire module. Thus it is very sensible to keep microservices as small as possible. On the other hand, microservices, unlike 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 for this is that

1. http://yobriefca.se/blog/2013/04/28/micro-service-architecture/
2. http://martinfowler.com/bliki/FirstLaw.html

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, because 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 limit 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 or she 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, because 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. If 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

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

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 7.3.

Team Size

The independent deployment of microservices and the division of the development effort 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 enables 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 enable the implementation of different features. When microservices are smaller, a team can be responsible for several microservices, which together enable 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, which is typically three to nine people. Thus a microservice should never grow so large that a team of 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 number of changes is necessary within a short time, a team can rapidly become overloaded. Section 12.2 highlights alternatives that enable 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 code for the microservice has to be versioned independently of that for 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 automation enables further reduction of the microservice size. Companies that have been working with microservices for some time usually simplify the creation of new microservices by providing the necessary infrastructure in an automated manner.

Additionally, some technologies enable reduction of the infrastructure overhead to such an extent that substantially smaller microservices are possible—however, with a number of limitations in such cases. Such nanoservices are discussed in Chapter 14, "Technologies for Microservices."

Replaceability

A microservice should be as easy to replace as possible. Replacing a microservice can be sensible when its technology becomes outdated or if 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 reasonably maintained anymore, its development has either to be continued in spite of the associated high costs or a similarly 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, for 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.
- **Consistency** means that data is consistent before and after the execution of a transaction—database constraints, for instance, are not violated.

- **Isolation** indicates that the operations of transactions are separated from each other.
- **Durability** indicates permanence: changes to the data are stored and are still available after a crash or other interruption of service.

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 made by all microservices would have to be reversed. This is laborious and hard to implement, for 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 no longer be possible to reverse a certain change. However, when microservices are kept from introducing data changes for some time, system throughput is reduced.

However, when communications occur via messaging systems, transactions are possible (see section 8.4). 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 be achieved only 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 exactly at the same point of time and maybe not even within one minute of order processing because 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 data being processed is part of the same microservice. Therefore, data changes 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 classic 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, rental car, and flight are available. If all are available, everything is reserved. If, for instance, the hotel room suddenly becomes unavailable, the reservations for the flight and the rental car have 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 also 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 Figure 3.1):

- The team size sets an upper limit; a microservice should never be so large that one very large team or 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—that is, 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 enables a developer to understand all its aspects 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 decides 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.

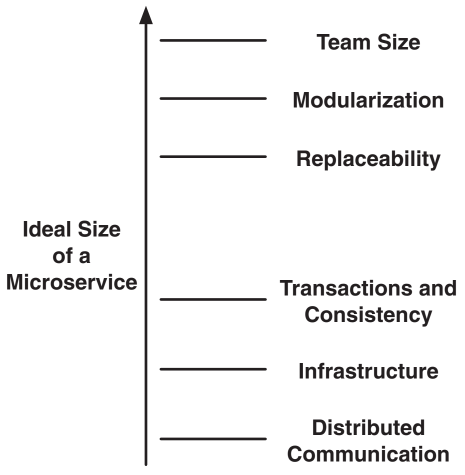
Team Size
Modularization
Replaceability
Ideal Size of a Microservice
Transactions and Consistency
Infrastructure
Distributed Communication

Figure 3.1 *Factors Influencing the Size of a Microservice*

- Similarly, distributed communication overhead 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 must be ensured across several microservices.

These factors not only influence the size of microservices but 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, along with 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 14 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 13, “Example of a Microservice-based Architecture.”

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 is it 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, so you should also think of appropriate limits in those terms.

3.2 Conway's Law

Conway's Law[4] was coined by the American computer scientist Melvin Edward Conway and indicates the following:

> 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 meant to apply not only to software but to any kind of design. The communication structures that 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.

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

Reasons for the Law

Conway's Law derives 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 that are responsible for the respective parts of the architecture.

From Conway's Law 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 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 the external features 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 that are geographically close to each other, because it is easier for them to communicate with each other. In the end, the Conway's Law focuses not 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 Conway's Law is viewed as a limitation, especially from the perspective of software development. Let us assume that a project is modularized according to technical aspects (see Figure 3.2). All developers with a 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.

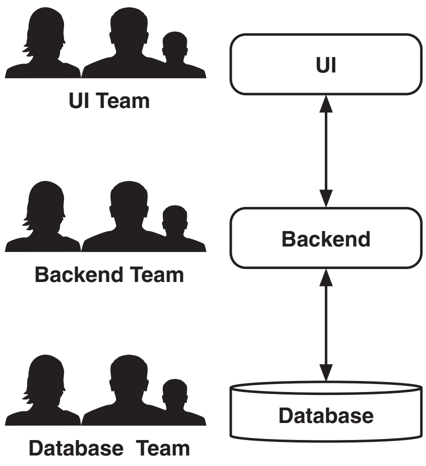

Figure 3.2 *Technical Project Distribution*

According to Conway's Law, 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, this distribution 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 a 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, Conway's Law 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. Figure 3.3 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, back-end, and database. By the way, the domain components are not explicitly named in the figure, for 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, developers only have to talk to developers 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.

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 overhead 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.

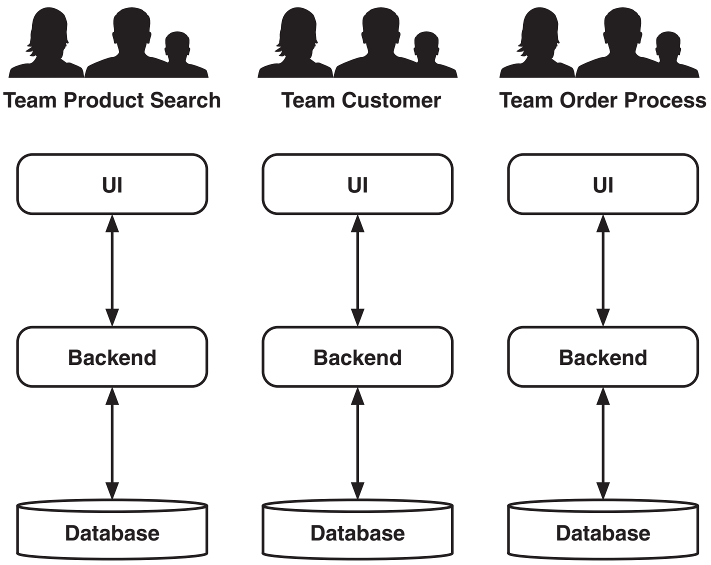

Figure 3.3 *Project by Domains*

The teams need requirements to work upon. This means that the teams need to contact people who define the requirements. This affects the organization beyond the projects, for the requirements come from the departments of the enterprise, and these also 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, each of which 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 also to bring the features 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 is it divided 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?

3.3 Domain-Driven Design and Bounded Context

In his book of the same title, Eric Evans formulated domain-driven design (DDD)[5] 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 from Evan's book are written in *italics*.

5. Eric Evans. 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley.

Domain-driven design is important for understanding microservices, for 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 maximal benefit to be derived from independent development in different teams, as several features can be implemented in parallel without the need for extended coordination.

Ubiquitous Language

DDD defines a basis for 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, for 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 of a *Value Object*, for 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 that 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*, for 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 consistency is necessary, 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. If 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, along 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, for 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.

Bounded Context: An Example

The customer from the e-commerce system shall serve as an example for a *Bounded Context* (see Figure 3.4). 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.

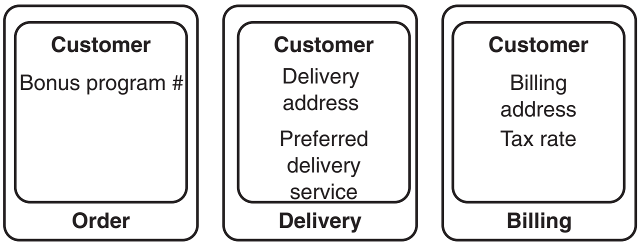

Figure 3.4 *Project by Domains*

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 (see below).

(*continued*)

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 7.2). 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 the 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, for 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 *Anticorruption Layer* translates a domain model into another one so that both are completely decoupled. This enables 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 a 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 a 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, designing microservices that contain multiple *Bounded Contexts* should be avoided, for 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, Anticorruption 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 coordinate their efforts closely: 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 (see Figure 3.5).

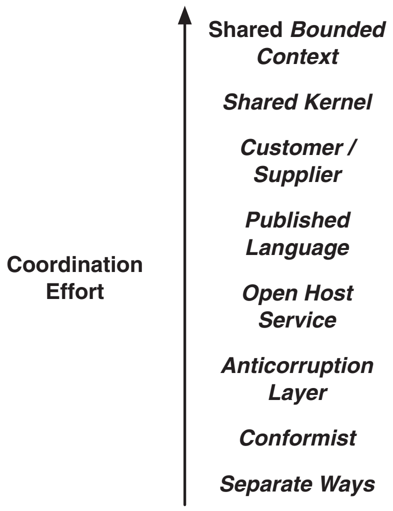

Figure 3.5 *Communication Effort of Different Collaborations*

As in the case of Conway's Law from section 3.2, 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 part of the architecture, 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* with respect to 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.
- A *Responsibility Layer* divides the system into layers with different responsibilities. Layers can call other layers only 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 it is best not to determine the overall structure too rigidly. Instead, the order 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 (see also Chapter 7, "Architecture of Microservice-based Systems").

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 7.2.
- How do the Bounded Contexts cooperate? (Anticorruption 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 because you need to get a new perspective on the system and how the domains are modeled in the system.

3.4 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, that is, if you've been focusing on individual systems so far, let me give you the 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 and 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 a 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 why that might seem a very compelling thing to attempt. After all, even a nontechnical 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 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 people 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 all the participants 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 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 easy to browse and search. (Think of providing a big elastic search index as opposed to a central UML model.)

What you actually need to do 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 enable people to work as independently as possible. A CDM of the kind I've described above is the exact opposite.

3.5 Microservices with a UI?

This book recommends that you 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 area of 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 of whether the integration of UI into microservices is really required. Ultimately, microservices should not be too large. And when logic is supposed to be used by multiple frontends, a microservice consisting of pure logic without a 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 enables implementation of features without coordination across teams.

Focusing on microservices with a 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 a protocol. The RESTful-HTTP interface delivers JSON or XML, the Web UI HTML. If the UI is a Single-Page Application, 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 8.1 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).[6] 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 SCSs 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 is something a team works on and which represents a unit in the domain architecture. This can be an order process or a 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 with the microservice definition put forward in this book. However, the size given in the SCS world corresponds to what this book denotes as nanoservices (see Chapter 14).
- This book refers to nanoservices as units that are still individually deployable but 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 the UI and logic should at least be implemented by the same team.

6. http://scs-architecture.org

3.6 Conclusion

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

- Section 3.1 focuses on the size of microservices. But a closer look reveals that the size of microservices itself is not that important, even though size is an influencing factor. However, this perspective provides a first impression of 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 3.2) shows that the architecture and organization of a project are closely linked—in fact, they are nearly synonymous. Microservices can further improve the independence of teams and thus ideally support architectural designs that 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 because of the possibility for technical independence.
- In section 3.3 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 3.5 demonstrates that microservices should contain a UI to be able to implement the changes for functionality within an individual microservice. This does not necessarily have to be a deployment unit; however, the 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:

- an organization (This is supported by Conway’s Law.)
- a technical approach (This can be microservices.)
- 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?

This page intentionally left blank

Index