From Monolith to Microservices: A not yet defined Approach

Bachelor's Thesis of

Niko Benkler

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Prof. B
Advisor:	Dr. Robert Heinrich

xx. Month 20XX – xx. Month 20XX

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

PLACE, DATE

. . . . . .

(Niko Benkler)

Abstract

Powered by the rise of cloud computing, agile development, DevOps and continuous deployment strategies, the microservice architectural pattern emerged as an alternative to monolithic software design. Microservices, as a suite of independent, highly cohesive but loosely coupled services, overcome the shortcoming of centralized monolithic architectures. Therefore, prominent companies recently migrated their monolithic legacy applications successfully to microservice-based architecture. They key challenge is to find an appropriate partition of legacy applications - namely *microservice identification*. So far, the identification process is done intuitively based on the experience of system architects and software engineers - mainly by virtue of missing formal approaches and a lack of automated tool support.
However, when application grow in size and become progressively complex, it is quite demanding to decompose the system in appropriate microservices. This thesis provides a formal identification model to tackle this challenge. The identification process is based on... // Hier jetzt noch kurz beschreiben was ich eigentlich machen will We use

Zusammenfassung

Deutsche Zusammenfassung

Contents

List of Figures

List of Tables

1. Introduction

The monolithic software architecture is the traditional pattern to design software, where functionality is bundled in one single, large application [5]. Although monoliths have their strength, like fast development and simple deployment, they become an obstacle when they grow in size and become more complex [21]. Incomprehensible code structure makes it difficult to add functionality, fix bugs and enable new software engineering approaches like Continuous Delivery and Continuous Deployment. Besides, the rise of cloud computing demands a new architecture that can fully exploit the rich set of features given by the cloud infrastructure [16].
Microservice Architecture is about to become a promising alternative to overcome the shortcomings of centralized, monolithic architectures. Inspired by service-oriented computing, microservices gain popularity in both, academia and industry [2]. Benefits like the increase of agility, resilience or scalability [22], the ability to use different technology stacks and independent deployment [3], and the efficient resource utilization in cloud environments [16] explain, why big companies like Google, Netflix, Amazon, eBay [5] and Uber [22] migrated their monolithic architectures to microservice-based applications.
This thesis describes the current state of the art regarding microservices extraction and provides a systematic approach to decompose a (legacy) application into microservices.

1.1. Motivation

Monolithic software applications develop over time and become more and more complex. The software structure is highly coupled and hard to maintain [9]. To tackle this issues, software engineers started to decompose their system into modules and provide the functionality over the network as Web Services [11]. The so-called *Service-oriented Architecture* (SOA) provides logical boundaries between the different software modules to address the design challenge of distributed systems. Nevertheless, Baresi et. al state that the boundaries between modules in SOA are too flexible and the application results in "a big ball of mud" [3]. Microservices make these boundaries physical as each service runs in its own process and only communicates with other services through well-defined lightweight mechanisms like REST [22]. Chen et al. consider the microservice architecture as a particular approach for SOA [5]. Others look at it as an evolution of SOA with differences in service reuse [3] or consider it to be the "contemporary incarnation of SOA" combined with modern software engineering practices like continuous deployment [11]. There is no consensus about the relationship between microservices and SOA, but clearly, SOA paved the way for the rise of the microservice pattern.
The microservice architecture has many advantages over the monolithic style. Sec.2.2 elaborates the main aspects of microservices, including several benefits. Netflix, for in-

stance, is able to cope with one billion calls a day to its video streaming API, by migrating their monolithic system to a high flexible, maintainable and scalable microservice architecture [5]. Consequently, moving existing applications to a microservice landscape is a hot topic in academia and industry [2].
Nevertheless, decomposing a system in loosely coupled, fine-grained and independent microservices is a time consuming task that requires tedious manual effort [11] and is technically cumbersome [6]. So far, it is done mainly intuitively and relies on the experience of software architects and system designers. Hence, a formal approach to identify microservices is required. This thesis intends to describe an approach to systematically decompose a monolithic system into loosely coupled, but high cohesive fine-grained microservices.

1.2. Problem Statement

The microservice architecture is a fast rising approach to structure a system in high cohesive but loosely coupled and independent services. Many companies like Amazon, migrated their monolithic legacy software to microservice in order to fully leverage the benefits of cloud computing and new software engineering approaches like Continuous Deployment [16]. Large applications are decomposed into small, independent microservices where each service can be independently scaled and deployed.
However, one of the biggest problem in designing a microservice architecture is to decompose a monolithic application into a suite of small services while keeping them loosely coupled and high cohesive. This challenging task is also known as *microservice identification* [2].

Baresi et al. state that "proper" microservice identification defines how systems will be able to evolve and scale [3]. Others claim, that finding the optimal microservice boundaries and service granularity is the key design decision to fully leverage the benefits of microservices [10] [12].
So far, the partition is performed mainly intuitively based on the experience and know-how of experts that perform the extraction. Hassan et al. criticises a lack of systematic approaches to reduce the complexity of the extraction process [12]. Extracting microservices from monoliths therefore requires tedious manual effort and can be very costly [22] [17]. This thesis aims to reduce the complexity by providing an approach to systematically decompose a monolithic application into microservice. In the following, the challenges of microservice identification are presented.

1.3. Contributions

1.4. Thesis Outline

2. Background

2.1. Monolithic Software Architecture

The monolithic software architecture is a well-known and the most widely used pattern for Enterprise Applications, which usually are built in three main parts (top to bottom): The client-side user interface (Tier 3), the server-side application (Tier 2) and the persistence layer (Tier 1). The server-side application - *the monolith* - is a single unit and deployed on one application server [21]. The software structure, if well defined, is composed of self-contained modules (i.e. software components), where each module consists of a set of functions [6]. The monolith implements a complex domain model, including all functions, many domain entities and their relationships. For small applications, this approach works relatively well. They are simple to develop, test and deploy [22]. Fast prototyping is supported by the current frameworks and development environments (IDE), which are still oriented around developing single applications [21].
But once they grow in size, they become exceedingly difficult to understand and hard to maintain without reasonable effort [22] [10]. A complex and large code base prevents a fast addition of new features and makes the application risky and expensive to evolve [15]. Alterations to the system, even though they might be small, result in a redeployment of the whole monolith application due to its nature being a single unit [22]. Moreover, it is difficult to adopt newer technologies without rewriting the whole application, as monolith are build on a specific technology stack [21] [17].
Scaling is only possible by duplicating the entire application - namely *horizontal scaling*. Consequently, large portions of the infrastructure remains unused, if only parts of the application need to be upscaled or even used [13] [9].
Chen et al. provide a short résumé:

"Successful applications are always growing in size and will eventually become a monstrous monolith after a few years. Once this happens, disadvantages of the monolithic architecture will outweigh its advantages." [5]

2.2. Microservices

"The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API." [8]

2.2.1. Definition

The above quotation is a widely adopted definition of the term *Microservice*, provided by M. Fowler and J. Lewis, the pioneers of the microservice architecture. However, the term is not formally defined. Amiri et al. describes microservices as a collection of cohesive and loosely coupled components, where each service implements a business capability. The author introduces three principles upon which the architecture is build: *Bounded Context, Size, Independence* [2].
The first principle is about related functionality, that is combined in a single business capability - the *bounded context* [22]. Each capability is implemented by one microservice. The *Size* of a microservice is defined by the number of features it provides (namely bundled functional capabilities) [14]. There is no consensus about the "proper" size of a microservice [20], but several guidelines exists: Services should focus on one business capability only [2]. Others state, that the size of a microservice should not exceed a level, where it cannot be rewritten within six weeks [14]. However, the sizes vary from system to system [22] and even different sizes for each microservice in a specific system are possible [20]. The bottom line of *Independence* is in Amiri's description of microservices as "a collection of high cohesive and loosely coupled components" [2]. High cohesive services implement a relatively independent piece of business logic (at the most one business capability). Further, microservices should hardly depend on each other, which is the idea of being loosely coupled [5].
Communication between microservices is achieved by lightweight message passing mechanisms such as *REST*. Each service exposes a well defined interface (*API*) with endpoints that provide information using standard data formats [22]. The design of microservices mainly follows the *Single Responsibility Principle (SRP)*: Each service should not have more than one reason to change [7]. The SRP mainly corresponds to the idea of not implementing more than one business capability. The following covers the benefits and challenges of the microservice architecture.

2.2.2. Benefits

Fast and Independent Deployment
As a matter of fact, each microservice is deployed independently [3]. Changes to the code do not result in a full redeployment of the entire application [22]. Consequently, software developers are able to react much quicker to changes in business requirements. This includes an acceleration in error correction. Per contra, any changes in a monolithic code base requires a time consuming build of a new version and the redeployment of the entire application [8].

Availability, Resilience and Fault Isolation
Microservices are designed to operate independently of each other and to tolerate failure of services [8]. Large parts of the application remain unaffected of partly failures and the availability of the system is, at least partly guaranteed. Monolithic application do not provide this type of fault isolation. If a failure occurs, the whole application remains unavailable as it is usually running in a single process [17].

Scalability and Resource Utilisation
Small and independent microservices allow more fine-granular horizontal scaling [14]. Single services can be duplicated to cope with changing workload during runtime [5]. Thus, dynamic (de-)allocation of resources on demand prevent infrastructure from being idle [6]. Scaling monoliths can only be attained by duplicating the entire application , leaving resources unused [10]. Further, each microservice is deployed on the best suitable infrastructure for its needs, allowing a more efficient system organization [21].

Improved Productivity
In traditional software development, teams are divided based on their expertise: Database architects, UI-developers and server-side engineers, resulting in a three-tiered application (cf. Sec.2.1). Additionally, software engineers are responsible for the development only. Deployment is part of the operations team. This team structure results in high communication overhead and slows down the productivity [18].
In contrast, microservices are organized around business capabilities requiring cross-functional and independent teams [2]. Each team has the full range of skills required for the end-to-end realization of a microservice, including UI-development, database architecture, back-end engineers and project management. This minimizes the communication and interaction between the teams and thus, speeds up the productivity. Ultimately, microservices enable a more agile flow of development and operation [10], also referred as *DevOps*.

Neutral development technology
Microservices are highly decoupled from each other, as they use standardized and lightweight communication mechanisms such as REST [22]. Microservices can be realized using different programming languages, technologies and even deployment environments [5]. Developers are consequently not longer limited to use a single technology for the whole application. They can choose the most appropriate technology for each particular business problem or try out some new technology without rewriting the whole application [11] [15].

2.2.3. Challenges

The previous section provides a vast amount of benefits that come with microservices. However, it is not the panacea of software engineering and has to face some challenges before being able to fully benefit from them. The challenges are further described in the following.

Expensive Communication
Microservice use network protocols such as *HTTP* to communicate with each other. Compared to standard, inter process communication (*IPC*) as used in monoliths, remote procedure calls a more expensive [1]. As a consequence, applications experience a decrease in performance as network communication is generally slower than IPC.

Technical Challenges

Microservices require a high degree of infrastructure automation [9]. The benefits of fast and independent deployment cannot be utilized, if it has to be done manually. Dynamic (de-)allocation of resources when scaling individual microservices need a well defined and structured cloud environment [16]. Besides, the distributed microservice landscape complicates the logging mechanisms and performance monitoring [1]. Traditional centralized logging, as it is used in monolithic applications, is not longer applicable. Instead, a careful aggregation system to gather logging and monitoring data from each service is required.

Organizational Challenges

The microservice approach needs the establishment of cross-functional teams [8]. Adopting *Continuous Practices*, such as *Continuous Deployment*, are essential for the success of a profitable microservice architecture. Therefore, closer collaboration between development teams, operational staff and management has to be established. In summary, a costly and time consuming restructuring process of the entire organization is required [4].

Data Consistency

Distributed systems need to share data. Heinrich et al. propose two concepts for the database architecture [22]: The first concept applies the basic idea of the microservice approach, as it splits the database into several parts. Each microservice has its own database which manages the entities that belong to the corresponding bounded context. Higher speed and horizontal scaling are facing data consistency issues. Data needs to be synchronized which leads to inconsistency, if services are unavailable. The second concept is about sharing a single database. This approach overcomes the issue of consistency, as data is stored centrally. But sharing results in a loss of independence. Scaling can only be achieved through replicating the whole database. Research revealed, that the first concept is preferred [22].

Decomposition

Decomposing a system into microservice is a very complex task that requires experienced system architects and domain experts [8]. Identifying the right granularity of microservice is one of the key issues. Too fine grained services cause inefficiency due to a high amount of expensive inter-service calls [20]. Developing the basic communication infrastructure adds additional complexity and slows down the initial developing process [21].

3. CoCoME

3.1. Introduction to CoCoME

[19]

4. State of the Art

This chapter outlines the current state of the art regarding microservice identification. Sec. 4.1 presents the search strategy and several existing approaches (Table 4.1) that deal with the identification of microservices. Thereupon, the approaches are further explained and finally compared on the basis of several criteria.

4.1. Literature Review

The approaches mentioned in table 4.1 are the result of an extensive literature research which was conducted using the digital libraries IEEE [1], ACM [2] and SpringerLink [3]. The web search enginge Google Scholar [4] provided further approaches and general information. [22] was provided by the supervisor of this thesis and [11] was cited by various approaches, including [3]. The following search string was used:

["identify" OR "identification" OR "migrating" OR "monolith" OR "decomposition" OR "decompose monolith" OR "decompose"] AND "microservice"

OR

"microservice" AND ["identification" OR "transformation" OR "refactor"]

Table 4.1 presents the 8 most promising approaches regarding the criteria mentioned in table //TODO!!!!!!. Now, a short introduction to each approach is given.

[1] http://ieeexplore.ieee.org
[2] http://portal.acm.org
[3] http://www.springerlink.com
[4] http://scholar.google.com

Link	Titel	Author (Year)	Origin	Search String
[17]	Extraction of Microservices from Monolithic Software Architectures	G. Matzlami et. al. (2017)	Google Scholar	*microservice identification*
[2]	Object-Aware Identification of Microservice	M. J. Amiri (2018)	IEEE	*identification microservices*
[3]	Microservices Identification Through Interface Analysis	L. Baresi et. al. (2017)	SpringerLink	*microservice identification*
[22]	Identifying Microservices Using Functional Decomposition	S. Tyszberowicz et. al. (2018)	*provided*	*n/a*
[20]	Partitioning Microservices: A Domain Engineering Approach	I. J. Munezero et. al. (2018)	ACM	*partition microservices*
[5]	From Monolith to Microservices: A Dataflow-Driven Approach	R.Chen et. al	IEEE	monolith microservice
[6]	Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems	A. De Alwis et. al. (2018)	Google Scholar	identify microservices
[11]	Service Cutter: A Systematic Approach to Service Decomposition	M. Gysel et. al. (2016)	[3]	*n/a*

Table 4.1.: List of authors and approaches

4.2. Approaches

Extraction of Microservices from Monolithic Software Architectures
The approach presented in [17] is a class based extraction model, that uses (meta-)information of a version control system *(VCS)* such as Git[5] to identify microservices. The approach is divided in two phases: The *Construction Phase* and the *Clustering Phase*. Starting with a given code base, the approach uses three different coupling strategies and the information provided by the *VCS* to transform the monolith into a weighted graph. Here, the nodes represent classes, and the edges have weights according to the chosen coupling strategy. In the second phase, a clustering algorithm determines possible microservices (each cluster is a microservice candidate).

Object-Aware Identification of Microservice

Microservices Identification Through Interface Analysis

Identifying Microservices Using Functional Decomposition

Partitioning Microservices: A Domain Engineering Approach

From Monolith to Microservices: A Dataflow-Driven Approach

Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems

Service Cutter: A Systematic Approach to Service Decomposition

[5] https://github.com/

5. Solution Overview

6. Evaluation Planning

6.1. Applicability to CoCoME

6.2. Comparison to Functional Decomposition Approach

7. Timetable

7.1. Milestones

Bibliography

[1] N. Alshuqayran, N. Ali, and R. Evans. "A Systematic Mapping Study in Microservice Architecture". In: (Nov. 2016), pp. 44–51.

[2] M. J. Amiri. "Object-Aware Identification of Microservices". In: (July 2018), pp. 253–256. ISSN: 2474-2473. DOI: 10.1109/SCC.2018.00042.

[3] Luciano Baresi, Martin Garriga, and Alan De Renzis. "Microservices Identification Through Interface Analysis". In: (2017). Ed. by Flavio De Paoli, Stefan Schulte, and Einar Broch Johnsen, pp. 19–33.

[4] Niko Benkler. *From Traditional Development to Continuous Deployment: Strategies and Practices in CI/CD Pipelines*. Accessed on 20.01.2019. URL: https://github.com/Benkler/Proseminar/blob/master/Niko_Benkler_Proseminar.pdf.

[5] R. Chen, S. Li, and Z. Li. "From Monolith to Microservices: A Dataflow-Driven Approach". In: (Dec. 2017), pp. 466–475. DOI: 10.1109/APSEC.2017.53.

[6] Adambarage Anuruddha Chathuranga De Alwis et al. "Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems". In: (2018). Ed. by Claus Pahl et al., pp. 37–53.

[7] D. Escobar et al. "Towards the understanding and evolution of monolithic applications as microservices". In: (Oct. 2016), pp. 1–11.

[8] Lewis Fowler. *Microservices*. Accessed on 17.01.2019. URL: https://martinfowler.com/articles/microservices.html.

[9] P. Di Francesco, P. Lago, and I. Malavolta. "Migrating Towards Microservice Architectures: An Industrial Survey". In: (Apr. 2018), pp. 29–2909.

[10] Jonas Fritzsch et al. "From Monolith to Microservices: A Classification of Refactoring Approaches". In: *CoRR* abs/1807.10059 (2018). arXiv: 1807.10059. URL: http://arxiv.org/abs/1807.10059.

[11] Michael Gysel et al. "Service Cutter: A Systematic Approach to Service Decomposition". In: (2016). Ed. by Marco Aiello et al., pp. 185–200.

[12] S. Hassan, N. Ali, and R. Bahsoon. "Microservice Ambients: An Architectural Meta-Modelling Approach for Microservice Granularity". In: (Apr. 2017), pp. 1–10.

[13] G. Kecskemeti, A. C. Marosi, and A. Kertesz. "The ENTICE approach to decompose monolithic services into microservices". In: (July 2016), pp. 591–596.

[14] S. Klock et al. "Workload-Based Clustering of Coherent Feature Sets in Microservice Architectures". In: (Apr. 2017), pp. 11–20.

[15] Alessandra Levcovitz, Ricardo Terra, and Marco Tulio Valente. "Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems". In: *CoRR* abs/1605.03175 (2016). arXiv: 1605.03175. URL: http://arxiv.org/abs/1605.03175.

[16] J. Lin, L. C. Lin, and S. Huang. "Migrating web applications to clouds with microservice architectures". In: (May 2016), pp. 1–4.

[17] G. Mazlami, J. Cito, and P. Leitner. "Extraction of Microservices from Monolithic Software Architectures". In: (June 2017), pp. 524–531.

[18] Genc Mazlami. *Algorithmic Extraction of Microservices from Monolithic Code Bases*. Accessed on 20.01.2019. URL: https://www.merlin.uzh.ch/contributionDocument/download/10978.

[19] Frank Mittelbach. "How to influence the position of float environments like figure and table in LaTeX?" In: *TUGboat* 35 (2014), pp. 248–254. URL: https://www.latex-project.org/publications/tb111mitt-float.pdf.

[20] I. J. Munezero et al. "Partitioning Microservices: A Domain Engineering Approach". In: (May 2018), pp. 43–49.

[21] Chris Richardson. *Microservices: Decomposing Applications for Deployability and Scalability*. Accessed on 08.01.2019. May 2014. URL: https://www.infoq.com/articles/microservices-intro.

[22] Shmuel Tyszberowicz et al. "Identifying Microservices Using Functional Decomposition". In: (2018). Ed. by Xinyu Feng, Markus Müller-Olm, and Zijiang Yang, pp. 50–65.

A. Appendix

A.1. First Appendix Section

Figure A.1.: A figure

...