Hochschule RheinMain

Design, Implementation and Evaluation of a System to Create a Data Set Supporting Research in the EnergieBroker Platform

Master Thesis

for the

Master of Science

in Computer Science

at RheinMain University of Applied Sciences

by

Niklas Sauer

January 11th, 2022

Project Duration	6 Months
Student ID	1209772
First Reviewer	Prof. Dr. Heinz Werntges
Second Reviewer	Prof. Dr. Robert Kaiser
Supervisor	Johannes Kaeppel

Abstract

Contents

Acronyms

SOA	Service-Oriented Architecture
API	Application Programming Interface
SRP	Single Responsibility Principle
DDD	Domain-Driven Design
ACID	Atomicity Consistency Isolation Durability

List of Figures

List of Tables

Listings

1. Introduction

1.1. Motivation

1.2. Goals and Scope

1.3. Thesis Overview

2. Theoretical Framework

2.1. Microservice Architecture

Popularized by companies like Amazon, Netflix, Uber, LinkedIn and SoundCloud, the microservice architecture has emerged as a pattern to avoid the problems of conventional monolithic designs [8, p. 847] [13, p. 584]. This section provides an overview to the problems faced in monolithic applications, distinguishes the microservice architecture from a traditional Service-Oriented Architecture and lays out its core principles, while noting some of the newly introduced challenges.

2.1.1. The Monolith Problem

A monolith is a software application whose modules cannot be executed independently [4, p. 1]. Hence, a monolith is characterized by requiring to be deployed as a united solution [3, p. 24]. Based on this definition, a set of obstructive characteristics inherent to monolithic applications can be derived:

Maintainability

When developing an application with a single large codebase, it naturally becomes harder to maintain and comprehend [4, p. 2]. The latter is especially true for beginners, slowing down their productivity [3, p. 24]. Further, refactoring changes may touch many parts of the software which might lead to a situation in which refactoring is ignored because it becomes too risky [5, p. 35].

Dependencies and technology lock-in

Monoliths typically suffer from the "dependency hell" problem where an application's modules depend on conflicting versions of a shared library [4, p. 2]. In cases where this is solved, the likelihood of having to make many changes to update to a newer version of a library again increases with a growing codebase. Next to that, it becomes very difficult to change the technology stack, leading to a lock-in and forcing developers to use the same language in every problem domain [3, p. 24] [4, p. 2].

Deployment

Rolling out a new application version requires the complete set of services to be restarted, regardless of whether a service has been altered or not [4,

p. 2]. Similarly, failure of one service leads to downtime across all services [11, p. 970]. The deployment can therefore be viewed as a single point of failure [13, p. 584]. Moreover, deployments are likely sub-optimal due to conflicting resource requirements (e.g. CPU vs. memory-intensive). Developers often have to compromise with a one-size fits all configuration [4, p. 2].

Scaling

By combining multiple services into a single process, scaling can lead to resource wastage since the whole application needs to be scaled up even if an increase in traffic stresses only a subset of modules [8, p. 850] [4, p. 2]. Less popular services consume unnecessary (idle) amounts of resources [13, p. 584]. Setting appropriate scaling thresholds also becomes more challenging because different components may have different resource requirements [3, p. 24].

Kalske, Mäkitalo, and Mikkonen, however, acknowledge that the monolith approach might be the correct choice if the codebase is relatively small or the need for fine-grained scaling has not come up [5, pp. 34, 36]. Villamizar et al. add that monoliths are faster to set up [13, p. 589]. Empirically, most organizations start with something big and slowly transition to a decomposed architecture when scaling problems arise [10, p. 113] [13, p. 590]. Yet, it can be argued that an organization should spend more time on the design upfront since it is easy to introduce tight coupling, hereby hindering future refactoring endeavors [5, p. 34].

2.1.2. Definition

A microservice-based application is one in which the core functionality has been decomposed into many small units that can be independently developed and deployed [6, p. 43] [11, p. 970]. Each unit, a *microservice*, is modeled around a single, clearly defined set of closely related functionalities that can be used independently over the network through a well-defined interface [1] [9, p. 56] [12, p. 176] [2, p. 30]. This definition implies that microservices:

- use independent codebases, thus can build on different technology stacks
- run in distinct processes, thus can fail and scale independently
- are decoupled but can be used as building blocks to form larger services

[1] Cerny, Donahoo, and Trnka draw a comparison to three Unix ideas: a program should fulfill only one task well, be able to work with other programs and use a universal interface [2, p. 31].

2.1.3. Decomposition Techniques

Spitting an application into services should happen along the lines of related processes that can be carried out in isolation. It is not about arbitrarily distributing features across services [9, p. 61]. As in traditional software engineering, the term cohesiveness is used to indicate that a service implements only functionalities strongly related to the concern that it is meant to model [4, p. 2]. Various techniques are cited to determine the breadth of concern:

Single Responsibility Principle (SRP)

Defines a responsibility of a class as a reason to change and states that a class should only have one such reason [7, p. 36] [10, p. 116]. Is analogously applied to microservices. Leads to a large amount of services.

Y-axis of scale cube

Splits an application into distinct sets of related functions. Each set is implemented by a microservice. In a verb-based approach, sets consist of a single function that covers a specific use case, whereas the noun-based approach creates sets of functions responsible for all operations related to a particular entity [7, p. 36]. Leads to large and medium amount of services, respectively.

Domain-Driven Design (DDD)

Refers to the application's problem space, i.e. the business, as the domain. This domain consists of multiple subdomains (e.g. product catalog, order management). Each subdomain is represented by a microservice [1, p. 3]. Leads to a small amount of services.

add cita
tion

Irrespective of the technique chosen, the overall goal should be to minimize later interface changes, i.e. to establish proper service contracts [3, p. 26].

2.1.4. Service Registry Pattern

The law of conservation of complexity states that the complexity of a large system does not vanish when the system is broken up into smaller pieces. Instead, the complexity is pushed to the interactions between these pieces [7, p. 38] [10, p. 114]. Applied to microservice-based applications, this means that developers need to deal with the challenges innate to distributed systems [3, p. 24] [13, p. 589]. One such challenge is the fact that services can no longer be invoked through language level method calls but rather only through the network. Moreover, given the need for scaling, clients are now required to make requests to a dynamically changing set of service instances. And since it is unfeasible to run these instances at fixed locations [2], a pattern known as the service registry is commonly employed [7, p. 37].

A service registry acts as a database of services, storing the various instances along their locations, i.e. the IP address and port number. Instances are added on startup and removed on shutdown. Keeping this in mind, two types of service discovery mechanisms are distinguished:

Client-side

To contact a service, clients obtain the locations of all service instances by querying the registry. The client then needs to perform a load balancing algorithm to decide which instance will be contacted.

Server-side

To contact a service, clients make a request to the service's load balancer which runs at a well known location. This load balancer queries the registry and forwards the request to an available instance.

In any case, the service registry is a critical component and thus, must be highly available.

2.1.5. API Gateway Pattern

Depending on the decomposition technique chosen (see subsection 2.1.3), microservices might provide very fine-grained APIs. In turn, this means that clients may need to interact with multiple different services to carry out a high-level business process. To hide this complexity from clients and ensure consistent behavior, a pattern known as the API gateway is employed.

An API gateway represents the single entrypoint for all clients in which some requests are simply proxied while others fan out to and consume multiple services. In the latter case, gateways can be viewed as orchestrators and as such, typically do not have persistence layers [13, p. 585]. They may, however, cache responses. Another important task in orchestrating multiple microservices is managing distributed transactions [3], i.e. ensuring atomicity guarantees for a set of distributed resources [2, p. 32]. Finally, gateways may also deal with generic features such as authentication and authorization or implement the circuit breaker pattern to prevent a service failure from cascading to other services [5, p. 41] [7, p. 37].

add cita-
tion

[2] In the event of a network partition, a standard recovery process would attempt to restart a service in the healthy partition, thus leading to a new network location for this service.

[3] Distributed transactions are commonly implemented using the two-phase commit protocol. The no-ACID transaction type has also been proposed for this context, which is known as a compensation transaction [2, p. 32].

It shall be noted that gateways incur a performance penalty because they introduce an additional network hop [7, p. 37]. This is generally true for proxying gateways. Orchestrating gateways, on the other hand, have the potential to decrease latency since the various requests being collapsed now already originate from the target network. In both cases, the performance degradation will heavily depend on the system's interconnectedness [4, p. 9].

add cita
tion

2.1.6. Database-per-Service Pattern

To keep microservices loosely coupled, a pattern known as database-per-service is employed. This pattern calls for each microservice to have its own database, compared to sharing one across multiple services. Sharing is achieved by making the data accessible via the service's API [7, p. 36] [9, p. 59]. Messina et al. discern between three levels of pattern conformity [7, p. 37]:

Private tables
Each service has a set of tables private to that service.

Private schema
Each service has a database schema private to that service.

Private database
Each service has its own database.

While this pattern contributes to service intimacy, it comes at the cost of having to redefine data models and restate business rules across services [4] [2, p. 30].

2.1.7. Delineation from Service-Oriented Architecture

Historically, the complexity of monolithic applications (see subsection 2.1.1) has already been addressed using different Service-Oriented Architecture (SOA) approaches that also decompose a large system into many smaller services. Academia, however, is undecided whether microservices should be considered as a subset or superset of SOAs or whether it constitutes a new, distinct idea [13, pp. 584–585] [2, p. 30]. The systematic mapping study conducted by Cerny, Donahoo, and Trnka in [2] spends a great deal on contrasting the two architectures. A short summary is given in the following.

[4] In Domain-Driven Design, the concept of a Bounded Context describes that services operate with business objects in a specific context and therefore, only need to model a subset of the global object's attributes [2, p. 30].

In both approaches, services cooperate to provide functionality for the overall system. However, the path to achieving this goal is different. This is most obvious when looking at the interaction patterns between the services involved. SOAs rely on orchestration, whereas microservice-based applications prefer choreography. The former expects a centralized business process to coordinate activities across services and combine the outcomes, whereas the latter expects individual services to collaborate based on their interface contracts. Orchestration differs from choreography with respect to where the logic that controls the interactions should reside. The two terms describe a centralized and decentralized approach hereof, respectively.

An important remark that Cerny, Donahoo, and Trnka make, is that orchestration through an integration layer, such as a messaging bus, oftentimes leads to a situation in which the system parties, i.e. the services, agree on a standardized representation of the business objects they exchange. The system ends up with one kind of business object each. This is known as a canonical data modal. The danger being that a change in one of the business objects necessitates changes in all of the services that deal with this object. As a result, deployments in a SOA again happen in a monolithic fashion. In the microservice architecture, such a change can at most propagate to the API gateways (comp. subsection 2.1.5), though this is less likely because gateways integrate services based on their interfaces, not on their models.

Lastly, albeit obvious, the biggest drawback of SOAs and their centralized orchestration model must be stated. Having a centralized business process integrate all services again entails having a single point of failure.

2.1.8. Operational Benefits and Challenges

2.2. Virtualization

2.2.1. Hypervisor-based

2.2.2. Container-based

2.3. Container

2.3.1. LXC

2.3.2. Docker

2.4. Container Orchestration

2.5. Kubernetes

2.5.1. Overview

2.5.2. Cluster Architecture

2.5.3. Containers

2.5.4. Workloads

2.5.5. Services, Load Balancing and Networking

2.5.6. Storage

2.5.7. Configuration

2.5.8. Security

2.5.9. Policies

2.5.10. Scheduling, Preemption and Eviction

2.6. Agile Software Development

3. Concept

3.1. Overview

3.2. Functional Requirements

3.3. Non-Functional Requirements

4. Architecture

4.1. Component Design

4.1.1. Hardware

4.1.2. Backend

4.1.3. Maintenance and Support Plan

4.2. Component Interaction

4.3. Component Mapping

4.4. Design Rationale

4.4.1. Modeling as Cloud-Edge Computing Problem

5. Implementation

6. Testing

7. Conclusion

8. Summary

9. Outlook

Bibliography

[1] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. "Microservices architecture enables devops: Migration to a cloud-native architecture". In: *Ieee Software* 33.3 (2016), pp. 42–52.

[2] Tomas Cerny, Michael J Donahoo, and Michal Trnka. "Contextual understanding of microservice architecture: current and future directions". In: *ACM SIGAPP Applied Computing Review* 17.4 (2018), pp. 29–45.

[3] Namiot Dmitry and Sneps-Sneppe Manfred. "On micro-services architecture". In: *International Journal of Open Information Technologies* 2.9 (2014).

[4] Nicola Dragoni et al. "Microservices: yesterday, today, and tomorrow". In: *Present and ulterior software engineering* (2017), pp. 195–216.

[5] Miika Kalske, Niko Mäkitalo, and Tommi Mikkonen. "Challenges when moving from monolith to microservice architecture". In: *International Conference on Web Engineering*. Springer. 2017, pp. 32–47.

[6] Asif Khan. "Key characteristics of a container orchestration platform to enable a modern application". In: *IEEE cloud Computing* 4.5 (2017), pp. 42–48.

[7] Antonio Messina et al. "A simplified database pattern for the microservice architecture". In: *The Eighth International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA)*. 2016, pp. 35–40.

[8] Vindeep Singh and Sateesh K Peddoju. "Container-based microservice architecture for cloud applications". In: *2017 International Conference on Computing, Communication and Automation (ICCCA)*. IEEE. 2017, pp. 847–852.

[9] Davide Taibi and Valentina Lenarduzzi. "On the definition of microservice bad smells". In: *IEEE software* 35.3 (2018), pp. 56–62.

[10] Johannes Thönes. "Microservices". In: *IEEE software* 32.1 (2015), pp. 116–116.

[11] Leila Abdollahi Vayghan et al. "Deploying microservice based applications with kubernetes: Experiments and lessons learned". In: *2018 IEEE 11th international conference on cloud computing (CLOUD)*. IEEE. 2018, pp. 970–973.

[12] Leila Abdollahi Vayghan et al. "Microservice based architecture: Towards high-availability for stateful applications with Kubernetes". In: *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE. 2019, pp. 176–185.

[13] Mario Villamizar et al. “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud”. In: *2015 10th Computing Colombian Conference (10CCC)*. IEEE. 2015, pp. 583–590.

Glossary

scale cube

Describes a three-dimensional model for scaling an application. X-axis scaling refers to running multiple instances of an application behind a load balancer. Y-axis scaling splits an application into multiple, distinct services. Z-axis scaling partitions the data to be processed across a set of application instances [7, p. 36].

Appendices

A. Concept

B. Architecture

B.1. Representational State Transfer (REST)

B.2. Certificate-based Authentication

C. Conclusion