MASTER'S THESIS

The implementation of microservices architectural patterns in open-source Java projects

Braber den, N.J. (Nico)

Award date:
2021

The implementation of Microservices Architectural patterns in Open-Source Java projects

by

N.J. den Braber

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University, Faculty of Management, Science and Technology
Master Software Engineering
to be defended publicly on Thursday, October 14, 2021 at 1:00 PM.

Student number:
Course code: IM9906
Thesis committee: dr. ir. Alaaeddin Swidan (chairman), Open University
dr. ir. Fenia Aivaloglou (2nd supervisor), Open University

October 5, 2021

Content

List of figures

List of tables

1 Summary

The architecture of software applications based on microservices can use architectural patterns to guide their development and deployment. Microservices are small independent software modules that communicate with each other, external applications, or end-users and manage their own data. Architectural patterns describe proven solutions for the problems that software architectures, including those based on microservices, will address: responsiveness, availability (7x24), and scalability to process a high traffic demand.
This thesis explores how architectural patterns are implemented in Open-Source microservices projects. As a first step (Phase 1), we used existing studies on architectural patterns for architectures based on microservices and identified seven patterns, which we described. We made a restriction on patterns related to the communication and data-layer to limit the research area. As a second step (Phase 2), we identified twenty Open-Source projects that use microservices.
We manually investigated ten projects and created software programs for a static code analysis tool. Based on the software quality tool JQAssistant combined with Neo4J, this static code analysis tool can detect architectural patterns in these Open-Source projects.
We found that the implementations of microservices are based on a few core frameworks like Spring and Microprofile. In addition, the implementations of some patterns result from the effort of a few high-tech companies like Netflix.
We used the ten other Open-Source projects to validate the results of the static code analysis tool. We found that there is a significant variation for implementing patterns in Open-Source projects. Patterns implemented with the frameworks mentioned above can easily be detected if they use predefined annotations.

We believe that the software produced for this thesis is very useful for detecting and visualizing the architectural patterns and validating constraints created for these patterns.
JQAssistant is a code analyzer tool, and Neo4J is a Graph database, which can execute queries based on the Cypher language. Graph databases are very suitable for efficiently displaying relationships between software concepts such as files, packages, classes, and methods.

We recommend that the Java language or additional frameworks are enriched with standard annotations which implement MSA patterns. Patterns implemented with project-specific source code or configuration files are much harder to detect by the static code analysis tool.
If this is true for such tooling, then it is likely the case for (junior) developers who need to understand or implement the MSA patterns in their projects.
In addition, JQAssistant and Neo4J can also be easily integrated into the software development process, which guides developers in creating the correct implementation of a pattern in their code based on detailed rules and constraints that more experienced developers have determined for a specific pattern.
Another area for future research is using a Graph database to query many software projects for rules and constraints.

2 Introduction

Many organizations such as banks, insurers, social media companies, and government institutions face higher demands on their IT systems' availability and performance [51]. The use of online services still increases; users of the internet (e.g. for webshop) like to manage their business at any time of the day and in different ways, such as mobile phones, tablets, or PCs. A website or app's responsiveness is crucial; otherwise, people will choose a competitor or lose confidence in the service provider.

Jonas Bonér, co-founder of IT company Lightbend together with Dave Farley, Roland Kuhn, and Martin Thompson, understood that IT systems need to be designed in such a way that they can cope with these ever-growing demands for availability and responsiveness. So they wrote a new "reactive manifesto" to express the requirements that today's IT systems must meet[1]. The manifesto has four key-level characteristics:

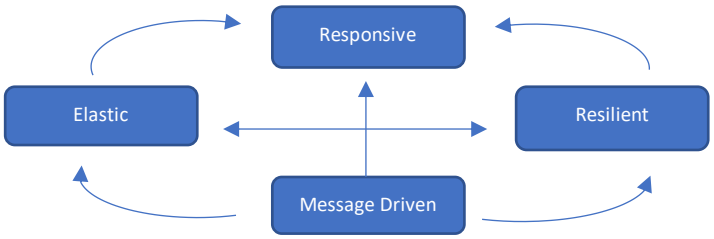


Figure 1 - Reactive Manifesto

- **Responsive:** This is the cornerstone. It is required to react to user demands in a timely manner.
- **Resilient:** A system must stay responsive, also in case of failures.
- **Elastic:** A system must react by adding or removing resources when load increases or decreases,
- **Message-Driven:** Non-blocking communication will lead to less overhead and loose coupling.

Organizations are moving to private and public cloud infrastructures that embrace the reactive manifesto requirements [32]. A cloud service will offer customers hardware and software resources in private (not shared) or public (shared with others) environments. These infrastructures are often implemented as a container platform managed by software such as Docker [33][2]. A Docker image is an application that runs stand-alone with its dependencies in a Docker virtual container. These containers are very lightweight and can run on every Linux server or virtual machine.

Microservices are very suitable for running in containers. Microservices have become popular because well-known companies such as Amazon, LinkedIn, Netflix and Spotify embrace them [1]. Fowler and Lewis provide the most well-known definition of microservices [2]:
"In short, 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. These services are built around business capabilities and are independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."

[1] https://www.reactivemanifesto.org/
[2] https://www.docker.com/

Microservices have drawn more attention from the research community recently. Many studies have been published on the usage of architectural patterns in microservices-based systems [1, 3, 4]. These studies show that microservices patterns are very diverse, e.g., a high-level pattern can describe a specific architectural style, like the traditional Monolithic architecture (e.g., 3-layer MVC Java enterprise application), a service-oriented architecture (SOA) or a microservices architecture (MSA). Other, more fine-grained patterns describe deployment (DevOps), IoT (Internet of Things) or communication solutions. This research focuses on the usage of architectural patterns in Open-Source microservices projects. We describe how standard solutions (patterns) implement the requirements related to the reactive manifesto, both from a theoretical (literature review) and a practical side (Open-Source implementations).

2.1 Research questions

The thesis's primary goal is to investigate the implementation of architectural patterns in Open-Source (OS) software projects that use microservices. As mentioned above, there are several architectural patterns on different levels of the application landscape. To limit the thesis's scope, we will focus on architectural patterns used for the communication between microservices and those used in the data-layer of a microservice. Running in its own process and communicating with lightweight mechanisms are essential aspects of microservices [2]. Therefore, our focus area covers a broad aspect of microservices' architecture style at the application architecture level.

In this thesis, we aim to answer the following research questions:

Theoretical questions (Phase 1):

1. Which are the common architectural patterns identified in the literature that are part of a microservices architecture communication and data-layers (RQ1)?
2. How are these architectural patterns defined in literature? What is the extent of variations in the definitions (RQ2)?

Practical question (Phase 2):

3. What are the most common microservices communication and data-layer architectural patterns in OS software projects? What are the characteristics of these implemented patterns (RQ3)?

2.2 Research phases

The thesis will be split into two phases to achieve the goal mentioned above. During Phase 1, what we call the theoretical phase, we will investigate architectural patterns via a literature study and describe a selection of seven patterns used for the communication between microservices and the data-layer of a microservice. The sources found in the literature study are the input for a detailed description of the selected patterns. Phase 2, what we call the practical phase, is based on a manual examination of the architecture patterns selected in Phase 1 and used in various OS Java projects that use microservices. We selected twenty OS Java projects in GitHub and ten of them will be investigated manually. The implementation of the patterns of Phase 1 found in these ten OS projects will be used to develop a static code analysis tool. Although analyzing software is not trivial, we have designed queries that can be used to detect the seven MSA patterns. A software project's source code is parsed to nodes and relations (edges) between the nodes, to create a Graph model [51] to achieve this goal. We will use the existing program JQAssistant[3] , which can parse Java code keywords (like class, annotation). That program will be extended with specific information to detect architectural patterns. The other ten OS projects will be used to validate the static code analysis tool to see if the seven patterns will automatically be detected in these projects.

2.3 Contributions

This thesis contains a detailed literature study about architectural patterns in microservices architecture communication and data-layers. However, the literature sources found during the literature study have limited descriptions of such patterns; therefore, we will elaborate in this thesis on these MSA patterns' characteristics and describe them in Chapter 6 via the pattern layout.
To detect such patterns in Open-Source software projects, we developed queries described in Chapter 9, which can be used to query a large set of Open-Source programs for MSA patterns. The usage of Graph databases fits well for analyzing software programs. It is also possible to query one specific implementation of an MSA pattern. Adding detailed rules (concepts and constraints) for a specific implementation will guide developers to correctly implementing such a pattern [53].

2.4 Outline of the thesis

In the next chapter, Research Background (Chapter 3), we first describe what is understood by microservices and architectural patterns. The research we perform in this thesis for microservices and architectural patterns is organized in a theoretical Phase 1 and a practical Phase 2.
The theoretical Phase 1 starts with the research questions and the research method (Chapter 4) and will then describe the results of the literature study (Chapter 5) and the detailed descriptions of the selected MSA patterns (Chapter 6). Chapter 5 is the result for answering research question RQ1 and Chapter 6 is the result for answering research question RQ2.
The practical Phase 2 starts with a description of the research question and research method (Chapter 7) and will then describe the implementation of MSA patterns in Open-Source projects (Chapter 8). These implementations are the basis for a static code analysis tool for MSA patterns (Chapter 9). Chapter 10 will present an overview of pattern implementations as an answer to research question RQ3. Chapter 11 contains a discussion on the results of Phase 1 and Phase 2. Finally, the thesis will end with conclusions and recommendations for future work (Chapter 12).

[3] https://jqassistant.org/

3 Research background and related work

We first discuss the background of microservices architecture, the reasons behind its emergence, and advantages and disadvantages of other architectural styles. Following, we explain the usage of a template for architectural patterns, which will be used to describe patterns found in academic and industrial papers. Finally, the last two subchapters describe popular frameworks supporting microservices and the usage of Open-Source software repositories.

3.1 Evolution to the microservices architecture

Microservices are independently deployable services that communicate with each other via a lightweight protocol [2]. A microservice is built around a business capability (e.g. consumer, order) and manages the data that applies to that business capability. Microservices architecture is seen as a follow-up to the SOA architecture. The SOA architecture is considered a follow-up to the monolithic architecture described by Jamshidi et al. [34].
Figure 2 shows the evolution of the software architecture [3]:

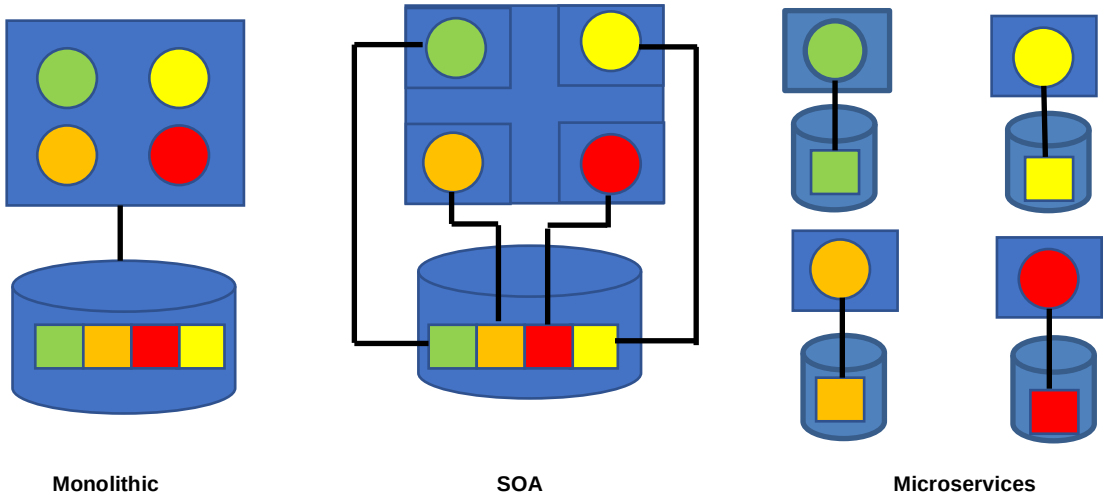


Figure 2 - Evolution of application architecture [3]:

An application has different functionalities with a monolithic architecture, represented in Figure 2 by colored circles (functions) and squares (data). However, these functionalities are not independent. The application must be redeployed entirely when a function is adjusted. The used technology is often limited to one programming language, like Java or C#, and one runtime environment like a JEE server or Microsoft server with a specific database like Oracle or MS SQL.

With an SOA architecture of an application (landscape), the functions (services) with data are managed separately. Therefore, it is possible to deploy a new or updated service independent of other services.
However, an SOA architecture still uses many infrastructure (middleware) components (webserver, database, message bus) for all the services. Therefore, this dependency has an impact on all the services when one of those components fails.

Jamshidi et al. [34] compared the SOA and microservices architecture styles. When it comes to differences, they identify that SOA uses well-defined webservice protocol standards (WS-*, SOAP/XML), which are seen as complicated, while microservices use lightweight protocols such as

REST (Representational State Transfer) and HTTP, which gives more freedom in the chosen format (JSON, XML, etc.). On the other hand, Jamshidi et al. identify that SOA and microservices also have many similarities. For example, both use the term services, and in both cases, the development team is responsible for one service. However, while the same terms are used, they still differ in the details. In a microservices architecture, functions (services) and data are strictly separated. This separation gives extra possibilities to implement services with other technologies and "scale" services independently, depending on the service's load. Teams embracing microservices also embrace approaches like DevOps and Continuous Delivery and Integration (CD/CI). As a result, they are more aware of the runtime environment and the problems that may occur. Applications are built with design for failures and infrastructure automation (installation, scaling).

The essential advantages of microservices are faster delivery (CI/CD) of smaller functionality, improved fine-grained scalability, and greater autonomy [2]. This autonomy is not only related to the runtime environment but also development teams. Each microservice (team) chooses what best fits their experience with technology and program language. As Figure 2 shows, the microservices are independent of each other, and they each manage their data. This independence is of great importance to our thesis as we explore how communication and data-layers are implemented.
The challenge in using microservices is to define the scope and size of a microservice. Another concern is that more and more common functions (discovery, fault tolerance traffic management) are moved to the underlying platform, increasing dependency on such platforms, one of SOA architectures' disadvantages [34].

3.2 Definition of an architectural pattern

Software architecture is defined according to the ISO/IEC/IEEE standard 42010 as “the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution” [35]. A decision to choose a specific architecture is based on the desired quality attributes [36]. This decision can be based on experience with architectural choices in previous systems. Such architectural experience is recorded in architectural patterns [37].

Definition of an architectural pattern:
In software development, the application of patterns has become known through the Gang of Four book [39]. A pattern describes a general design that is used to solve a recurring problem [7, 38].
An architectural pattern is applied at the architecture level and is a proven structural organization schema for software systems. Patterns are described using a pattern template or pattern schema. All of the many different templates have at least the components as shown in Table 1 [7, 38]:

Context	A typical situation giving rise to a problem. E.g. an online store with many consumers who search for products and place orders.
Problem	In the context described above, a recurring problem occurs. The problem is related to quality attributes, like availability, performance, or usability on an architectural level. E.g. if many consumers visit an online store simultaneously, the responsiveness of the website degrades.
Solution	A proven solution for the problem is given as a structure with components and relationships and a description of how they collaborate. A solution should solve the problem and should also consider the constraints. Quality attributes may conflict with each other (performance may conflict with extensibility, for instance). E.g. a solution could describe the usage of a load balancer to increase performance. The load balancer distributes the traffic across multiple servers.
Constraints	A description of what an architectural pattern does not enable or where it has a negative effect.
Generic structure	This will be an architectural diagram explaining the architectural pattern. It will show microservices and their communication and the relation between each other.

Table 1 - Standard components of an architectural pattern

3.3 Open-Source frameworks supporting microservices

Software applications that embrace the MSA architecture are developed with the help of commercial and Open-Source frameworks. Spring[4] [3] has multiple frameworks for supporting microservices. Spring Boot supports the development and deployment of stand-alone Java applications, which is very suitable for microservices. Spring Cloud offers a wide range of features to support the applications in a container environment. Other Spring frameworks support messaging and integration. Table 2 lists the frameworks supported by Spring. We will not provide all details of these frameworks, but Spring's documentation offers a good entry point for more detailed information.

Name	Description
Spring Framework	This is the core framework of Spring, with support for webservices, transaction management, data access and caching.
Spring Boot	Spring Boot enables easy development and deployment of stand-alone applications.
Spring Cloud	This framework has support for Java applications running in a distributed environment (containers). Examples of supported features are configuration, service discovery, circuit breaker and intelligent routing and messaging. Source code donated by Netflix is also part of this framework.
Spring Data	This framework supports a wide range of database implementations like traditional rational databases, non-relational databases, big data, and cloud features (e.g. map-reduce).
Spring Integration	This framework has support for gateways and messaging.
Spring AMQP	This has support for AMQP-based messaging (like RabbitMQ).
Spring Apache Kafka	This has support for Kafka-based messaging.

Table 2 - Overview of Spring frameworks

The second framework we will highlight is Eclipse Microprofile[5] [52]. The industry behind Java EE (Oracle, Red Hat, IBM) has initially been the monolithic application development vendor. They understood that they should embrace the ideas behind microservices and started supporting the Open-Source Microprofile framework as part of the Eclipse Jakarta EE suite. Table 3 shows the features of Microprofile:

Name	Description
Open Tracing	Distributed tracing of calls across multiple microservices.
Open API	Document REST APIs and build client code from it (Swagger).
Rest Client	JAX-RS client.
Config	Configuration at a central place external from the microservice.
Fault Tolerance	Dealing with failures, e.g. a circuit breaker.
Metrics	Data from applications and the java virtual machine.
JWT Propagation	JSON Web Token authentication and authorization.
Health check	Keeps track of the health of microservices.
CDI	Context and Dependency Injection
JSON-P	APIs to parse, generate, transform, and query JSON documents
JAX-RS	Java REST implementation
JSON-B	Convert Java objects to/from JSON messages

Table 3 - Overview of Microprofile framework

[4] https://spring.io/projects
[5] https://microprofile.io/

We will highlight two other frameworks, that provide support for microservices [52]. The first is the Open-Source Eventuate framework developed by Chris Richardson, author of Microservices Patterns' popular book [41]. The second framework is the Open-Source Axon framework developed by AxonIQ, a company headquartered in the Netherlands [3]. These two frameworks support the concepts shown in Table 4[6]:

Name	Description
DDD	Domain-Driven Design is an approach to develop a business domain model and was initially described by Eric Evans.
Event sourcing	Pattern for data storage. Instead of storing the current state of an entity, all changes to the state are stored.
CQRS[7]	Command Query Responsibility Segregation supports a different model for writing information (commands) and reading the information (queries).

Table 4 - Event driven architectural concepts

3.4 The usage of Open-Source repositories in software engineering research

The thesis's primary goal is to investigate the implementation of architectural patterns in Open-Source software projects designed following the microservices architecture style.
This investigation requires software repositories that are available for analysis. Analyzing software repositories is usually referred to as “mining software repositories (MSR)” [48]. MSR extracts data from repositories, processing the data to information, and analyzing the required knowledge. Software repositories can be used to analyze several aspects of software development and evolution, such as bugs, changes, source code, and team interaction.

We will use Open-Source projects on GitHub. GitHub is an open and closed source platform in which software projects are stored in a version control system of git. GitHub provides features that support the social interaction between developers and allow them to collaborate. In addition, other features support the monitoring of the development process and sharing knowledge. GitHub is the leading software development platform with a community of 40 million people and hosting 100 million projects (as mentioned on the webpage). Besides GitHub, there are also other software development platforms such as Bitbucket and SourceForge. Furthermore, GitHub is often used as a source for mining software repository studies to understand and support software development practices [49]. In 2018, it was announced that Microsoft would acquire GitHub, leading to the creation of an alternative platform named GitLab.
We will only investigate projects with source code, written in Java because of this programming language's available knowledge and its popularity in GitHub [46, 47]. Therefore, we will only look at the last committed source code for the thesis, including build and configuration scripts.
More details about the selection of the projects and the nature of the projects can be found in Subchapter 7.2.1.

[6] https://axoniq.io/resources/architectural-concepts
[7] https://martinfowler.com/bliki/CQRS.html

4 Research Phase 1 (Theoretical)

This chapter describes the thesis's theoretical phase and starts by highlighting the research questions related to this phase in Subchapter 4.1. Then, the used research method by means of a literature study is explained in Subchapter 4.2. The outcome of the literature study is described in Chapter 5, and the detailed descriptions of the selected architectural patterns are described in Chapter 6.

4.1 Research questions (RQ1 and RQ2)

The thesis's primary goal is to investigate the implementation of architectural patterns in Open-Source (OS) projects that use microservices.
The research is limited to patterns in the field of:
- Communication between microservices
- Data-layer of microservices

Before we are able to do this investigation, we first need to identify the architectural patterns that are popular within the MSA communication and data-layer.
We aim to answer the following research questions as a part of Phase 1:

1. Which are the common architectural patterns identified in literature that are part of a microservices architecture communication and data-layers (RQ1)?
2. How are these architectural patterns defined in literature? What is the extent of variations in the definitions (RQ2)?

4.2 Research method (RQ1 and RQ2)

To answer the research questions of Phase 1, we will perform a literature study containing two steps. The first step is a literature scan, identifying articles describing MSA patterns in the literature. In this literature scan, we searched Google Scholar for scientific articles published between 2016 and 2019. using the terms: "architectural", "micro service or microservi or micro-servi", "pattern" and "design" [3].
For the resulting articles, we set the following inclusion criteria:
- It is written in English
- Also execute a literature study on MSA patterns
- Only one per author (or same group of authors)
- Mention multiple MSA patterns (more than three)
- Do not describe the migration of a monolithic architecture to a microservices architecture
- Related to architecture (not only on design)
- Not only related to security
- Have multiple citations in other articles (more than three)
- Do not use studies on patterns described before 2016

This step generates a set of papers, which will be used in the second step. The second step investigates the patterns in dept and excludes patterns not relevant to our research questions. We create a grouping of the architectural patterns to select only those related to communication between microservices and the microservices' data-layer. As mentioned in the introduction, we have to limit many architectural patterns to narrow the research area and make it suitable for the time period that stands for the thesis. This limitation means that we need to exclude patterns that are related to:
- SOA architecture

- Migration to MSA
- Deployment of applications
- Management of applications in a (production) environment
- Internet Of Things
- Testing
- Security
- User Interface
- Design and not architecture

The thesis is limited to specific microservices architecture patterns. Subchapter 3.1 mentions that SOA and microservices architecture have the necessary similarities, and several patterns can therefore be applied to both architectures [1, 3]. We only select patterns that were introduced or largely adjusted because of MSA.
We also combine more fine-grained patterns in a high-level pattern to limit the number of patterns. For the same reason, we combine architecture patterns with different names but are variations of the same pattern.
The patterns are described in the architectural pattern format as listed in Subchapter 3.2. We describe the constraints that are caused by an architectural pattern, e.g. impact on performance. We create a simple picture per architectural pattern for each pattern (see generic structure) without using a formal modeling language like UML or Archimate. The pictures are straightforward, and using a formal notation would not add extra information. The patterns together with a simple picture are described in Chapter 6.
Variations of the same pattern are recorded as the answer to research question 2. We decide on keywords describing the patterns, which can be used in Phase 2 to find patterns in source code.
To describe this extra information, we extend the description for architectural patterns with the following information:

Variations	One of the thesis's main topics is investigating the differences between the various literature descriptions of an architectural pattern. We record that under variations.
Keywords	The analysis of the architectural patterns' implementations will be based on searching for specific pattern keywords. Therefore, we record that under keywords.

Table 5 - Extra components for a MSA pattern

5 Literature study (RQ1)

As described above, under the research method, we first execute a literature scan to retrieve papers containing several MSA patterns. This set of papers will be used in step two to filter the relevant MSA patterns to answer our research questions of Phase 1.
The result of the literature study is a selection of MSA patterns, which answers RQ1.

5.1 Selection of research sources

For the literature scan, we searched Google Scholar for scientific articles published between 2016 and 2019, using the terms: "architectural", “micro service or microservi or micro-servi”, “pattern” and “design” [3]. The result (36 papers) of the literature scan is available in Appendix A.
We use the selection criteria for papers of step 1 (see 4.2) of the literature study to retrieve a set of articles. As mentioned in the introduction, several existing literature studies about microservices related to architecture and architectural patterns exist. We encountered three existing studies [1, 3, 4] with the same research area (microservices and architectural patterns) as this thesis.
We also decided to include Messina et al. [5]; although they did not have done a literature study, they did elaborate on several microservices patterns and paid much attention to the database layer patterns. In addition, we use the microservices.io website [6] of Richardson with a collection of 48 microservices patterns as a reference from the industry. This website was mentioned by all the four papers mentioned above. The website is also mentioned several times in other articles, as seen by an extra remark after the article's title in Appendix A.
We only use one paper of these selected authors (group), excluding 14 papers. We exclude those papers that are not written in English (1x), describe only a few patterns (14x), are related to migration, design or security (4x), have only a few citations (11x), describe the usage of patterns before 2016 (1x).
This first step results in four academic papers [1, 3, 4, 5] and one website [6] and they are the basis for the next step.

5.2 Categorization of the selected sources

This subchapter describes the second step of the literature study. First, we describe per selected source (four papers and one website) which categories (grouping) and architectural patterns were found. Then, we use these categories to find the patterns related to the categories communication and data-layer. As mentioned before, the focus of this thesis will be on communication and data-layer patterns. Our categories communication and data-layer are directly related to the two main components (functionality and data) of a microservice mentioned in 3.1.
We use the exclusion criteria for patterns mentioned in step 2 of 4.2 to filter those patterns in the two categories, which are not applicable for our research.

Francesco et al. [1] created a framework for classifying, comparing, and evaluating architectural solutions, methods, and techniques specific for microservices and selected 103 academic and industrial papers to create an overview of publications of architecting with microservices. Table 6 shows eight architectural patterns found by Francesco et al. as a result of their research. Although they did not have defined categories, they are all related to communication between microservices or between microservices and an external application. In addition, Francesco et al. created a list of used sources for each pattern (see Appendix B), which we use for our pattern descriptions in Chapter 6.

Category	Architectural patterns
Communication	API gateway; publish/subscribe; proxy; circuit breaker; discovery patterns; load balancer; service registry; service bus

Table 6 - Patterns found by Francesco et al. [1]

We exclude the service bus already used in SOA and not in MSA because it is replaced by a lightweight message bus [9]. We exclude load balancer because this is not a new MSA pattern and is already used in SOA [3]. We will combine discovery patterns and service registry because the discovery patterns always use a service registry [4]. We will include the Proxy pattern in the API Gateway pattern because it is one of the API Gateway functions [10]. We use the name Messaging pattern instead of Publish/Subscribe pattern to address also other asynchronous communication [6]. Francesco et al. found more patterns, as shown in Table 7, but they mentioned that only one study addressed such patterns. We will not include these patterns for that reason, except for the data-layer (database is the service), because we encountered four references (see Appendix B) and will use this pattern under the name database per service pattern.

Category	Architectural patterns
Loose coupling	Location independence; communication independence; security independence; instance independence
Hexagonal architecture	Ports and adapters
CI/CD	Immutable server
Data provision	Data adapter
Fault recovery	Bulkhead
Cloud	Twelve-factor and cloud computing
Data-layer	The database-is-the-service

Table 7 - Additional patterns mentioned by Francesco et al. [1]

Table 6 results in the following list of selected patterns for Francesco et al.: API gateway (including proxy), messaging (including publish/subscribe), circuit breaker, service discovery (including service registry) and database per service. We exclude load balancer and service bus as explained above.

Márquez et al. [3] created a catalog of microservices patterns reported in 69 academic and industrial papers. They mentioned 17 architectural patterns. Márquez et al. defined 11 categories but did not find patterns for all of them, as shown in Table 8.

Category	Microservices architectural patterns
IoT	None
DevOps	None
Front-end	None
Back-end	**Result cache; page cache; scalable store;** key-value store
Orchestration	**Container; enable cont. integration; API gateway**; load balancer
Migration	Service discovery
Communication	**API gateway; log aggregator;** service registry; messaging
Behavior	None
Design	None
Mitigation	Health check; monitor
Deployment	**Database is the service; backend for frontEnd;** circuit breaker

Table 8 - Categories and patterns found by Márquez et al. [3]

Márquez et al. mention that many patterns are also related to SOA and that only 10 of the 17 described patterns are really MSA patterns according to them. The categories back-end, orchestration, communication, and deployment contain these MSA patterns, and these patterns are shown in bold in Table 8. For our communication category, we will select the patterns in the orchestration and communication categories in Table 8. For our data-layer, we select the patterns in the back-end category of Table 8. API gateway will include the backend for frontend pattern as a variation of this pattern [6]. We disagree that it is a deployment pattern. We agree that some

patterns (e.g. load balancer) are already part of SOA and can be excluded. However, we disagree with Márquez et al. that existing patterns like service discovery, service registry, messaging and circuitbreaker do not require a different MSA implementation. We will combine result cache and page cache in a general caching pattern for the data-layer category. We combine scalable store with the database is the service in the more general database per service pattern.
We do not directly relate the container and enable continuous integration and log aggregator patterns to our two categories and exclude them.
This results in the following list of selected patterns from Márquez et al.: API gateway (including backend for frontend), messaging, circuit breaker, service discovery (including service registry), caching (result cache and page cache) and database per service (scalable store and database is the service).

Taibi et al. [4] selected 42 papers and created a pattern catalog for seven architectural patterns. Taibi et al. mentioned categories and patterns related to communication and data-layer, except for the deployment strategies & patterns, as shown in Table 9.

Category	Microservices architectural patterns
Orchestration and Coordination - Composition	API Gateway;
Orchestration and Coordination – Service discovery	Client-side discovery; Server-side discovery;
Hybrid	API Gateway and Service Registry;
Deployment Strategies & Patterns	Multiple services per Host; Single service per Host;
Data Storage	Database per service; Database Cluster; Shared Database Server

Table 9 - Categories and patterns found by Taibi et al. [4]

We will combine client-side discovery, server-side discovery and service registry in one pattern service discovery. We will combine the data storage patterns in one pattern called database per service.
This results in the following list of selected patterns from Taibi et al.: API gateway, service discovery, database per service.

Messina et al. [5] did not mention categories but named an area where the pattern relates. In our opinion, they are all related to communication and data-layer, as can be seen in Table 10. We will here also combine client-side discovery, server-side discovery and service registry in one pattern service discovery. We will combine the data-layer and 'client access and data-layer' in one database per service pattern.

Area	Microservices architectural patterns
Client access	API Gateway
Routing	Client-side and server-side discovery
Location	Service Registry
Data-layer	Database per Service
Client access and data-layer	Database-is-the-Service

Table 10 - Categories and patterns found by Messina et al. [5]

This results in the following list of selected patterns from Messina et al.: API gateway, service discovery, database per service.

Richardson [6] has an enormous list of patterns, all grouped in categories, as shown in Table 11. We think that communication style, external API, service discovery and reliability are related to communication and data management is related to data-layer. We will pick the database per service pattern from the data management category and include the shared database pattern for this data-layer category. We will also add CQRS (together with Event sourcing) as an important new pattern for our data-layer category. CQRS will split the database into a database for writing and a database for reading. The other patterns in this category are, in our opinion, on a lower (design) level, like the saga [52], API composition and domain event.
We will pick from the communication style the general messaging pattern (including remote procedure invocation) for our communication category. From external API, we will pick API gateway in combination with Backend for frontend. We will also include the Service discovery category patterns combined in one pattern with the same name for our communication category. For Reliability, we will add the Circuit breaker pattern. The domain-specific protocol and idempotent consumer patterns of the communication style are, in our opinion, on a lower (design) level and are therefore excluded.

Category	Microservices architectural patterns
Decomposition	Decompose by business capability; Decompose by subdomain; Self-contained Service; Service per team
Refactoring to microservices	Strangler Application; Anti-corruption layer
Data management	Database per Service; Shared database; Saga; API Composition; CQRS; Domain event; Event sourcing
Transactional messaging	Transactional outbox; Transaction log tailing; Polling publisher
Testing	Service Component Test; Consumer-driven contract test; Consumer-side contract test
Deployment patterns	Multiple service instances per host; Service instance per host; Service instance per VM; Service instance per Container; Serverless deployment; Service deployment platform
Cross-cutting concerns	Microservice chassis; Externalized configuration
Communication style	Remote Procedure Invocation; Messaging; Domain-specific protocol; Idempotent Consumer
External API	API gateway; Backend for frontend
Service discovery	Client-side discovery; Server-side discovery; Service registry; Self-registration; 3rd party registration
Reliability	Circuit Breaker
Security	Access Token
Observability	Log aggregation; Application metrics; Audit logging; Distributed tracing; Exception tracking; Health check API; Log deployments and changes
UI patterns	Server-side page fragment composition; Client-side UI composition

Table 11 - Categories and patterns found by Richardson [6]

This results in the following list of selected patterns from Richardson: API gateway (including backend for frontend), messaging (including remote procedure invocation), circuit breaker, service discovery, database per service (including shared database), CQRS (including event sourcing).

5.3 Overview MSA patterns in the selected sources

The five selected sources' literature study results in four architectural patterns for the communication category and three for the data-layer category, as shown in Table 12.

Category	Architectural pattern	Found?	Remarks
Communi-cation	API gateway	100%	Mentioned in all sources [1, 3, 4, 5, 6]. Different names as aggregator, proxy, backend for frontend are used.
	Messaging	60%	Mentioned in three sources [1, 3, 6]. The term publish/subscribe is also used.
	Circuit breaker	60%	Design for failures in communication between microservice. Mentioned in three sources [1, 3, 6].
	Service Discovery	100%	Mentioned in all sources [1, 3, 4, 5, 6]. Client-side and service-side solutions are possible.
Data layer	Database per server	100%	Mentioned in all sources [1, 3, 4, 5, 6]. Multiple options are possible: shared, cluster.
	Caching	20%	Data is directly available. Mentioned in one source [3].
	CQRS	20%	Command Query Responsibility Segregation. Mentioned in one source [6].

Table 12 - Selected categories and patterns

We used the category information provided in all selected sources (except [1], which had no such information) and the exclusion criteria described in 4.2. The next step is to use these selected sources with their references as input to describe the architectural patterns in detail. The references of these selected sources can be found in Appendix B – Literature study of selected MSA patterns. The duplicate documents mentioned in more than one source per pattern were removed. Some documents were not available and removed from the list because they were not accessible for free (mainly via IEEE). The result of the literature study of these sources is the pattern descriptions of Chapter 6.

6 MSA patterns descriptions (RQ2)

This chapter investigates the five selected sources and their information and references to detail the seven selected MSA patterns related to communication and data-layer. The references used of the five selected sources can be found in Appendix B – Literature study of selected MSA patterns.
The description of each pattern will use the structure of an architectural pattern described in 3.2 and the extra items 'variations' and 'keywords' as explained in 4.2. These descriptions are an answer to research question RQ2.

6.1 API Gateway

Context	External clients have different needs and ways to communicate with the functionality of a microservices-based system.
Problem	How do external clients communicate with the microservices?
Solution	A service that provides each client with a unified interface to services [8, 9]. It hides the (changing) partitioning and endpoint locations of microservices for the client [1, 3, 4, 5, 6] and aggregates multiple microservices' results. It has optional functions like: Authentication and authorization (security), Protocol conversion, Rate limiters, Monitoring [6], Load balancing [8], Consistency checks [11], Reverse proxy [10], Filtering or Page caching [17].
Constraints	All traffic between clients and microservices is routed via the API gateway, so changes in the internal microservices can be hidden [13]. An API gateway does not have a persistence layer [12]. An API gateway should not implement business logic [17]. Microservices APIs should be business-oriented rather than technically oriented. A business-oriented component API can be more readily understood [17]. It is important to manage strict versioning between API Gateway and Backend-end microservices to prevent breakages [12]. Communications should be limited to the Secure Hypertext Transfer Protocol (HTTPS) to ensure sensitive information transmission [8]. There is additional network traffic because an extra 'hop' is made via the gateway service [1, 3, 4, 5, 6].
Variations	A lightweight protocol is one specific like REST/HTTP [14], or multiple such as webpages/HTTP, SOAP/HTTP and REST/HTTP [12]. Depending on the user requirements, the API Gateway can be implemented as a single service or multiple independent services [9]. The Backend for Frontend pattern defines a separate API gateway for each type of client [18]. [15] mentions that protocol conversion is not a function of the API Gateway. It also internally connects microservices with each other [10].
Keywords	API, gateway, proxy, aggregate, façade, dispatcher

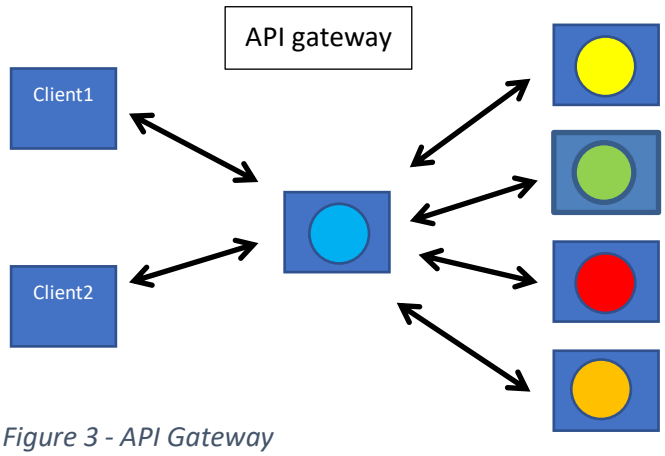


Figure 3 - API Gateway

6.2 Messaging

Context	Because of their distributed nature, microservices will have to exchange data via a communication channel.
Problem	An RPC-like communication protocol (such as REST) is generally used. It has the disadvantage that it is synchronous and, therefore, blocks if the request or channel receiver is not available. Another disadvantage is that a sender must have knowledge of the receiver. A recipient may be interested in the information now but no longer at a later time. There must be an easy mechanism to subscribe and unsubscribe.
Solution	Use an asynchronous method of exchanging real-time information. This can be done by using a message bus to connect the sender and receiver. The service has a mechanism to subscribe to a specific topic by a receiver. This results in a loose runtime coupling and improved availability because the sender is not directly dependent on the receiver. Therefore, it is possible to add new (versions of) microservices without affecting the sender or the already subscribed receivers [23, 24]. Several message serialization formats will be supported, like XML and JSON [23]. Monitoring of the events (messages) is important for maintenance [25]. The message bus can be deployed in a clustered environment to improve scalability by distributing the workload by load balancers [23].
Constraints	It depends on a message bus which adds complexity and should be highly available. However, the message bus is more lightweight than the traditional Enterprise Service Bus that provides functionality like transformation and choreography. The message bus does not contain business logic [9]. It follows the principle of smart endpoints and dumb pipes, which implies the usage of lightweight middleware components such as messaging systems (dumb pipes) and intelligent services themselves (smart endpoints) [27].
Variations	There are several ways to exchange information [1, 3, 4, 5, 6]: -request by the sender with an asynchronous response of the receiver -notifications by the sender, the receiver does not reply -publish/subscribe: a sender publishes a message to one or more receivers -some receivers do send a reply on some occasions. The used protocol could be implementation-specific or based on a standard like AMQP [26] or MQTT [9].
Keywords	Publish, subscribe, pub, sub, messaging, non-blocking

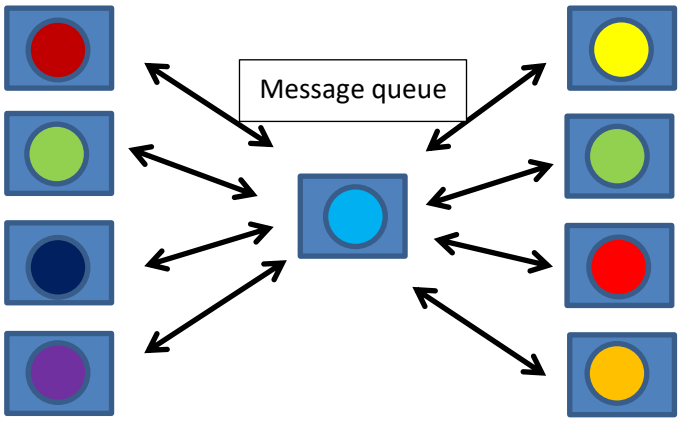


Figure 4 - Messaging

6.3 Circuitbreaker

Context	Microservices work together to handle a specific service request. For this, they communicate with each other to exchange data.
Problem	Sometimes a microservice is temporarily unavailable, or the communication channel between the microservices is temporarily unavailable. If a calling microservice constantly calls the unavailable microservice, this leads to a lot of extra requests. These extra requests can cause the system to use up its resources, making other microservices unavailable. This increases the effect of unavailability and resource consumption even more.
Solution	Place a circuit breaker between two microservices. The circuit breaker detects that service requests repeatedly fail. The circuit breaker will now not pass on any service requests (state is open). Only after a time-out, a few requests will be allowed (state is half-open). If the requests fail again, then it will wait again (state is again open). This loop continues to repeat until the microservice to be called is available again (state is closed). The circuit breaker works in combination with a load balancer [19]. The load balancer will only route requests to services with a circuit breaker in state closed and lowers the requests to services, which circuit breaker has a 'half-open' state.
Constraints	Microservices always communicate via a circuit breaker [16]. A circuit breaker should log the number of failed requests for monitoring by the system's administrator [20]. A client using the circuit breaker should design a strategy when a service is unavailable [21].
Variations	Many different techniques can be used to detect a failure situation (state will be open), like communication failures, malware detections and resource availability exceptions [16]. Many different criteria can be used to change the state to open, half-open or closed [20, 21, 22]. A circuit breaker can be implemented at the client-side, service-side, or combination of the client- and server-side [21].
Keywords	Circuit, breaker, failure, time-out, unavailable

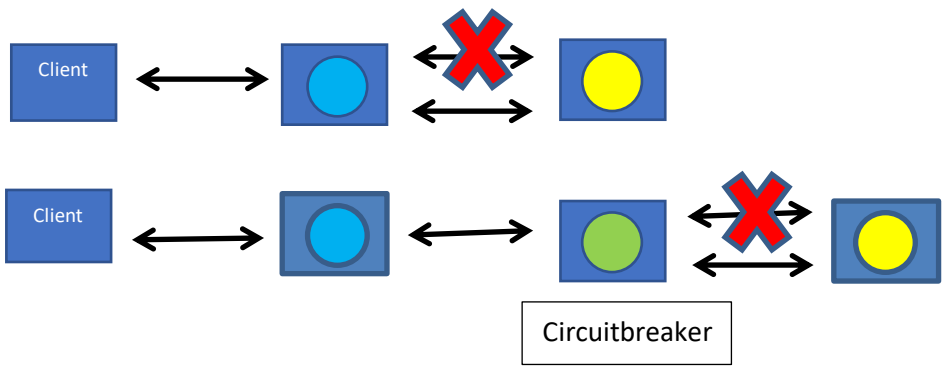


Figure 5 - Circuitbreaker

6.4 Service discovery

Context	A user of a microservice (client or another microservice) uses one or more microservices.
Problem	In a microservices-based environment, microservices can be scaled up or scaled down or (on the fly) move to another server. As a result, the locations of the microservices can change dynamically. How does a user of a microservice know the location of the microservice it needs to call.
Solution	The microservices must register themselves in the service discovery component (often called service registry) [21]. This component keeps track of the address and port number of a microservice instance [13]. The user (client-side) or router/gateway (server-side) queries the service discovery for the exact location of a microservice (CS-1 or SS-2 in Figure 6 below). The client or gateway/router can now pass the request to the appropriate microservice (CS-2 or SS-3). The router/gateway acts as a load balancer to be able to distribute the load on the available microservice instances. The client-side and server-side approaches can be used in combination. The server-side discovery service is used for public services to the outside world, and the client-side discovery service is used for internal interactions [21]. The sophisticated discovery service can use a mechanism for service consumers to know which microservice to use, e.g. minimize the network hops to the provider or minimize the total number of providers used by a given consumer or by using the oldest or newest provider available [29].
Constraints	A service registry is required to register the available services [13]. The service instances are registered with the service registry on start-up and removed from the register on shutdown or unhealthy status [5]. Implementing a health check is required to prevent microservices from being stuck in a failure state [6]. Clients of microservices should not use microservices' physical addresses, which applies to development, test and production environments [17]. The client-side approach causes a tight coupling between the client, service registry and microservices. This has the advantage that no additional hop via the router/gateway is required [4]. A service registry is a critical component and should be highly available [5].
Variations	As already mentioned in the solution, there is a client- or server-side approach possible. A central service or a distributed service can be used at the server-side [28]. A microservice can register itself and implements a heartbeat health check. It can also register itself via a separate registration service, which also monitors the service status [6]. [29] mentions the usage of a fully decentralized architecture with homogeneous nodes in a completely connected graph, avoiding a central registry, the designation of “master” nodes, or complex network topology. They also mention a lightweight and extensible monitoring and self-healing mechanism essential for any viable service discovery solution. [6] mentions a special health check API for microservices to be used by the service registry. A service discovery should not contain the service description, but that should be retrieved from the service itself to have the latest version. However, to avoid network overhead, using the service discovery for it is better [28]. The service registry could also contain (besides information about the service itself) information about the health status or uptime of the microservice [17]. A service registry can be omitted if there are only a few services because of the overhead [17]. Sometimes the service registry is mentioned as a separate pattern [5, 6]. [6] mentions the possibility to cache the information of the service registry for performance and availability reasons.
Keywords	Discovery, registry, lookup, registration, router

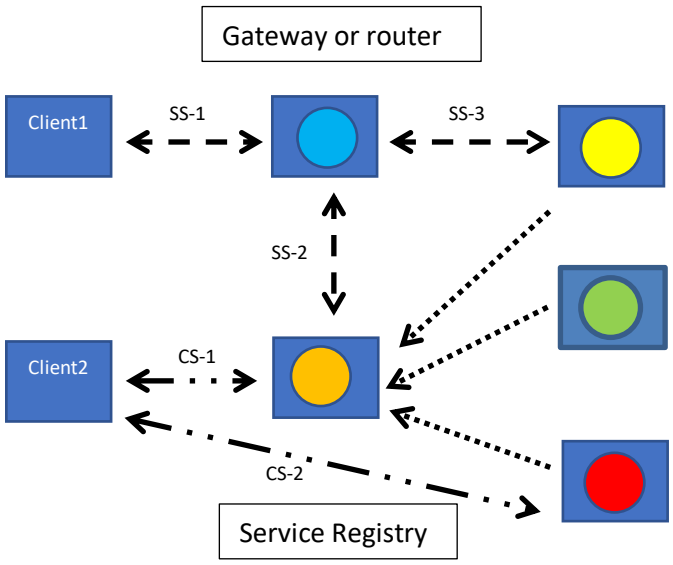


Figure 6 - Service discovery

6.5 Database per service

Context	Every microservice needs access to data or has to store data.
Problem	How do microservices interact with databases?
Solution	The starting point is that every microservice manages its own data [2]. The persistent data is kept private for that microservice and is only accessible via its API. The database is part of the microservice's implementation and cannot directly be accessed by other microservices [5]. A service transaction only involves its own database [6]. This approach leads to services that are loosely coupled. It is easier to change a microservice's data model because that service only uses it. There are multiple options for storing and managing the data a. Use a shared database server, and each microservice has its own set of tables or schemas. b. Choose a database implementation that best fits the service's operation (e.g. SQL, NoSQL, Search and Graph databases) [27]. c. Use a database cluster to separate the microservice and the database (if traffic is high). Use policies to enforce data separation, like different database users/passwords per microservice [6].
Constraints	A (shared) database can become a resource problem if it is not horizontally scalable [23, 24]. A database should be shared between multiple instances of microservices to avoid inconsistent data [27]. Storing in or retrieving (joining) data from multiple microservices is more complex. Distributed transactions should be avoided, and the usage of compensating transactions for rollback is promoted [6].
Variations	A variant is that the database implements the microservice (database is the service) [5]. [30] proposes to use a Graph database for microservices because of their flexible usage. [31] proposes the usage of sharding, where the database is split into a number of shards. User-related data is kept in one shard, which is synchronized with the central database.
Keywords	Database, data, cluster, store, SQL

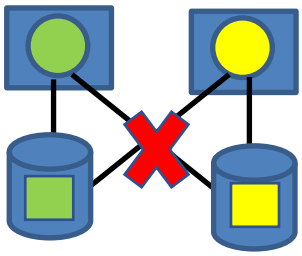


Figure 7 - Database per service

6.6 Caching

Context	Clients need to retrieve information from microservices. They will have to make several calls over a communication channel.
Problem	Due to network latency, unavailability or long processing time, clients may experience problems in being responsive if they need to retrieve information from microservices.
Solution	The communication between clients and microservices is often via the API gateway (or Frontend for Backend). Using a cache at the client-side avoids making several calls to the gateway [5].
Constraints	The lifetime of the data in the cache needs to be managed, which can add complexity to the client application [17].
Variations	There are several implementations of a local (near) cache like a hashtable data structure or a HTML5 local storage. On mobile applications, the Core Data or Property Lists can be used on iOS, and SQLite can be used on both iOS and Android [17]. Some architectures rely on caching at the microservices (like page cache in Gateway [17] or database. There is no need to implement it on the client-side [23, 24].
Keywords	Cache, network, latency, responsive

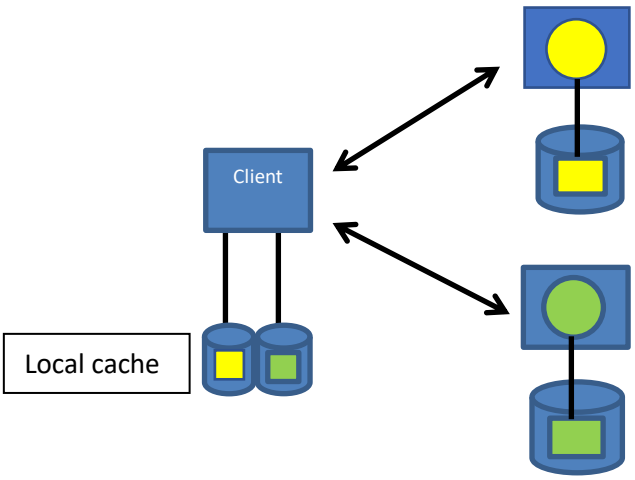


Figure 8 - Cache

6.7 CQRS

Context	Consumers are writing to and reading data of microservices [6, 20].
Problem	Customers that read data have other requirements than consumers writing the data. They may even conflict with each other, resulting in poor performance.
Solution	Split the write and read models into separate models. Both models can be optimized for their purpose. It is possible but not required to use different database implementations for read (queries) and write (commands). The read model is synchronized with the write model as indicated with the arrow from left to right in the picture below. The read models can be duplicated for performance reasons. The commands should be task-based rather than data-based (placeOrder i.s.o. set orderStatus to NEW) to hide the implementation details. They can be placed in an asynchronous message queue for performance optimization. Queries never modify the store's data and are retrieved as simple DataTransferObjects (DTO) without providing additional domain information. The authorization policies on the write and read models can differ, making it simpler to implement security requirements. E.g. only a few actors can access the write models for updates.
Constraints	The read model is 'eventually consistent' meaning that the information can take some time to be updated. If the write model is updated by a command and a read model is updated by an event, then the command and the event should be done in the same transaction to ensure data is not lost or duplicated. Standard ORM (Object Relation Mapping) tools do not support the separation of read and write models. The solution can become complex and resource-consuming if the read models need to be generated from write model events.
Variations	A read model can be part of one service as the counterpart of that service's write model. The read model can also be an aggregated view of multiple services. There are multiple implementations of the read and write models possible [41, 52]: Read: e.g. materialized views, document store Write: e.g. relational, event store
Keywords	Command, query, responsibility, segregation, read, write, model

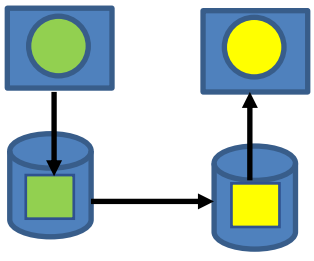


Figure 9 - CQRS

7 Research Phase 2 (Practical)

This chapter describes the thesis's practical phase and starts by highlighting the research question and research method related to this phase. This research question is answered by investigating MSA patterns' implementations in several Open-Source projects, that use microservices. These implementations are described in detail in Chapter 8 and are input for a static code analysis tool to search for MSA patterns in these Open-Source projects. This static code analysis tool, described in Chapter 9, will create an overview of used MSA patterns in the Open-Source projects (see Chapter 10).

7.1 Research question (RQ3)

The thesis's primary goal is to investigate the implementation of architectural patterns in Open-Source (OS) projects that use microservices.
The research is limited to patterns in the field of:
- Communication between microservices
- Data-layer of microservices

After identifying the communication and data-layers' patterns and their corresponding definitions in Phase 1, we move to identify those patterns in existing projects. In this phase, we aim to answer the following research question:

3. What are the most common microservices communication and data-layer architectural patterns in Open-Source software projects? What are the characteristics of these implemented patterns (RQ3)?

7.2 Research method (RQ3)

The research method for the second practical phase of this thesis is an investigation of MSA patterns in twenty OS projects that use microservices. We split the twenty projects into ten so-called whitebox projects and ten blackbox projects. The ten whitebox projects are investigated manually (see Chapter 8) and are the basis for an MSA patterns static code analysis tool (see Chapter 9). The output of the static code analysis tool shows the MSA patterns found in the ten OS projects and is our answer to RQ3 (see Chapter 10).
The ten blackbox projects are not manually investigated but are used to verify the quality of the static code analysis tool. We execute the same programs on the blackbox projects as done on the whitebox projects and see if the results match the outcome of the whitebox projects (see also Chapter 10).
Subchapter 7.2.1 describes the selection criteria for the twenty OS projects. The manual investigation of the ten whitebox projects is described in Subchapter 7.2.2. The development of MSA patterns analysis programs for a static code analysis tool is described in Subchapter 7.2.3. The static code analysis tool is described in 7.2.4.

7.2.1 Selection of OS projects using microservices

In this subchapter, we define the selection criteria for the OS projects in GitHub. During the literature study of Phase 1, we encountered two sources that also used OS projects from GitHub. Márquez et al. [3] analyzed source code projects in GitHub and applied the following search string: ("micro services" OR "microservices" OR "micro-services") AND ("system"). They used the following inclusion criteria: benchmark requirements for microservices projects proposed by Aderaldo et al. [44] and projects with source code available.
The benchmark includes 12 requirements (related but not the same as the 12-factor app[8]) as shown in Table 13:

Context	Requirement	Remarks
Architecture	R1: Explicit Topological View	A view is needed to keep track of services and deployments.
	R2: Pattern-based Design	For our thesis, this is the most crucial requirement.
DevOps	R3: Easy Access from a Version Control Repository	Continuous Integration and Continuous Deployment (CI/CD) are essential aspects of microservices architectures.
	R4: Support for Continuous Integration	
	R5: Support for Automated Testing	
	R6: Support for Dependency Management	
	R7: Support for Reusable Container Images	
	R8: Support for Automated Deployment	
	R9: Support for Container Orchestration	
General	R10: Independence of Automation Technology	For the CI/CD requirements mentioned above, it should be possible to choose several implementations.
	R11: Alternate Versions	Different technology implementation choices.
	R12: Community Usage & Interest	"Easy to use and of interest to its target

[8] https://12factor.net/

		research community".

Table 13 - Benchmark requirements for microservices architecture research

Márquez et al. [3] had as exclusion criteria: projects with no solid information (basic examples, projects in progress), tools to build microservices (instead of frameworks), component-based projects to build microservices (e.g., projects just for gateway components to microservices), and projects used as an example for lectures or talks.

The analysis code of the method used by Márquez et al. [3] was not published, and many manual steps were needed. The architectural patterns of the GitHub projects were reported in excel sheets. The sheets show a list of projects using framework dependencies and a list of frameworks implementing microservices architectural patterns.

Imranur et al. [45] also selected microservices projects from GitHub. In GitHub, they used the following search string: "micro-service" OR microservice OR "micro-service" filename:Dockerfile language:Java. They also got feedback from developers on forums, which made them add specific projects to their list. They exclude libraries, tools to support the development, including frameworks, databases, and others. They implemented a simple report functionality to show these microservices projects' framework dependencies without showing the relations to architectural patterns.

FTGO (Foot To Go) is the example application of the book of C. Richardson [6, 41]. As it contains a lot of microservices patterns, it will be very useful for analysis purposes.
We added project Alibaba, which we encountered on GitHub as an MSA project, to have a set of precisely 20 projects.

Márquez et al. [3], Imranur et al. [45], C. Richardson [6, 41] and the Alibaba project provide helpful information for selecting projects for our thesis:

1. They are Java projects because of the available knowledge of this programming language. Java is also a very popular language in GitHub [46, 47].
   AND
2. All the Java projects selected by Márquez et al. [3].
   AND
3. All the Java projects selected by Imranur et al. [45].
   However, we only include those projects with a large number of Lines of Code (LoC > 10.000)
   AND
4. FTGO application of Richardson [6] and MSA project Alibaba

One-half of the OS projects (whitebox) will be used to develop the static code analysis tool, and the other half of the OS projects (blackbox) will be used to validate the static code analysis tool. One important reason for choosing a project as blackbox project is the lack of good (English) documentation.
This results in the following list:

Whitebox OS projects	URL	No.	Year
Piggy Metrics	github.com/sqshq/PiggyMetrics	1	2015
FTGO Application	github.com/microservices-patterns/ftgo-application	2	2017
E-Commerce App	github.com/venkataravuri/e-commerce-microservices-sample	3	2016
Blog post	github.com/fernandoabcampos/spring-netflix-oss-microservices	4	2016
Tap&Eat	github.com/jferrater/Tap-And-Eat-MicroServices	5	2016
Delivery system	github.com/matt-slater/delivery-system	6	2016
Graph Processing	github.com/kbastani/spring-boot-graph-processing-example	7	2015
Lakeside	github.com/Microservice-API-Patterns/LakesideMutual	8	2018

Micro company	github.com/idugalic/micro-company	9	2016
SiteWhere	github.com/sitewhere/sitewhere	10	2014
Blackbox OS projects	URL	No.	
CAS Microservice	github.com/ArcanjoQueiroz/cas-microservice-architecture	1	2017
Freddy's bbq joint	github.com/william-tran/freddys-bbq	2	2016
Genie	github.com/Netflix/genie	3	2013
microService	github.com/bishion/microService	4	2016
Microservices book	github.com/ewolff/microservice	5	2015
Movie recommendation	github.com/mdeket/spring-cloud-movie-recommendation	6	2016
Netflix microservice	github.com/yidongnan/spring-cloud-netflix-example	7	2016
Share bike	github.com/JoeCao/qbike	8	2017
Task track support	github.com/yun19830206/CloudShop-MicroService-Architecture	9	2017
AliBaba	github.com/alibaba/spring-cloud-alibaba	10	2017

Table 14 - Selected Open-Source projects with year project started

7.2.2 Manual investigation of MSA patterns

We manually investigate MSA patterns in ten OS projects that use microservices. The criteria for selecting these projects as 'whitebox' projects are described in Subchapter 7.2.1.
The selected projects will be stored in a local Git repository and compiled with Maven or Gradle tools. The implementation of MSA patterns in these projects will be found by manually investigating the project source code, configuration files and available documentation. The pattern descriptions of Chapter 6 are input for this manual step. The keywords recorded for each MSA pattern can be used in a simple search program described in 7.2.2.1, which can perform a text-based search in the OS projects. The manual investigation results are the implementations of the MSA patterns and will be described in Chapter 8.

7.2.2.1 Keyword matching of MSA patterns

Keyword matching will be used as an extra opportunity to find MSA patterns in Java projects.
A number of keywords are already defined per pattern in Chapter 6, and a simple grep-based script will search for these keywords in files of a Java project.
We will use the following of grep as shown in Table 15:

Grep command options	Remarks
-i	Ignore case
-r	Recursive in directory
-H	Print file with output lines
--color	Display the word in color
'keyword1|keyword2'	Multiple keywords (OR)

Table 15 - Grep-command options for keyword matching

An example for the API gateway pattern:
grep –i –r –H –-color "gateway\|proxy\|aggregate\|facade\|dispatcher"

We will also use the Eclipse IDE[9] to read and search for keywords in the source code, configuration, and build files of the projects.

7.2.3 Static code analysis tool for MSA patterns

We create programs containing queries for a static code analysis tool that can automatically detect the selected MSA patterns in the ten whitebox OS projects. An analysis tool has the advantage above

[9] https://www.eclipse.org/ide/

a manual investigation: it can query, save, and visualize the many different implementations of MSA patterns. It is also possible to detect a wrong pattern usage if certain pre-defined constraints are not met. Although there are several tools for static code analysis [50], most of them are used to analyze the code quality of projects. Quality analysis of source code focuses on code duplication, complexity or security.
For the analysis of MSA patterns, the focus is on the architecture level of a Java project. The architecture (3.2) of an application is based on characteristics such as:

- The layers that structure the application (view, service, data)
- The decomposition into components (internal and external (libraries))
- The interaction between these layers and components
- The interaction with other applications or users
- The usage of architectural patterns and their constraints

This leads to the following requirements for an analysis tool:

Requirement	Remarks
Identify an architectural concept of a layer	Layers are microservices with internal or external APIs, repositories
Identify an architectural concept of a component (e.g. class, interface, library)	These are the MSA patterns e.g. an API gateway.
Show a relation between architectural concepts	A communication path between two microservices or a dependency of a microservice on an MSA pattern.
Apply attributes to an architectural concept or relation	Examples of attributes are microservice-name, URL or communication protocol.
Detect an architectural constraint	An example is that multiple microservices should not use the same repository.

Table 16 - Requirements for a MSA patterns static code analysis tool

The third research question (RQ3) will be answered after manually analyzing the ten OS whitebox projects and the outcome of the static code analysis tool. This tool will show per OS projects how often the selected MSA patterns of Chapter 6 are used and show the type of implementation. The requirements of the static code analysis tool of Table 16 can be met by using the existing tool JQAssistant, which uses the Cypher query language and the Graph database Neo4J. Subchapter 7.2.4 describes the JQAssistant tool, Cypher[10] query language, Neo4J[11] database.

We will use ten other so-called 'blackbox' OS programs to validate the static code analysis tool. The blackbox OS projects will not be manually investigated, but we will compare the result (no. of MSA patterns found) with the result of the whitebox OS projects. This comparison will give an indication if the static code analysis tool is 'mature' enough to detect the MSA patterns in other OS projects. The criteria for selecting the ten blackbox OS projects are described together with the ten whitebox OS projects in Subchapter 7.2.1.

The two major steps of the research method for Phase 2 can be drawn in a process that contains six detailed steps, as shown in Figure 10. The first step is downloading the selected OS GitHub projects that contain microservices into a local Git repository. The second step is the manual investigation of the implementation of MSA patterns in the downloaded OS projects. The results of this investigation can be found in Chapter 8. The third step will create the Cypher queries based on the found implementations of MSA patterns. These queries will be part of a static code analysis tool, which will be described in Chapter 9. Step four will compile the OS projects, which results in JAR-files. Together

[10] https://neo4j.com/developer/cypher/
[11] https://neo4j.com/

with additional configuration files, these files will be scanned by JQAssistant and then loaded with JQAssistant into the Neo4J Graph database as a result of step five. The final step will run the analysis programs and create an overview of the found implementation of MSA patterns per OS project.

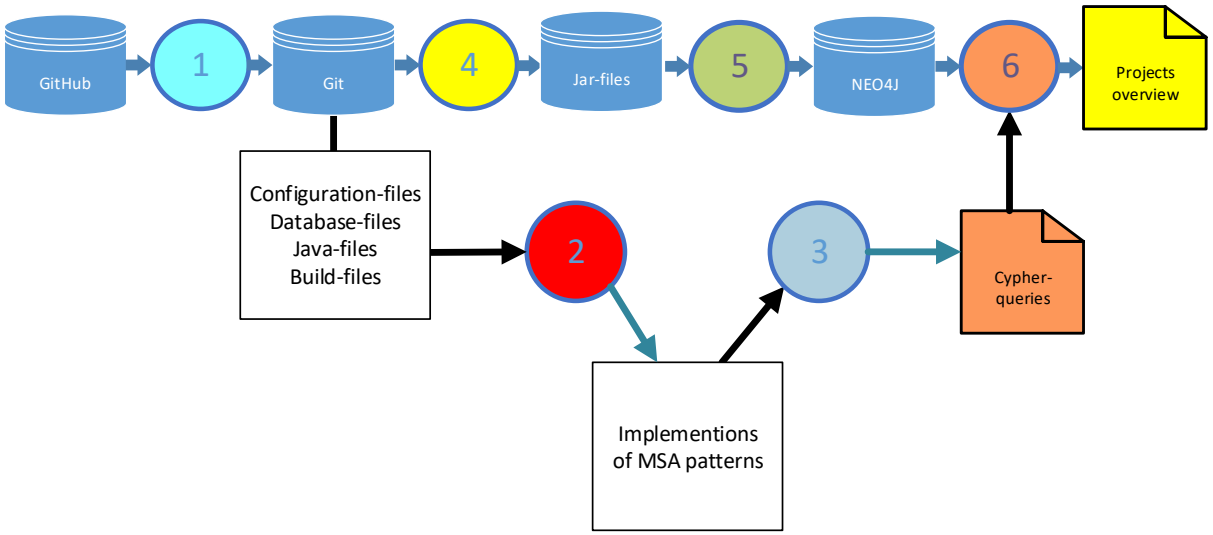


Figure 10 - Overview of the practical research process

The description of the steps:

Step	Description
1	Download selected Open-Source projects from GitHub into local Git
2	Search for implemented MSA patterns in the selected projects
3	Design Cypher queries based on these implementations as the basis for the analysis program.
4	Compile the selected Git projects
5	Import the selected projects (e.g. jar files) into Neo4J with JQAssistant
6	Create overviews of the MSA patterns for the selected projects with the help of the analysis program.

Table 17 - Steps of the research process for answering RQ3

We will exclude steps 2 and 3 for the ten blackbox projects. They are also downloaded in the local Git (step 1), compiled (step 4) and imported in Neo4J with JQAssistant (step 5). We use the existing programs developed for the whitebox projects to analyze the blackbox projects (step 6).

7.2.4 Static code analysis tool JQAssistant with Neo4J/Cypher Graph database

JQAssistant can scan compiled Java projects and detect basic Java concepts like classes, interfaces, methods, and relations. A class has e.g. a relation to the methods it contains. JQAssistant will store its data in a Neo4J Graph database. The Graph database has two elements: a node and a relationship. A node can represent anything like a car, a person, or a Java Language concept (like class, file). The relationship is an association between two nodes. The following example related to Twitter shows three nodes with the label USER with an attribute name and the nodes' relations with the label FOLLOWS.

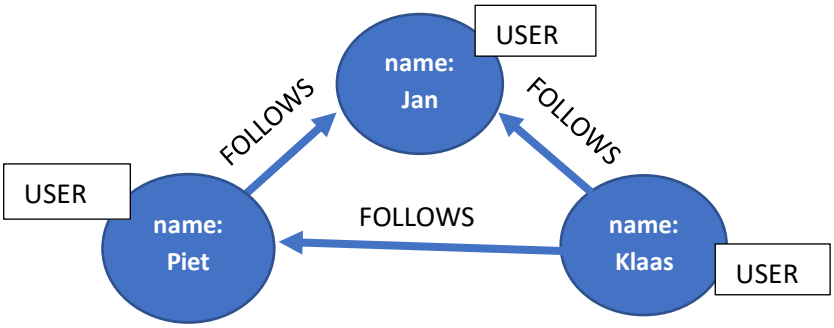


Figure 11 - Example of twitter users as Cypher nodes and relations

Cypher is the query language for Neo4J. For example, the nodes and relations in Figure 11 can be created with:

CREATE (j:User {name:'Jan') <- [:FOLLOWS] - (k:User {name: 'Klaas') –
[:FOLLOWS] -> (p:User {name: 'Piet') – [:FOLLOWS] -> (j)

It can be queried: e.g. return all users that Klaas is following:

MATCH (k:User {name:'Klaas') - [:FOLLOWS] -> (u:User) RETURN u

Alternatively, return all users that Klaas is following or that are following Klaas:

MATCH (k:User {name:'Klaas') - [:FOLLOWS] - (u:User) RETURN u

JQAssistant applies so-called concepts to compiled software artifacts. These concepts are nodes and relationships between these nodes and attributes of nodes or relationships. It has support via plugins for several parts of the Java Language: Java, EJB3, Java EE 6, Spring, Maven, XML, YAML [54].
The Java plugin has a Class scanner. These plugins will scan all files with suffix .class and create nodes with the following labels: Class, Interface, Annotation, Method, Enum, Field, Constructor, Parameter. The Java plugin will also add relationships between nodes. A Class node has, for example, a DECLARES relation with its methods. In addition, the Java plugins detect constraints related to the created concepts and relationships. Concepts and constraints are stored in so-called rules files. A rule file contains groups, concepts and constraints. Groups make it possible to group concepts and constraints. JQAssistant has pre-defined rule-files, but it can also create its own rule-files related to MSA-patterns[12] in AsciiDoc[13] style.

[12] https://jqassistant.org/fix-your-microservice-architecture-using-graph-analysis/
[13] https://asciidoctor.org/

The following example defines a group msa-patterns:Default with a concept msa-patterns:microservice-basis and constraint msa-patterns:microservice-springboot.

```
[[msa-patterns:Default]]
[role="group",includesConstraints="msa-patterns:*",includesConcepts="msa-patterns:*"]

[[msa-patterns:microservice-basis]]
[source,cypher,role=concept]
----
ADD CYPHER QUERY 1
----
[[msa-patterns:microservice-springboot]]
[source,cypher,role=constraint,requiresConcepts="msa-patterns:microservice-basis"]
----
ADD CYPHER QUERY 2
----
```

The constraint has a dependency on the concept. Therefore, the group will include all constraints and concepts in the directory, which name starts with 'msa-pattern:'.
JQAssistant will require the following two steps:

1. scan the project jar-file; this will apply the basic concepts of the Java language.
2. analyze the project with the rule files.

The following example shows the scanning of two jar files of a Java project and the analysis of the rules for group msa-patterns:Default in the rules directory.

```
>jqassistant.sh scan -f project-file1.jar project-file2.jar
>jqassistan.sh analyse  -groups msa-patterns:Default -r /c/user/rules-directory
```

JQAssistant will show all applied concepts and constraints violated as the scan and analyze phases' output and stores it in result files.
All concepts and relationships created by JQAssistant in the scan and analyze phase are stored in the Neo4J Graph database. In addition, Neo4J provides a user interface, making it possible to query the nodes and relations via Cypher commands. The output is shown in graphical or text form.
Visualizing concepts with different sizing and colors makes it easier to understand them. Neo4J has a standard style file that can be adapted. We will add each concept in this file with its own color and size. Neo4J also has a module called Neo4J Bloom[14]. Bloom enables creating pre-defined views per MSA pattern with Cypher queries and styles (colors and sizes of concepts).

[14] https://neo4j.com/bloom/

8 MSA patterns implementations

This chapter will describe the patterns' implementation details as part of step two of the research process described in Chapter 7.
We will use the description of the selected MSA patterns in Chapter 6. With the help of the Eclipse IDE and the keyword search tooling of Subchapter 7.2.2.1, we manually analyze the set of the ten so-called whitebox Open-Source projects listed in Subchapter 7.2.1. The implementation details are described in the current chapter and are the basis of the analysis programs described in Chapter 9 used by the static code analysis tool.
The analysis programs will provide an overview of the implemented MSA patterns in Chapter 10 to answer research question 3 (RQ3).

What we mean by implementation details are the software elements used to implement the patterns, such as:

- Java language elements (e.g. class, method, variable, annotation constructions)
- Configuration files (YAML and properties files)
- Build files (maven and gradle)
- Framework libraries (e.g. Spring, Microprofile)

We will mention the external frameworks libraries[15] related to the MSA patterns encountered in the selected projects. The code snippets (Listings) are general examples of how the pattern can be implemented, so they are not part of a specific project.
As mentioned in Subchapter 3.3, there are quite some frameworks supporting the microservices architectures for Java projects. Spring and Microprofile are 'core' frameworks supporting the development and deployment of microservices in a container environment. Eventuate and Axon frameworks are used in the implementations of the messaging and CQRS patterns.
The following subchapters will give a high-level overview of the implementations of microservices and the related MSA patterns, which we found by manual analyzing the ten OS whitebox projects. The projects implementing such a pattern are shown in the overview created by the static code analysis tool and can be found in Chapter 10.
We will start with the implementation details of microservices to be able to describe the implementation details of the related MSA pattern. The description and example (listing) of an implementation will refer to a project if an implementation is very specific for that project. We will describe these project-specific implementations at the end of each subchapter. Examples are the FTGO-application and the SiteWhere [42, 43] projects, both having many 'own' implementations.

[15] https://medium.com/microservices-architecture/top-10-microservices-framework-for-2020-eefb5e66d1a2

8.1 Basis of microservice implementations

Before starting with MSA patterns, the implementation of microservices will be described. In the ten investigated OS projects, we found that a Microservice communicates via endpoints with external parties (e.g. frontend applications) or via clients with other microservices. It also has to store data in its own repository.

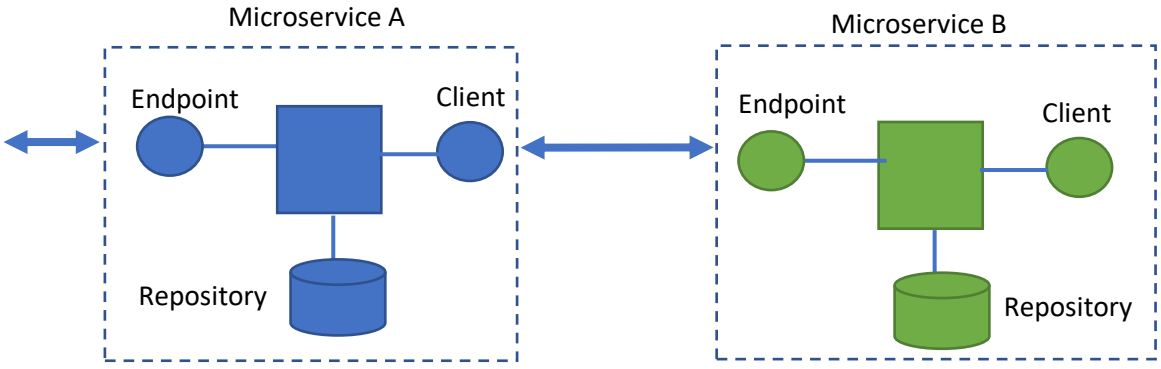


Figure 12 - Microservice endpoint, client and repository

We will describe in this subchapter the implementation details for microservice, endpoint, client, repository and will also mention how microservices will retrieve their configuration.

8.1.1 Microservice central element

We found that all whitebox projects, except SiteWhere (see below), use a Spring framework to define a microservice. The Spring annotation RestController on a class combines a Controller and a RequestMapping/ResponseBody annotation.
An example of a RestController annotation on a class looks like this:

```
@RestController
public class MyBusinessMicroService
```

Listing 1 – Spring Microservice

An older (via Spring Framework) and newer implementation (via Spring Boot) of a REST-service is also possible via annotation Endpoint on a class. The microservice implementation, which uses annotation Endpoint at the class level, supports the older and less lightweight SOAP protocol (Spring Framework - WS). Alternatively, it enables a newer REST implementation (Spring Boot - Actuator) via method annotations ReadOperation and WriteOperation with features like monitoring (health, metrics) and management.
The different types of Microservices are listed in Subchapter 9.1 and the projects which use these types of Microservices are shown in Subchapter 10.1.

8.1.1.1 Project SiteWhere

The SiteWhere project[16] uses Microprofile with the Path annotation on a class for defining a JAX-RS microservice as follows:

```
@Path("/my-service")
public class MyBusinessMicroService
```

Listing 2 - Microprofile Microservice

SiteWhere is a special (Internet-Of-Things) project and also has its own implementation of a microservice. These microservice will extend from the base-class MicroserviceApplication, which

[16] https://sitewhere.io/docs/2.1.0/

takes care of starting, configuring and stopping a microservice at a very low level. In addition, the configuration is prepared for messaging (Kafka), service discovery (Zookeeper) and analytics.
It has two basic types: Global and Multitenant microservices, which extend from the base class.
The Global microservices are required to manage the system and are split into four microservices:

Instance Management	Bootstraps another microservice (called an instance)
Tenant Management	Manages a group (called tenant) of microservices (instances)
User Management	Manages the users of the system
Web/REST	A REST frontend which queries other backend gRPC microservices

Table 18 - SiteWhere Global microservices types

The multitenant microservices are groups (tenants) of microservices configured in a chain to perform a specific function for an end-user. Each tenant runs isolated from other tenants and has its own database resources and messaging pipeline. Examples of tenants are device management, event sources, inbound- or outbound-processing.
The Spring Boot framework is only used during build time by SiteWhere for deployment in a container environment. Therefore, this is the only Spring dependency of SiteWhere.

8.1.2 Microservice endpoint

Microservices use a lightweight communication protocol [2] to communicate with external applications and each other via the REST protocol. A microservice (e.g. RestController class) should handle different requests/responses by enabling an HTTP endpoint to the class methods. These methods use annotations to specify the HTTP method (like GET, POST) and URL (path). The default return format is JSON, but XML is also possible.
Spring (used by all whitebox projects except SiteWhere) has a few variations, depending on the microservice implementation and the different HTTP methods (like GET, POST). Examples are RequestMapping, GetMapping or PostMapping. The following example shows a RequestMapping annotation on a method.

```
@RequestMapping(path="/myBusinessServiceURL, method=RequestMethod.GET)
public BusinessResponse getBusinessServiceDetails(String param1)
{
        …
        return BusinessResponse response;
}
```

Listing 3 - Spring Endpoint

As already mentioned under microservice, class annotation Endpoint of Spring enables an older SOAP or a newer REST implementation (Spring Boot - Actuator). The SOAP implementation uses method annotation PayloadRoot, and the REST implementation uses the method annotations ReadOperation and WriteOperation with features like monitoring (health, metrics) and management.

Microprofile (project SiteWhere) uses method annotations named GET, PUT, POST in combination with Class Annotation Path. The following example shows a Post annotation on a method:

```
@POST
public BusinessResponse getBusinessServiceDetails(@QueryParam("param1") String param1)
{
        …
        return BusinessResponse response;
}
```

Listing 4 - Microprofile Endpoint

8.1.2.1 Projects FTGO and SiteWhere

The FTGO application and SiteWhere projects use gRPC[17] as an alternative for REST as a protocol to communicate between microservices. gRPC is a low-level protocol for exchanging binary messages and has an interface for the messages described in an IDL[18] (Interface Definition Language) defined by Google. gRPC is like REST, a synchronous protocol, but it has a better performance and support for streaming data.

8.1.3 Microservice client calling other microservice endpoints

A microservice uses an HTTP client to communicate synchronously with other microservice endpoints. Such a client can be created by implementing an interface class with annotation FeignClient. Four out of ten projects (Piggy Metrics, Blog post, Tap&Eat, Delivery system) use it. FeignClient is part of the Spring Cloud framework and has some variants[19] in the library that implements it. These are Netflix Feign, supported by Netflix, but stopped and OpenFeign, Netflix donation, now supported by the Open-Source community. They will seamlessly integrate with the circuit breaker and service discovery patterns described in the following subchapters. The interface annotation contains the name of the called microservice, and the interface contains a method, which has a Request/Get/PostMapping annotation. This annotation has the details (method, URL) of the endpoint of the called microservice.

```
@FeignClient("other-service")
public interface MyMicroServiceClient {
        @RequestMapping (method= RequestMehod.GET, value="/other-service"
        List<Info> getInfo();
```

Listing 5 - Spring Client

The WebServiceGatewaySupport and RestTemplate classes of the Spring Framework support HTTP SOAP and REST clients' older implementations and are used by projects Graph Processing and Lakeside. However, they lack the integration of the patterns named above.

Although not used by any whitebox project, we looked at the Client implementation of Microprofile. Microprofile has a RestClientBuilder class or RestClient annotation to invoke other microservices. It is used by first defining the interface of the called microservice as follows:

```
@Path(/other-service)
public interface MyOtherMicroService {
        @Get
        @Path ("/serviceMethod"
        List<Info> getInfo();
```

Listing 6 - Microprofile Client

8.1.4 Microservice repository

Microservices will use a repository to manage their own data. The annotation Repository of the core Spring Framework is an abstraction to work with a persistence layer and is used by most whitebox projects. However, it is sometimes required to work with a specialized version (extension) of Repository like GraphRepository, CrudRepository, PagingAndSortingRepository or JpaRepository, which are part of the Spring Data framework.

[17] https://www.grpc.io
[18] https://developers.google.com/protocol-buffers/docs/overview
[19] https://www.baeldung.com/netflix-feign-vs-openfeign

```
public interface MyRepository<T, ID> extends CrudRepository<T, ID> {
        T findOne(ID id);
        T save(T entity);
}
```

Listing 7 - Spring Repository

A combination of microservice (RestController) and repository is created by using the annotation RepositoryRestResource on a class. This enables a REST HTTP endpoint to directly access a data access layer and is part of Spring Data. This REST-enabled repository is used by Tap&Eat, Delivery system, Graph Processing and Micro company.

8.1.4.1 Project FTGO and Sitewhere

Project FTGO uses the Spring Repository but also has its own implementation dependent on the Eventuate framework, mentioned at the start of this chapter and has another dependency on an AWS Dynamo Database framework.
Project SiteWhere has its own implementation of a data access layer called Persistance.
Microprofile projects like SiteWhere are based on the Java EE specification (JPA) and will use annotation PersistanceContext. However, SiteWhere does not use it.

8.1.5 Microservice configuration

Each microservice is part of a standalone JAR (Java executable), which is based on Spring Boot (via annotation SpringBootApplication) or MicroProfile (@ApplicationPath). A class with annotation SpringBootApplication has the main method: the starting point to run the program as a standalone process in a container environment.

```
@SpringBootApplication
Class MyBusinessServiceApplication {
        public static void main (String[] args) {
                SpringApplication.run(MyBusinessServiceApplication.class, args);
        }
}
```

Listing 8 - Spring Application

A Microprofile application is created as follows:

```
@ApplicationPath ("myServiceApplication)
Class MyBusinessServiceApplication extends Application {
}
```

Listing 9 - Microprofile application

Configuration is very crucial for microservices to be able to read the environment details. Therefore, the projects use standard dependency Injection with Spring annotations: Configuration, Autowired, Bean, Inject and Microprofile annotations: Inject, ConfigProperty.

To enable an easy way to retrieve external configuration details for its microservices, all whitebox projects (except FTGO, Lakeside and SiteWhere) use a central microservice with Spring annotation EnableConfigServer. This is a simple way to start a microservice correctly and have the configuration details at a central place for several hosting environments (development, test, production).
The annotation SpringBootApplication used by microservices includes the annotations Configuration and EnableAutoConfiguration. The microservices will connect during start-up with the configuration microservice via its bootstrap.yml file:

```
spring:
   application:
        name: my_business-service
   cloud:
        config:
          uri: http://config:8888
          fail-fast: true
```

Listing 10 – Configuration via Spring Bootstrap file

The configuration microservice returns the microservice configuration from its classpath or an external repository like git. Configuration is not only used for starting up but also for retrieving information used by the implementations of the MSA patterns, which will be described in the following subchapters. Configuration details will be stored in YAML files (see bootstrap.yml) or property files (see below).

```
spring.application.name = my_business_service

spring.datasource.url = jdbc:mysql://${DOCKER_HOST_IP:localhost}/my_business_service
spring.datasource.username = my_service_user
spring.datasource.password = my_service_password
spring.datasource.driver-class-name = com.mysql.jdbc.Drive

spring.kafka.consumer.bootstrap-servers = localhost:9092
spring.kafka.consumer.group-id = group_id
spring.kafka.producer.bootstrap-servers = localhost:9092
```

Listing 11 – Spring application properties

8.1.5.1 Project SiteWhere

Project Sitewhere uses Apache ZooKeeper[20] as a central place for configuration management, so no local storage is implemented for microservices. The microservices have a direct connection to Zookeeper and will automatically react to updates. The configuration is described based on Helm[21] charts and is used in a container environment based on Kubernetes.

[20] https://zookeeper.apache.org/
[21] https://helm.sh/

8.2 API Gateway implementations

Microservices communicate with external applications (e.g. frontend applications and mobile applications), and an API gateway will hide the implementation and deployment of microservices for these external applications. The API gateway is a microservice with this dedicated task, and we encountered three out-of-the-box implementations of an API Gateway in the investigated OS projects. The first implementation to create an API gateway service uses the annotations EnableZuulProxy or EnableZuulServer on a class. This kind of API gateway is used by projects Piggy Metrics, E-commerce App, Blog post, Graph processing, Delivery system and Micro company.
Zuul was developed by Netflix and is donated to the Open-Source community, and is part of Spring Cloud. It enables dynamic routing, load balancing and security. EnableZuulProxy is an extension of EnableZuulServer. It has additional routing filters (see class ZuulProxyAutoConfiguration of the Spring framework).

```
@EnableZuulProxy
@SpringBootApplication
public class MyServiceGateway {

    public static void main(String[] args) {
        SpringApplication.run MyServiceGateway.class, args);
    }
}
```

Listing 12 - Spring Zuul Gateway

The routing is defined in a configuration YAML file of the API gateway service:

```
# all external requests starting with /business-service are routed
# to the internal url mybusiness-service/myservice
zuul:
  routes:
    business-service:
       path: /business-service/**
       url: http://localhost:8081/mybusiness-service/myservice
```

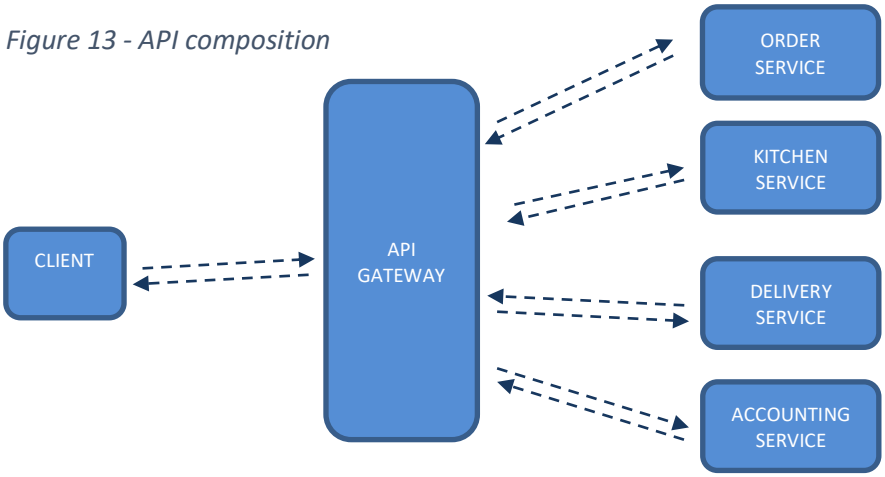
Listing 13 - Zuul routing configuration

The second way of creating an API gateway (project FTGO) is using the RouteLocator class of Spring Cloud. The usage of these classes makes it possible to implement API composition. One input request will lead to interaction with multiple microservices. The API gateway will combine all the results into one response.
The third implementation of an API Gateway (SiteWhere) is via OpenAPI of Microprofile. The OpenAPIDefinition class describes the header of the API. Multiple REST controllers handle the external requests and will forward the request to internal microservices. The complete API is automatically retrieved by combining all OpenAPI annotations in one overview.

8.2.1 Project FTGO

An example of API composition is used in the FTGO application, as shown in Figure 13:

Figure 13 - API composition



A client (e.g. a mobile phone or website) wants to retrieve the details of an order. So it sends a request to the API gateway. The API gateway has an OrderHandlers class that has a method getOrderDetails.
This method implements API composition. It will call via a so-called proxy-class all required microservices (asynchronous via the Mono framework of project Reactor[22]).

The FTGO-application API Gateway implementation uses classes OrderConfiguration and OrderHandler to configure the routes to post an order or retrieve an order from the HistoryService. It needs multiple other services to post an order, as shown in Listing 14.

```
@Configuration
@EnableConfigurationProperties(OrderDestinations.class)
public class OrderConfiguration {

  @Bean
  public RouteLocator orderProxyRouting(RouteLocatorBuilder builder,
                                        OrderDestinations orderDestinations) {
    return builder.routes()
            .route(r -> r.path("/orders").and().
                      method("POST").uri(orderDestinations.getOrderServiceUrl()))
            .route(r -> r.path("/orders").and().
                      method("PUT").uri(orderDestinations.getOrderServiceUrl()))
            .route(r -> r.path("/orders/**").and().
                      method("POST").uri(orderDestinations.getOrderServiceUrl()))
            .route(r -> r.path("/orders/**").and().
                      method("PUT").uri(orderDestinations.getOrderServiceUrl()))
            .route(r -> r.path("/orders").and().
                      method("GET").uri(orderDestinations.getOrderHistoryServiceUrl()))
            .build();
  }
  …
}
public class OrderHandlers {
  private OrderServiceProxy orderService;
  private KitchenService kitchenService;
  private DeliveryService deliveryService;
  private AccountingService accountingService;

  public OrderHandlers(OrderServiceProxy orderService,
                       KitchenService kitchenService,
                       DeliveryService deliveryService,
                       AccountingService accountingService) {
    this.orderService = orderService;
    this.kitchenService = kitchenService;
    this.deliveryService = deliveryService;
    this.accountingService = accountingService;
  }
  …
}
```

Listing 14 - FTGO-application OrderService routing

[22] https://projectreactor.io/

8.2.2 Project SiteWhere

Project SiteWhere uses the third implementation of an API Gateway via OpenAPI of Microprofile. An example can be found in Listing 15.

```
OpenAPIDefinition(info = @Info(title = "SiteWhere REST API", version = "3.0.0",
                description = "REST APIs for interacting with the SiteWhere data model.",
                license = @License(name = "CPAL 1.0",
                url = "https://opensource.org/licenses/CPAL-1.0")),
                components = @Components(securitySchemes = {
        @SecurityScheme(securitySchemeName = "basicAuth", type = SecuritySchemeType.HTTP,
                scheme = "basic", description = "Only used for getting JWT."),
        @SecurityScheme(securitySchemeName = "jwtAuth", type = SecuritySchemeType.HTTP,
                scheme = "bearer", bearerFormat = "JWT", description = "Used for all REST calls."),
        @SecurityScheme(securitySchemeName = "tenantIdHeader", type = SecuritySchemeType.APIKEY,
                in = SecuritySchemeIn.HEADER, apiKeyName = "X-SiteWhere-Tenant-Id",
                description = "Id of tenant to access."),
        @SecurityScheme(securitySchemeName = "tenantAuthHeader", type = SecuritySchemeType.APIKEY,
                in = SecuritySchemeIn.HEADER, apiKeyName = "X-SiteWhere-Tenant-Auth",
                description = "Auth token of tenant to access.") }))
@ApplicationPath("/sitewhere")
public class SiteWhereApplication extends Application {
}


/**
 * Controller for device command operations.
 */
@Path("/api/commands")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
@Api(value = "commands")
@Tag(name = "Device Commands", description = "Services used to interact with device commands
outside the context of a device type.")
@SecurityRequirements({ @SecurityRequirement(name = "jwtAuth", scopes = {}),
        @SecurityRequirement(name = "tenantIdHeader", scopes = {}),
        @SecurityRequirement(name = "tenantAuthHeader", scopes = {}) })
public class DeviceCommands {

    @Inject
    private IInstanceManagementMicroservice<?> microservice;

    ...
```

Listing 15 - SiteWhere API Gateway

8.3 Messaging implementations

Before describing the messaging pattern, we first have to clarify how microservices interact with each other [41]. The HTTP/REST protocol, as described in Subchapter 8.1, is often used for 'one-to-one, synchronous request-reply' communication of microservices. The process (thread) that sends a request will also wait for the reply. However, it is possible to implement it asynchronously (one thread sends an HTTP request, and another thread will handle the HTTP response).
The messaging pattern focuses on asynchronous communication with messages, where a sender is also decoupled from one or more receivers.
Table 19 gives an overview of possible messaging interactions of microservices with each other:

No.	Messaging interactions	Description
1	One-to-one, synchronous request-reply	A client (e.g. frontend or microservice) interacts directly with a microservice.
2	One-to-one, asynchronous request-reply	A client interacts directly with a microservice but does not block (waiting in the same thread) for the response.
3	One-to-one, fire-and-forget	The client sends a request to a microservice and does not get a reply.
4	One-to-many, publish-subscribe without reply	A client publishes a message, and multiple microservices can subscribe. However, the publisher is not aware of the subscribers.
5	One-to-many, publish-subscribe with an asynchronous reply	A client publishes a message, and multiple microservices can subscribe. The subscribers send an asynchronous reply.

Table 19 - Overview of messaging interactions

The usage of messaging and decoupling of sender and receiver (2-5 in Table 19) is achieved using a messaging protocol combined with a lightweight message bus. Message protocols supported are JMS (Java Messaging Service), AMQP (Advanced Messaging Queuing Protocol) and WebSockets. Examples of message-bus are Kafka[23] (FTGO, SiteWhere), ActiveMQ[24] (Lakeside), and RabbitMQ[25] (Micro company) [40]. Those implementations are all Open-Source frameworks.
Dependencies on these frameworks can be detected in the configuration- or YAML-files of the MSA projects as shown in Subchapter 8.1.5. The microservices (or clients using them) will be the publisher and subscriber using the message bus.

8.3.1 Websockets

Projects Delivery system, Graph Processing, Lakeside and Micro company use WebSockets for a message protocol, enabling bi-directional HTTP communication between a client and a microservice of so-called Stomp (Simple or Streaming Text Orientated Messaging Protocol)-messages. This is achieved by using the annotation EnableWebSocketMessageBroker of the Spring Framework. The class with this annotation has to implement methods that deal with topics and endpoints. The messages are stored in memory by default, and it is possible to use an external message bus.

[23] https://kafka.apache.org/
[24] http://activemq.apache.org/
[25] https://www.rabbitmq.com/

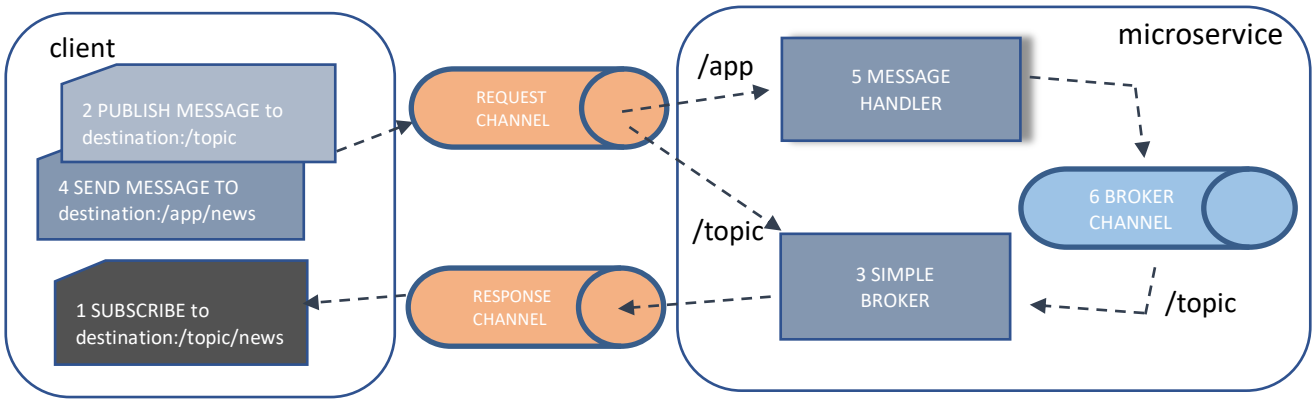


Figure 14 - WebSocket Message Handling

It is possible for a client to subscribe (see 1 in Figure 14) to a topic and to publish (see 2) a message e.g. news, to a topic via a simple broker (see 3). The message will be received by the clients that subscribed to the topic. It is also possible to send a message (see 4) e.g. news via URL /app, to an endpoint message handler (see 5) annotated with MessageMapping. This mapping looks like a RestController RequestMapping; however, it can include a SendTo annotation. The SendTo annotation can also put the message on the /topic/news via an internal broker channel (see 6). A more mature approach to handling messages is to use an external message bus (like Kafka) and not to store them in memory.

8.3.2 Project FTGO

The FTGO-application project has its own framework, called eventuate[26] , to handle messaging and corresponding fault handling. Messages have two types [41]:

- Command: an operation that needs to be executed with data
- Event: the occurrence of a domain event, often with a state change of a domain object

Fault-handling is implemented by so-called Saga's [52]. Richardson [41] described this as a pattern for executing commands where multiple microservices are involved. Failure to execute a command will lead to a rollback by executing compensation commands to all involved microservices.

The eventuate framework creates publishers and subscribers for command (request and reply) and event channels of a message bus (RabbitMQ for this project) to deal with these messages. Figure 15 shows that each service has a request channel for receiving commands with a corresponding reply channel.

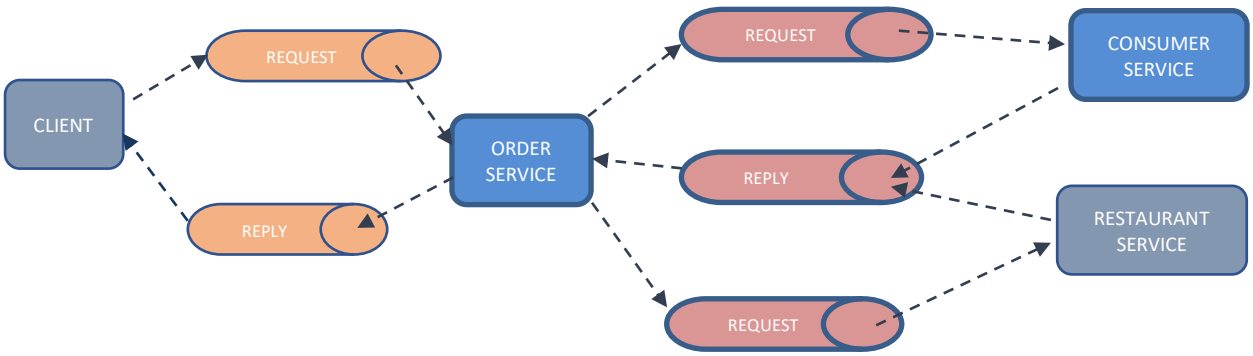


Figure 15 - Command request and reply channels

[26] https://eventuate.io/

A client's request (e.g. an API gateway) to the Order service will result in two separate requests: one to the Consumer service and one to the Restaurant service. They both send a reply to the same reply channel of the Order service.
The order service will combine the replies in one reply to the channel of the originating client.
A change in the domain data of a microservice may also trigger an event. The domain data can change if a command is processed as described above. The event is published on the event channel of a microservice. Other microservices can subscribe to events of this microservice.

In Figure 16, the Order service is subscribed to the Consumer service and the Restaurant service events.

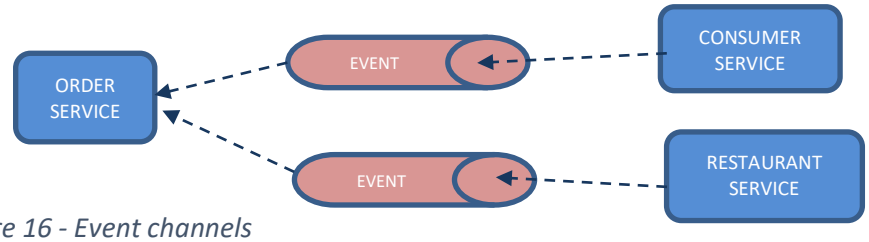


Figure 16 - Event channels

Order Service can use the events to update an internal replica of the data of the consumer service and restaurant service (see also Caching pattern in Subchapter 8.7).

8.3.3 Project SiteWhere

Project Sitewhere uses event-based messaging built on top of Apache Kafka. Microservices will consume events from inbound topic channels, process this data, and then produce events on outbound topic channels.

8.4 Circuit breaker implementations

A microservice with synchronous communication to another microservice has to deal with failures. The other microservice's unavailability can cause these failures due to maintenance or a not working network. A circuit breaker is used to handle these failures, and both Spring Cloud and Microprofile Fault Tolerance have an implementation of a circuit-breaker.
All whitebox projects (except FTGO, E-commerce app and SiteWhere) use Spring.
Netflix Hystrix[27] is the default implementation used by Spring. The configuration of Netflix Hystrix is part of the microservice configuration and makes it possible to specify time-outs, request interceptors, log levels. Netflix Hystrix also enables a fallback when a circuit is open. Fallback means that a handler is provided for the error situation (e.g. returning a previously cached value).
FeignClient described in Subchapter 8.1.3 also includes the Netflix Hystrix implementation. Listing 16 shows an example of a class AppService that has a method getInfoItems that calls another microservice. If the method fails, it will use the fallback method. The SprintBootApplication using this class will use annotation EnableCircuitBreaker to enable the circuit breaker inside the microservice.

```
@Service
public class AppService {
    ...
    @HystrixCommand(fallbackMethod = "fallback")
    public List<Items> getInfoItems() {
          ...
          <Call an external microservice to retrieve the items>
          return items;
    }

    public List<Item> fallback() {
         ...
         //Returns a default fallback
         return fallbackitemslist;
    }
}

import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker
@SpringBootApplication
@EnableCircuitBreaker
public class Application {
   ...
   @AutoWired
   private AppService appService;
   ...
}
```

Listing 16 - Spring Circuitbreaker

Hystrix takes care that the microservice connection is set into open state after a certain amount of failures so that subsequent calls automatically fail and use the fallback. After a while, the circuit breaker goes to half-open state, and if the call succeeds, the state will become closed again.
Hystrix[28] is no longer in active development and is currently in maintenance mode what can be read on the Github project website. Netflix is now using resilience4J[29] for new development projects. Hystrix should be pre-configured with time-out values, where relisience4J can adjust values based on the application's real-time performance.
The annotation EnableCircuitBreaker of Spring Cloud is a façade for using different implementations of the circuit breaker pattern. Spring Cloud also supports the newer framework Resilience4J.

[27] http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html
[28] https://github.com/Netflix/Hystrix
[29] https://github.com/resilience4j/resilience4j

Microprofile uses Resilience4J as standard for the Circuit breaker implementation, and an example is shown in Listing 17. However, Resilience4J is not used by any whitebox project.

```
@Path(/myService)
public interface MyMicroService {
        @Get
        @Path (”/serviceMethod”
        @TimeOut(value=1000, unit=ChronoUnit.MILLIS)
        @CircuitBreaker(requestVolumeThreshold=5,
                              failureRatio=0.25,delay=5000L,successThreshold=1)
        @Fallback(DefaultInfoFallBackHandler.class)
        List<Info> getInfo();
```

Listing 17 - Microprofile circuitbreaker

8.5 Service Discovery implementations

A microservice often needs to connect to other microservices for retrieving data or executing certain business logic. A central registry is used to keep track of the available microservices. A microservice will register itself during startup, and another microservice can ask the registry for the location of a microservice. This mechanism is called client-side discovery (see 6.4). Clients using the service registry are the standard microservices, but also the API gateway and configuration microservice use it.

Multiple implementations are supported by Spring of a central registry like Netflix Eureka, Apache Zookeeper, Consul [34]. The service registry for Netflix Eureka is created as a dedicated microservice using annotation EnableEurekaServer on a class. These annotations are part of the Spring Cloud (Netflix) framework and are used by all whitebox projects except SiteWhere.

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}
```

Listing 18 - Spring EurekaServer

Microservices that register themselves use a servicename that is defined in the variable spring.application.name (properties-file) or spring: application: name: (YAML-file).
The microservices using the service registry uses annotation EnableEurekaClient (Eureka only) or the more general EnableDiscoveryClient; both are part of Spring Cloud.

```
@EnableDiscoveryClient
@SpringBootApplication
public class ServiceRegistrationAndDiscoveryClientApplication {

        public static void main(String[] args) {
                SpringApplication.run(ServiceRegistrationAndDiscoveryClientApplication.class,
                        args);
        }
}
@RestController
class ServiceInstanceRestController {

        @Autowired
        private DiscoveryClient;

        @RequestMapping("/service-instances/{applicationName}")
        public List<ServiceInstance> serviceInstancesByApplicationName(
                        @PathVariable String applicationName) {
                return this.discoveryClient.getInstances(applicationName);
        }
}
```

Listing 19 - Spring DiscoveryClient

Annotation SpringCloudApplication also includes EnableDiscoveryClient. A microservice will register itself by the Eureka server and can ask the registry for microservice instances.
The service registry implementation can be found in the YAML configuration files defined as spring: cloud: zookeeper or spring: cloud: consul.
Implementation of server-side discovery, where the registry takes care of routing the request to the appropriate microservice, is related to API gateway, which acts as a router for the incoming request. However, we did not encounter an implementation for server-side discovery.
Microprofile does not support service discovery in the specification. It can be used in combination with external frameworks that add support for it.

8.6 Database per Service implementations

The database per service pattern will add constraints to the Repository implementation, and the most important one is that microservices do not share a repository.
This can be avoided if:

- microservice and repository are part of the same package structure or
- microservice and repository are part of the same (Spring of Microprofile) application.

We will create constraint rules for the static code analysis tool so that we do not have to manually analyze the whitebox projects' source code.
When these two constraints are not met, it will not directly mean the database-per-service pattern is not used. Service-code and database-code (repository) may be split in the source code repository and deployment, but the database is not shared with other microservices.

If a database per service is used, it is also possible to use different database implementations per service. The core Spring and Spring Data frameworks used by the implementation of Repository (see 8.1.4) support popular database implementations like:

Database	Description
MongoDB	JSON-like documents distributed NoSQL database
Redis	In-memory key-value database and cache
Apache Cassandra	Distributed NoSQL (Wide-column) database
Elasticsearch	Distributed full-text search database
Neo4J	Graph database with Cypher query language
H2	Mainly used as an in-memory database (will not persist data on disk)
MySQL	Well known SQL database
DynamoDB	NoSQL database of Amazon

Table 20 - Database implementations supported by Spring (Data)

A Repository will use a specific database (see Table 20) implementation via several database frameworks, which will retrieve configuration from YAML- and property files. These files are very diverse, and in the static code analysis tool of Chapter 9, we will only use a few implementations we encountered in the selected OS projects.

8.6.1 Project SiteWhere

Project SiteWhere uses Rook.io[30], which supports data storage in a distributed environment managed by Kubernetes. In addition, Messaging provider (Kafka) and configuration management (Zookeeper) will use Rook.io as a manager for their databases.

[30] https://rook.io/

8.7 Caching implementations

Caching is used to decrease the number of calls from a microservice to another microservice or to the microservice's repository. Many implementations for caching mechanisms are available, but JSR 107 has defined a formal Java specification called JCache. Most implementations support this specification. Spring enables caching by using annotation EnableCaching on a Configuration class e.g. annotated with SpringBootApplication. Spring uses a concurrent hashmap as the default cache. By overriding cacheManager, another caching mechanism can be configured in the class file or XML configuration file. Annotation Cacheable is used on a method and means that the output values of the method are cacheable.
The only project that is using Spring Caching is the E-commerce app. The following example in Listing 11 shows an example of a SpringBootApplication with a method getAddress, which uses a cache for returning the address.

```
@SpringBootApplication
@EnableCaching
public class CachingApplication {

    public static void main(String[] args) {
       SpringApplication.run(CachingApplication.class, args);
    }
}

# On a service method
@Cacheable("addresses")
public String getAddress(Customer customer) {...}
```

Listing 20 - Spring Caching

Spring has support for the following cache providers:

Caching providers	
JCache (JSR-107) (Multiple implementations possible)	Couchbase
EhCache	Redis
Hazelcast	Caffeine
Infinispan	Simple

Table 21 - Spring supported caching providers

8.7.1 Project SiteWhere

Project SiteWhere uses EhCache, and the application logic will first query the cache for relatively static data before calling a microservice via gRPC.

8.7.2 Project FTGO

The FTGO-application uses messages (commands, replies and events) to communicate with other microservices (see Subchapter 8.3.2). A microservice can use the events to create a replica of the data of another microservice. Figure 17 shows a replica (dashed line) of consumers and restaurants in the Order service database.

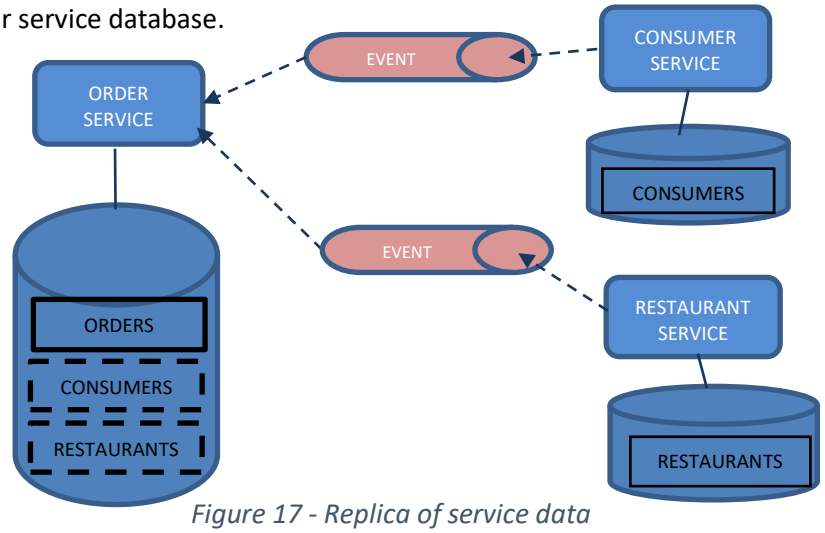


Figure 17 - Replica of service data

8.8 CQRS implementations

The CQRS pattern is only used in the projects FTGO- and Micro company.

8.8.1 Project FTGO

Microservices use commands and replies for 'CRUD' actions on other microservices, as described under Messaging (see Subchapter 8.3.2). This can be used for simple (Read) queries but is not always the best choice for complicated queries. Creating events (also called event sourcing) makes it possible to build a separate read-model when a change in a domain entity (a.k.a. the aggregate in DDD) takes place on a received command. Separating read and write models is called CQRS, which stands for Command Query Responsibility Segregation (see 6.7).
The Eventuate framework mentioned at the beginning of Chapter 8 is used by the FTGO-application to create a separate read-model for querying the order history. The order history service will subscribe to events of other microservices and stores them in its own database. The order history service provides a separate query endpoint for querying the order history data in its database.

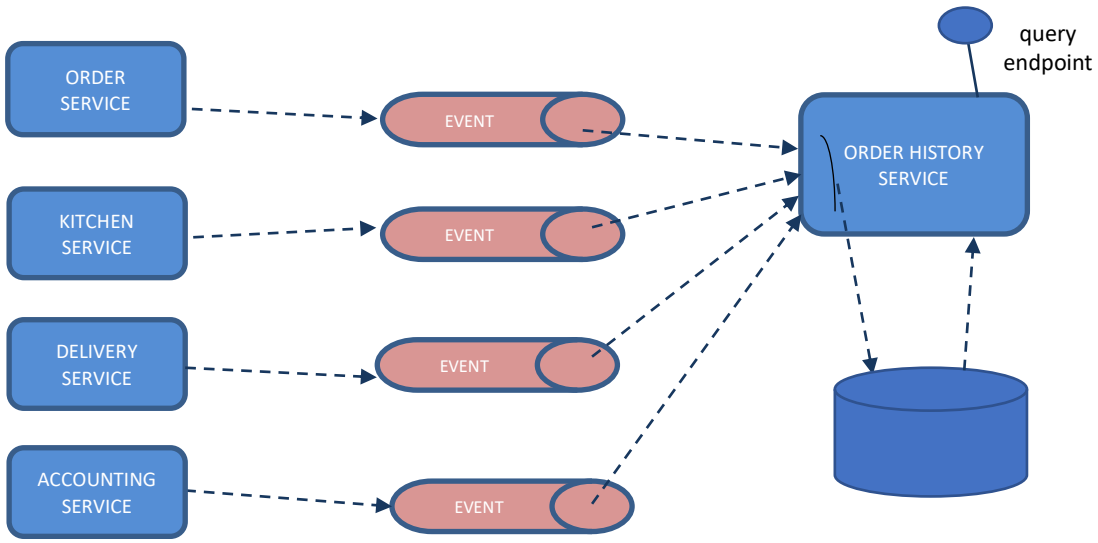


Figure 18 - Order History based on event sourcing

8.8.2 Project Micro company

The Micro company project creates events based on the Axon framework, also mentioned at the beginning of Chapter 8. The following events are published on receipt of four commands by the two aggregates (domain entity) 'blog' and 'project':

- BlogPost Created and BlogPost Published
- Project Created and Project Updated

These events are published, and two microservices (query-side-blog and query-side-project) will be subscribed to these events. The class EventHandler of the Axon framework handles the events and stores them in two read-model repositories for blogs and projects. There are no repositories for the write model in this project, which has to do with the demonstration purpose of this project.

9 Static code analysis tool for MSA patterns

The previous chapter describes the implementation of microservices, endpoints, repositories, clients and related MSA patterns for the communication and data-layer based on the OS whitebox projects. We also described how microservices retrieve their configuration as an import aspect of MSA-based applications. The possibility to use different database implementations is also what is to be expected in MSA-based applications, which was described as part of the database per service pattern.
One of the contributions of this thesis is a set of analysis programs containing Cypher queries that can detect and visualize the microservices and related MSA patterns in Java Open-Source projects. This requires a static code analysis tool that can read java source code files and configuration files like build scripts and property files.
Subchapter 7.2.4 described the tools JQAssistant and Neo4J, which scan the OS projects and run Cypher queries to detect and visualize MSA patterns.
The following paragraphs will describe the basis for the Cypher queries for microservices and the selected MSA patterns, which can be found in a file per MSA pattern (see Appendix C). In addition, Subchapter 7.2.4 explains the two steps which need to be executed to detect microservice concepts and related MSA patterns :

- Scan the project(s) jar files and additional configuration files
- Analyze the project(s) with the msa-pattern files

The description of the implementations in Chapter 8 found in the OS whitebox projects is the basis for the analysis programs. The programs will be executed on the selected OS projects (see 7.2.1) via the static code analysis tool to detect these projects' MSA concepts and patterns. Finally, the results are presented in an overview in Chapter 10.
The graphs shown below can also be created by JQAssistant and are available in the directory ‘pictures’ mentioned in Appendix B. They are modeled with the help of a style.grass file. See Appendix D for an example of the OS whitebox project PiggyMetrics.

9.1 Microservice, MsEndpoint, MsClient and repository nodes, relations and constraints

Before analyzing MSA patterns, we first have to find the implementations of microservices and related endpoints, clients, and repositories in a project's source code as described in Subchapter 8.1. Here we mentioned that the implementations are dependent on Spring and Microprofile frameworks. Except for the SiteWhere project, which has its own implementation of a microservice.

Table 22 shows an overview of the different types of a Microservice node, that is used in the static code analysis tool related to the concept of Microservice:

Type	Node Microservice with attribute serviceName
1	Class with annotation RestController (Spring Framework)
2	Class that extends class MicroserviceApplication (Own implementation of project SiteWhere)
3	Class with annotation Endpoint (Spring Framework)
4	Class with annotation Endpoint (Spring Boot)
5	Spring Controller-class with annotation RequestMapping (Spring Framework)
6	Interface with annotation Path (Microprofile)

Table 22 - Microservice types

Node MsEndpoint is used so that other microservices can access the microservice. The standard communication way for an endpoint is to use HTTP/REST. A REST endpoint uses a standard HTTP GET,

POST, or PUT method and can be called via an URL. The URL is defined in the source code and optionally in YAML configuration files. Table 23 shows an overview of the different types of a MsEndpoint node.

Type	Node MsEndpoint with attributes method, URL
1	Microservice method with annotation RequestMapping (Spring Framework)
2	Microservice method with annotation GetMapping (Spring Framework)
3	Microservice method with annotation PutMapping (Spring Framework)
4	Microservice method with annotation PostMapping (Spring Framework)
5	Microservice method with annotation PatchMapping (Spring Framework)
6	Microservice method with annotation DeleteMapping (Spring Framework)
7	Microservice method with annotation PayloadRoot (Spring Framework)
8	Microservice method with annotation ReadOperation (Spring Boot)
9	Microservice method with annotation WriteOperation (Spring Boot)
10	Microservice method with annotation DeleteOperation (Spring Boot)
11	Microservice method with annotation GET (Microprofile)
12	Microservice method with annotation PUT (Microprofile)
13	Microservice method with annotation POST (Microprofile)
14	Microservice method with annotation PATCH (Microprofile)
15	Microservice method with annotation DELETE (Microprofile)

Table 23 - MsEndpoint types

A microservice uses a so-called node MsClient to communicate with another microservice MsEndpoints. Table 24 shows an overview of the different types of a MsClient node.

Type	Node MsClient with attributes method, URL, usedServiceName
1	Interface annotation FeignClient (open implementation via Spring Cloud)
2	Interface annotation FeignClient (Netflix implementation via Spring Cloud)
3	Interface annotation FeignClient (other implementation via Spring Cloud)
4	Class with variable of type RestTemplate (Spring Framework)
5	Class which extends WebServiceGatewaySupport (Spring Framework)
6	Class with dependency on class RestClientBuilder (Microprofile)
7	Class with annotation RestClient (Microprofile)

Table 24 - MsClient types

The static code analysis tool will scan the source code for these three concepts: Microservice, MsEndpoint and MsClient and will label them with those names and add extra attributes.
A Microservice node has a standard (class-method of JQAssistant) relation DECLARES with a MsEndpoint node. The relation between a Microservice node and a MsClient node is created if there exists (directly or indirectly) a DECLARES or DEPENDS_ON relation between the two nodes. The relation is called USES_LOCAL_CLIENT.
A MsClient node has a URL and an (HTTP) method. If these match the URL and the method of a MsEndpoint node of a Microservice node, a relation INVOKES_REMOTE is created between the MsClient and the MsEndpoint nodes.

Another basic concept we need to define is the Repository node. We distinguish between a standard Repository, which is a part of a Microservice node and a MsRepository node, which is in itself a Microservice. A MsRepository node also has a relation to one or more MsEndpoint nodes. Table 25 shows an overview of the different types of Repository, MsRepository nodes and an additional MsEndpoint type related to MsRepository.

Type	Node Repository
1	Node with label Spring:Repository (Spring Framework)
2	Interface with dependency on GraphRepository (Neo4J) (Spring Data)
3	Interface with dependency on JpaRepository (Spring Data)
4	Node that extends class Persistence (Own implementation of project SiteWhere)
5	Node with parameter of type AggregateRepository (Own implementation of Eventuate framework)
6	Node that implements interface and depends on class DynamoDB (AWS DynamoDB)
Type	Node MsRepository
1	Interface with annotation RepositoryRestResource (Spring Data)
Type	Node MsEndpoint with attributes method, URL (see also Table 22 for node Microservice)
16	Interface with annotation RepositoryRestResource (Spring Data)

Table 25 - Repository, MsRepository and additional MsEndpoint types

A relation STORES_DATA will be created between node Microservice and node Repository. JQAssistant will already show which executable Jar-file contains the nodes mentioned above. A jar-file is a 'Jar' node, and it has a CONTAINS relation to the nodes belonging to the same jar.
Figure 19 shows an overview of all nodes, attributes and relations of microservices and repositories. The static code analysis tool will use the same colors for the recognisability of the discussed concepts.
The pictures created per project by Neo4J are similar to the picture shown in Figure 19. These pictures and the used Cypher programs are available in an attachment (see Appendix C).

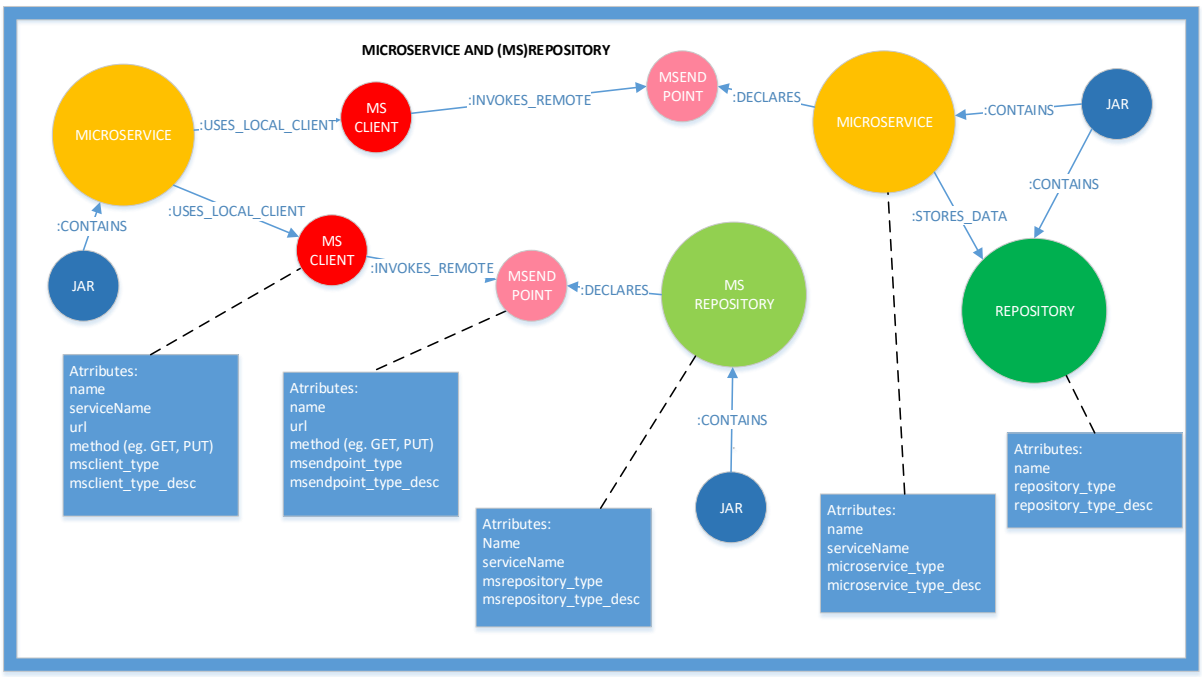


Figure 19 - Microservice, MsEndpoint, MsClient and (Ms)Repository nodes and relations

9.2 Microservice and MsClient constraints

We have defined the following constraints for microservices:

Code	Constraint name	Description
MSCON1	microservice-client-constraint	Check if all MsClients belong to a microservice
MSCON2	microservice-ms-constraint	Check if all microservices are Springboot or Microprofile applications
MSCON3	microservice-file-constraint	Check if there is a maximum of one microservice per file

Table 26 - Microservice constraints

9.3 MsConfiguration nodes

Most microservices-based applications use a standard (centralized) mechanism to retrieve the configuration for each microservice. Therefore, retrieving configuration can be seen as a different MSA pattern.
A dedicated microservice is used to manage the configuration of the other microservices. During start-up, each microservice retrieves its own configuration via the configuration-microservice.

We will create a node MsConfiguration in the static code analysis tool based on:

Type	Node MsConfiguration
1	Class with annotation EnableConfigServer (Spring framework)

Table 27 - MsConfiguration type

9.4 Gateway and ExtEndpoint nodes, relations and constraints

A gateway routes external requests via an external endpoint (ExtEndpoint) to MsEndpoints of internal microservices.

We will create a node Gateway and ExtEndpoint in the static code analysis tool based on:

Type	Node Gateway
1	Class with annotation EnableZuulProxy (Netflix via Spring Cloud)
2	Class with annotation EnableZuulServer (Netflix via Spring Cloud)
3	Class declares method with returntype RouteLocator (Spring Cloud)
4	Class with annotation OpenAPIDefinition (Microprofile)
Type	**Node ExtEndpoint with attributes fullUrl and method**
1	YAML configuration with key Zuul (Netflix via Spring Cloud)
2	Class ConsumerConfiguration of project FTGO with RouteLocator (Spring Cloud)
3	Class OrderConfiguration of project FTGO with RouteLocator (Spring Cloud)
4	Class annotated with Path (Own implementation of project Sitewhere)

Table 28 - Gateway and ExtEndpoint types

A relation CALLS is created between the ExtEndpoint- and Gateway-nodes.
An ExtEndpoint node has a URL and an (HTTP) method. If this matches the URL and the method of a MsEndpoint of a Microservice or a MsRepository, a relation ROUTES is created between the ExtEndpoint and the MsEndpoint nodes.

A gateway should not store data. A constraint (GWCON1) is used to check if a Gateway does not have a relation (relation depth=2) to a Repository or a MsRepository node.
An ExtEndpoint should not route directly to a MsEndpoint. A constraint (GWCON2) is used to check if these relations do not exist.

Code	Constraint name	Description
GWCON1	gateway–repository-constraint	A gateway should not have a repository (relation STORES_DATA)
GWCON2	gateway–client-constraint	An ExtEndpoint should not directly route to a MsEndpoint

Table 29 - Gateway constraints

The nodes, relations and constraints are shown in Figure 20. The static code analysis tool will not show the red ROUTES and STORES_DATA, because they should not exist. Only in case the constraint is broken, the arrow will be shown.

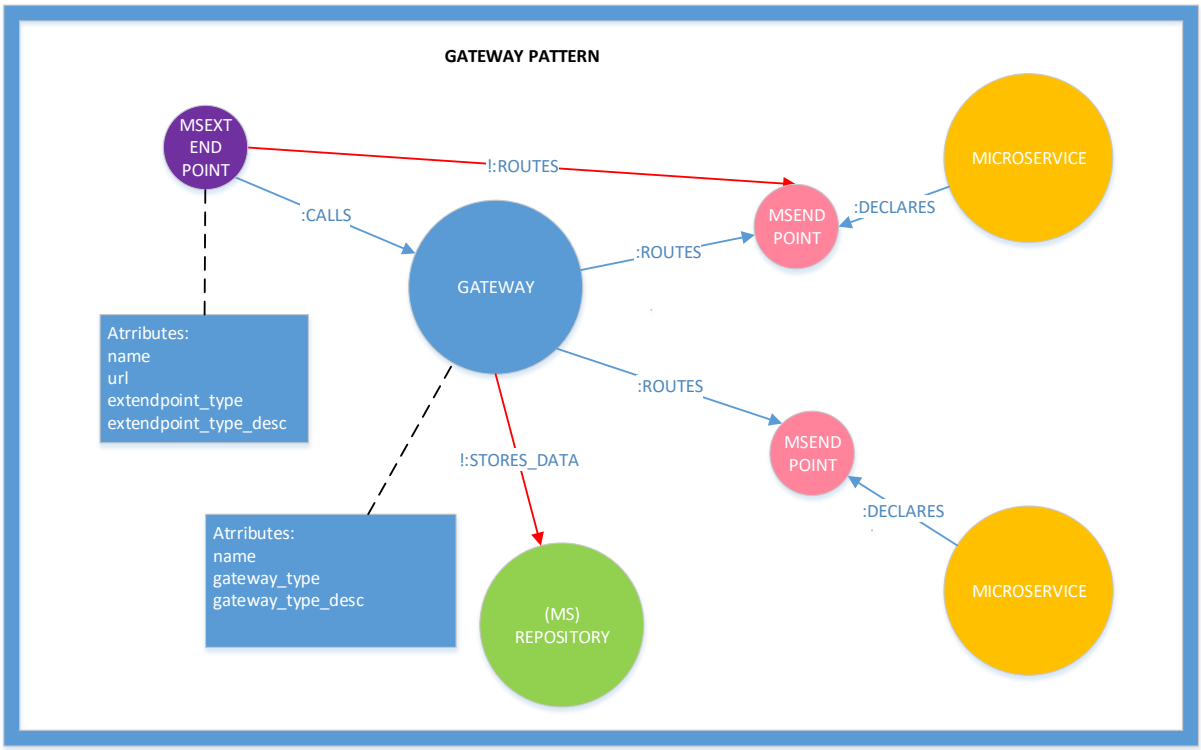


Figure 20 - Gateway and ExtEndpoint nodes and relations (Gateway pattern)

9.5 MessageBus, MsPublisher, MsSubscriber nodes and relations

The messaging pattern implementations are pretty diverse. The communication between microservices does not use the synchronous request/reply protocol of HTTP/REST with messaging. Instead, the communication is asynchronous via a lightweight message bus, decoupling the consumer and provider of the microservice.

We create the node labels MessageBus, MsPublisher and MsSubscriber as follows:

Type	Node MessageBus
1	Class with annotation EnableWebSocketMessageBroker (Spring Framework)
2	YAML configuration file with key Kafka (Apache framework)
3	YAML configuration file with key RabbitMQ (RabbitMQ framework via Spring AMQP)
4	Configuration class with dependency on ActiveMQ (ActiveMQ framework via Apache)
Type	**Node MsPublisher**
1	Microservice node with parameter depending on SimpMessagingTemplate (Spring Framework)
2	Class parameter with dependency on JmsTemplate (Spring Framework)
3	Class (Proxy) dependency on CommandEndpoint (Eventuate framework - command)
4	Class dependency on CommandHandlerReplyBuilder (Eventuate framework - reply)
Type	**Node MsSubscriber**
1	Class with annotation EnableWebSocketMessageBroker (Spring Framework)
2	Class method dependency on annotation JmsListener (Spring Framework)
3	Class (Proxy) dependency on CommandEndpoint (Eventuate framework - reply)
4	Class dependency on CommandHandlers (Eventuate framework - command)
5	Class with annotation RabbitListener (Spring AMQP)

Table 30 - MessageBus, MsPublisher and MsSubscriber types

Figure 21 shows the nodes MessageBus, MsPublisher and MsSubscriber with their relations:

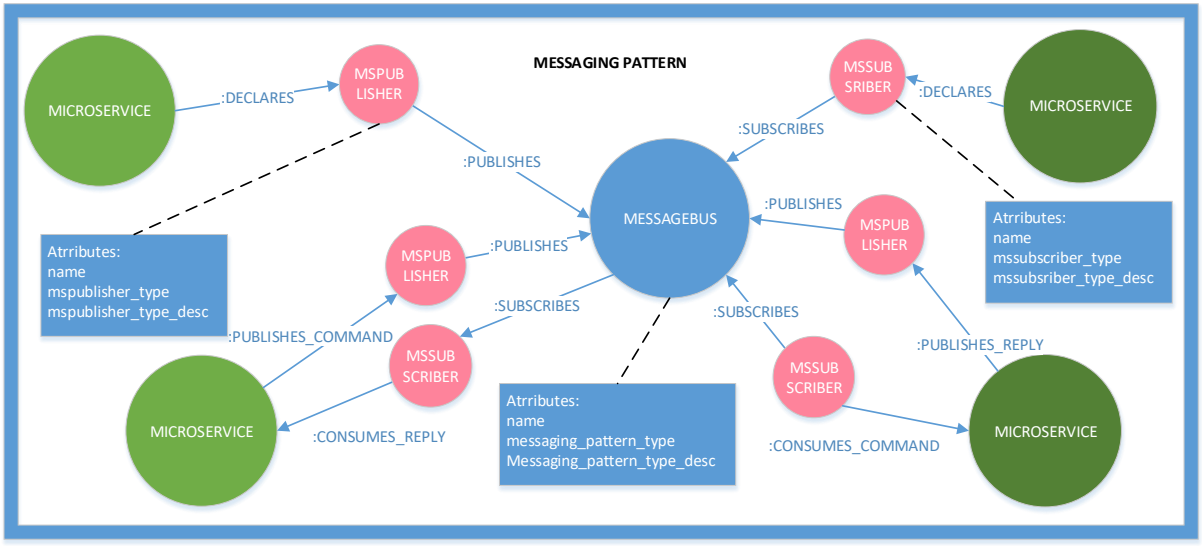


Figure 21 - MessageBus, MsPublisher and MsSubscriber nodes and relations (Messaging pattern)

We create the relations PUBLISHES and SUBSCRIBES between the MsPublisher/MsSubscriber and the MessageBus. A microservice has a PUBLISHES_COMMAND and PUBLISHES_REPLY relation with a MsPublisher and a CONSUMES_COMMAND and CONSUMES_REPLY relation with a MsSubscriber. COMMAND and REPLY are only used for the FTGO-application.

9.6 Circuit breaker relation

We model a circuit breaker as a relation of a node to itself. It will be detected by one of the following criteria:

Type	Relation CIRCUITBREAKER
1	Method with annotation FeignClient (Spring Cloud)
2	Method with annotation HystrixCommand (Netflix)
3	Microservice node with annotation SpringCloudApplication (Spring Cloud)
4	Class with annotation EnableHystrix (Spring Cloud)

Table 31 - Circuitbreaker types

Figure 22 shows the circuit breaker relation:

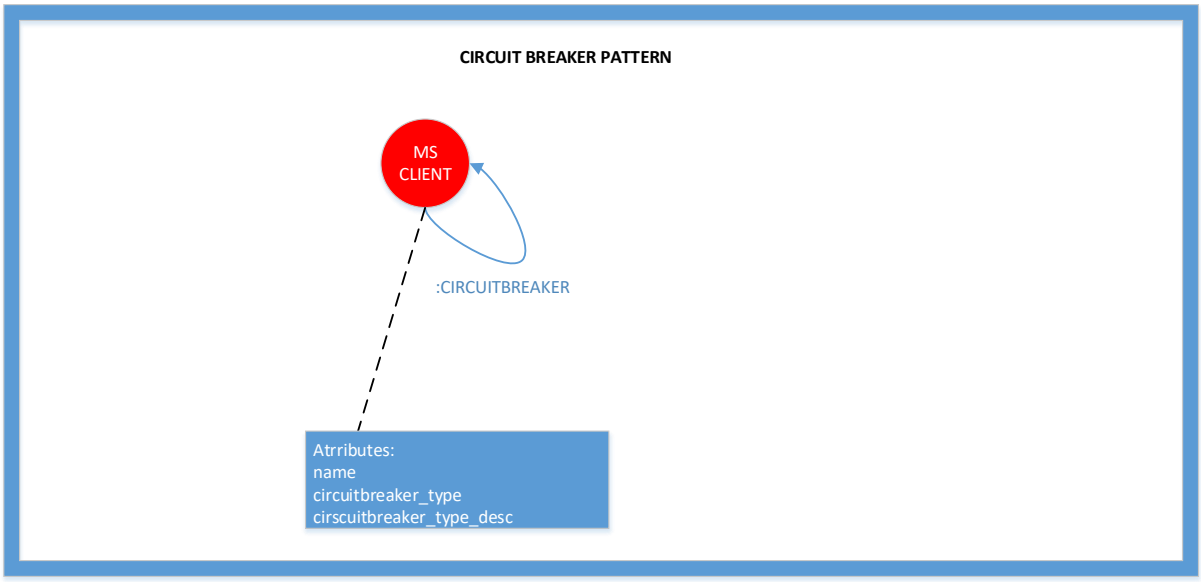


Figure 22 - Circuitbreaker pattern

9.7 DiscoveryClient and DiscoveryService nodes, relations and constraint

For the Service Discovery pattern, we need to set DiscoveryClient and DiscoverServer nodes.

Type	Node DiscoveryClient
1	Microservice, MsRepository, Gateway, ConfigService with annotation EnableDiscoveryClient (Netflix framework via Spring Cloud)
2	Microservice, MsRepository, Gateway, ConfigService with annotation EnableEurekaClient (Netflix framework via Spring Cloud)
3	Microservice, MsRepository, Gateway, ConfigService with annotation SpringCloudApplication (Netflix framework via Spring Cloud)
Type	Node DiscoveryService
1	Class with annotation EnableEurekaServer (Netflix framework via Spring Cloud)
2	YAML configuration with key Zookeeper (Apache framework)

Table 32 - DiscoveryClient and DiscoveryService types

Each DiscoveryClient will register itself by the DiscoveryService and is modeled as a REGISTERS relation. MsClient nodes have a usedServiceName attribute, and if this matches a serviceName of a Microservice or a MsRepository, then a LOOKS_UP relation is drawn between the MsClient and the DiscoveryService.
There is a constraint (DISCON1) to check that all MsClient nodes have a LOOKS_UP relation.

Code	Constraint name	Description
DISCON1	discovery–constraint	All MsClients have a LOOKS_UP relation

Table 33 - Service Discovery constraint

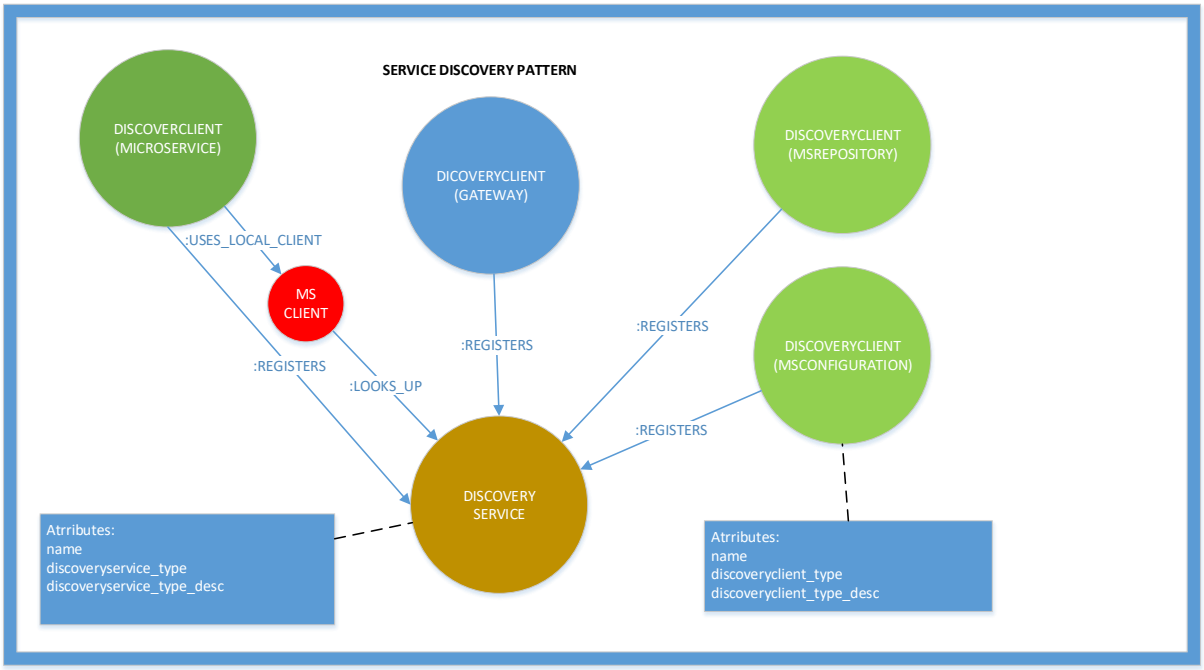


Figure 23 - DiscoveryClient and DiscoveryService nodes and relations (Service Discovery pattern)

9.8 Database per Service constraints and database types as relation

The database per service pattern does not need extra concepts.
We will add constraints to check that:

- Microservice and repository are part of the same package (DBCON1)
- Microservice and repository are part of the same file (DBCON2)
- A Repository should have one relation STORES_DATA to a Microservice and not also to another Microservice (DBCON3)

Code	Constraint name	Description
DBCON1	microservice-repository-same-package	a microservice only uses a repository in the same package
DBCON2	microservice-repository-same-jar	a repository and a microservice are part of the same springboot-file
DBCON3	microservices-do-not-share-repository	a repository is not shared with other microservices

Table 34 - Database constraints

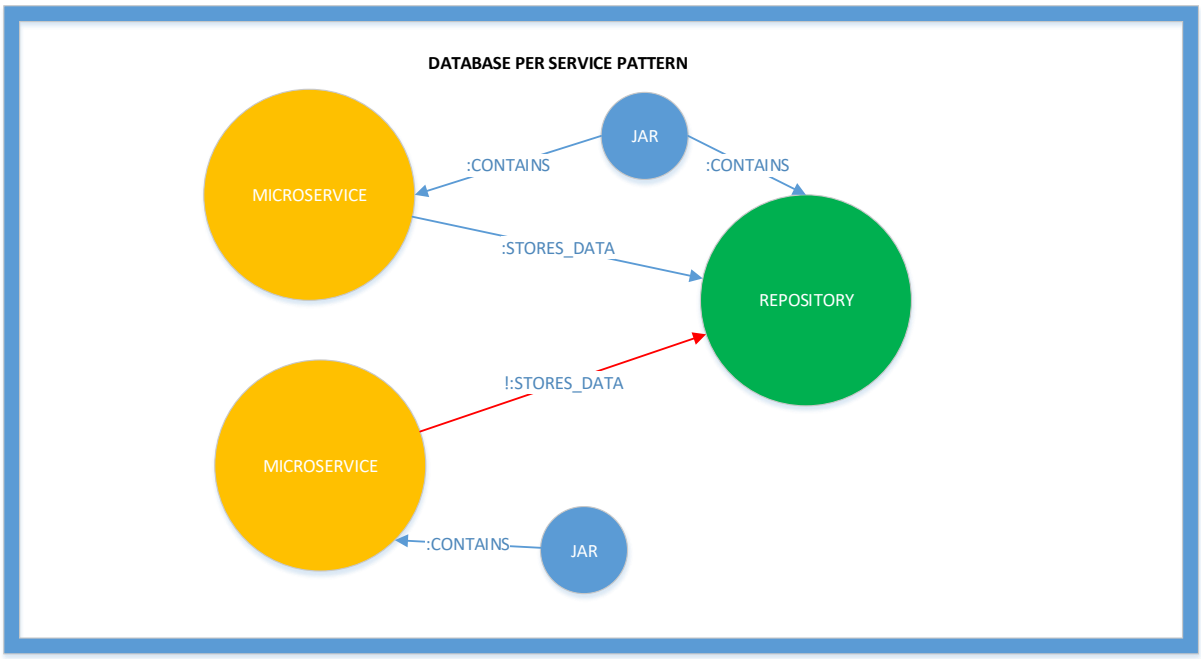


Figure 24 - Database per Service pattern

We will also model a database type as a relation to a Repository node. The database type is not part of the pattern but will show that multiple implementations of databases can be used in an MSA-architecture

Type	Relation MONGODB
1	Entity with annotation data.mongodb.core.mapping.Document (via Spring Data)
2	Field of type com.mongodb.async.client.MongoDatabase (MongoDB)
3	Repository type starts with com.mongodb (MongoDB)
Type	Relation REDISDB
1	Field of type redis.core.RedisTemplate (via Spring Data)
Type	Relation NEO4JDB
1	Node with annotation EnableNeo4jRepositories (Spring Data)
Type	Relation H2DB
1	Property file has jdbc.h2
2	Pom.xml has a dependency on com.h2database
Type	Relation MYSQLDB
1	Property file has property ds.name = spring.datasource.url
Type	Relation DYNAMODB
1	Property file has property ds.name = aws.dynamodb.endpoint.url

Table 35 - Database types

9.9 Caching relation

We model the Caching pattern in the same way as a Circuit breaker, as a relation CACHING to itself. In this case, the node is a Repository.

Type	Relation CACHING
1	Class with annotation EnableCaching (Spring Framework)

Table 36 - Caching type

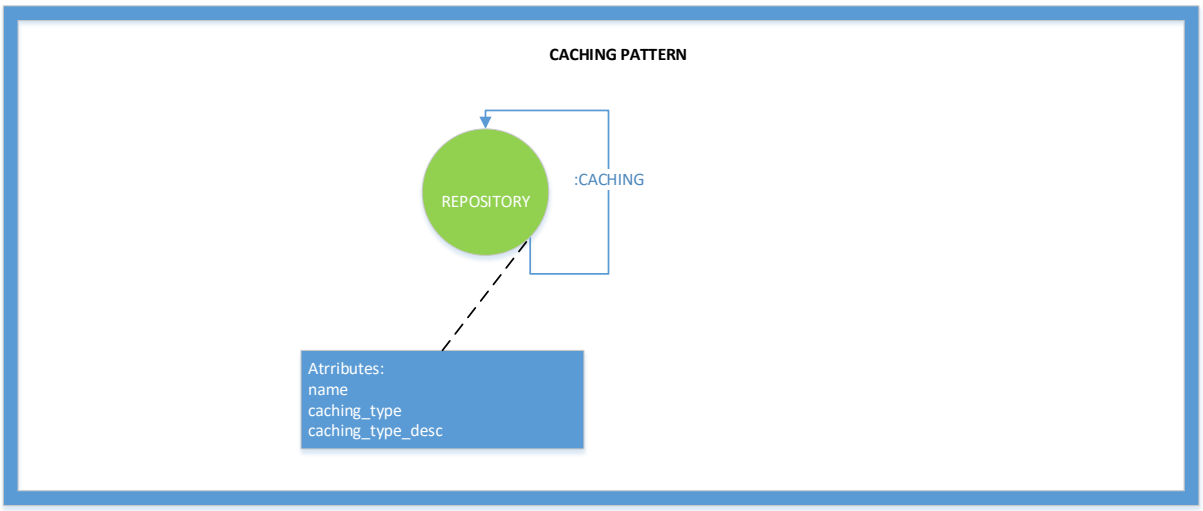


Figure 25 - Caching pattern

9.10 MsPublisher and MsSubscriber nodes and relations for CQRS

CQRS (Command Query Responsibility Segregation) pattern is dependent on the messaging pattern. The messaging pattern defines publishers and subscribers of a message bus. They will exchange commands and replies via the message bus. The CQRS pattern will add the publishing of events and the consumption of these events via the message bus. We will model this also as a MsPublisher and MsSubscriber. The MsPublisher will create a PUBLISHES_EVENT relation between a node and itself. The MsSubscriber will create a CONSUMES_EVENT between itself and another node.

Type	Node MsPublisher
1	Class (Proxy) dependency on AbstractAggregateDomainEventPublisher or DomainEventPublisher (Eventuate framework)
2	Class declares method with annotation CommandHandler (Axon framework)
Type	**Node MsSubscriber**
1	Class declared method of type DomainEventHandler (Eventuate framework)
2	Class declared method with annotation EventHandler (Axon framework)

Table 37 - CQRS MsPublisher and MsSubscriber types

Figure 26 shows how the MsPublisher has a relation PUBLISHES_EVENT and MsSubscriber has a relation CONSUMES_EVENT.
Projects FTGO and Micro company use the CQRS pattern.

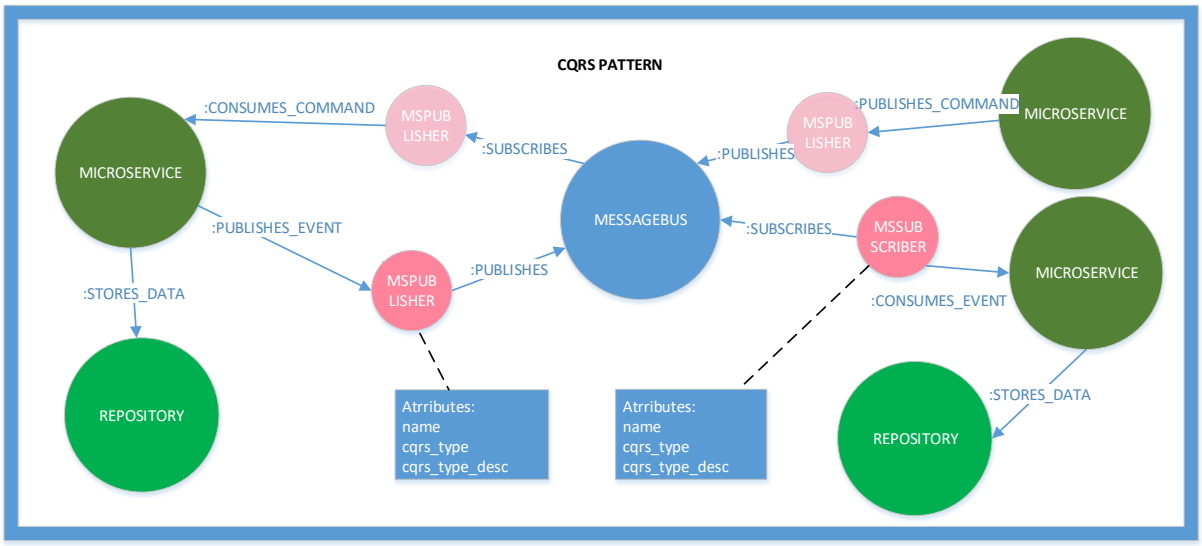


Figure 26 - MsPublisher and MsSubscriber with CQRS pattern

10 Results of Phase 2 (RQ3)

Chapter 8 describes the implementations of MSA patterns found in the ten selected whitebox OS projects. That description was the input for creating a static code analysis tool, described in Chapter 9, which is able to query for these implementations of MSA patterns in Java projects.
This chapter shows in Subchapter 10.1 an overview of the MSA patterns found in the ten whitebox OS projects. This overview and the named type of implementation, based on Chapter 9, answer research question RQ3.
In a second run, we also executed the static code analysis tool on the blackbox OS projects. The result of this run is presented in Subchapter 10.2. A summary of the results of the whitebox and blackbox projects in Subchapter 10.3 will be used in Chapter 11, where we discuss the results of Phase 1 and Phase 2 of the thesis. Finally, we will make some remarks in Subchapter 10.4 about the blackbox OS projects, indicating the general usability of the MSA queries in the static code analysis tool at this stage.

10.1 MSA patterns in ten whitebox OS projects

We have scanned ten whitebox OS projects with JQAssistant and loaded them into a Neo4J database. This resulted in 271.000 nodes and 809.000 relations detected by the standard rules defined in JQAssistant and the extension made for the MSA patterns (see Appendix C).
Table 38 shows an overview of all concepts per whitebox project detected by the static code analysis tool. The first number is the type of concept as defined in Chapter 9. The second number is the number of occurrences of that type. As an example, PiggyMetrics has four implementations of type 1 for the concept microservice. Project Sitewhere has much specific code for several concepts (see X in the table); thus, we did not program Cypher queries for them.

Concept / Project	Piggy Metrics		FTGO		E-Com App		Blog post		Tap &Eat		Delivery		Graph Proc.		Lake side		Micro Comp		Site Where	
Microservice	1	4	1	7	1	2	1	3	1	1	1	2	1	1	1 3	14 1	1	2	2 6	15 27
MsEndpoint	1	11	1	13	1	9	1	7	1 16	5 5	1	8	1 16	1 1	2 3 4 5 6	20 5 7 3 1	1 16	4 2	11 12 13 15	110 20 48 23
MsClient	1	4	-	-	-	-	2	8	2	2	2	1	4	1	1 4 5	3 2 1	-	-		
MsRepository	-	-	-	-	-	-	-	-	1	5	1	1	1	1	-	-	1	2		
Repository	1	4	1 5 6	6 1 1	1	2	-	-	1	5	1	1	2	2	3	6	1	4	4	8
Gateway	1	1	3	2	1	1	1	1	-	-	1	1	1	2	-	-	1	1	4	1
ExtEndpoint	1	4	2 3	2 2	1	2	1	3	-	-	-	-	-	-	-	-	1	4		
MessageBus	-	-	2	1	-	-	-	-	-	-	1	1	1	3	1 4	1 2	1 3	1 1	X	X
MsSubscriber	-	-	3 4	4 4	-	-	-	-	-	-	1	1	5	1	1 2	1 5	1	1	X	X
MsPublisher	-	-	3 4	4 4	-	-	-	-	-	-	1	1	-	-	1 2	1 3	-	-	X	X
Circuit breaker	1	4	-	-	-	-	1 2	2 1	1	2	1 2	1 1	3	1	1 2	3 1	4	1		
DiscoveryClient	1	4	-	-	1	2	1	3	1	7	2	4	1	1	-	-	1	6	X	X

DiscoveryService	1	1	1	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	X	X
CQRS write			1	4													2	2		
CQRS read			1	4													2	2		
ConfigService	1	1	-	-	1	1	1	1	1	1	1	1	1	1	-	-	1	1	X	X
Caching	-	-	-	-	1	1	-	-	-	-	-	-	-	-	-	-	-	-	X	X
DB (Mongo)	1	4	-	-	3	1	-	-	-	-	-	-	-	-	-	-		2	X	X
DB (Redis)	-	-	-	-	1	1	-	-	-	-	-	-	-	-	-	-				
DB (Neo4J)	-	-	-	-	-	-	-	-	-	-	-	-	1	2	-	-				
DB (H2)	-	-	-	-	-	-	-	-	2	5	2	1	-	-	2	6				
DB (MySQL)	-	-	1	6	-	-	-	-	-	-	-	-	-	-	-	-				
DB (Dynamo)	-	-	1	1	-	-	-	-	-	-	-	-	-	-	-	-				
DB (Cassandra)																			X	X

Table 38 - MSA concepts and patterns in whitebox OS projects

Table 39 shows the constraints, which were hit for the whitebox OS projects. The DBCON3 is directly related to the database per service pattern and shows the projects where multiple microservices use the same repository.

Constraint	Description	Projects
MSCON1	All MsClient nodes belong to a Microservice node	PiggyMetrics, Graph Proc., Lakeside
MSCON2	All Microservice nodes are Spring Boot or Microprofile applications.	FTGO, Micro Comp
MSCON3	Only one microservice per jar-file	FTGO, Lakeside
GWCON1	A Gateway does not have a relation (relation depth=2) to a (Ms)Repository	Delivery, Graph Proc., SiteWhere
GWCON2	An ExtEndpoint should not route directly to a MsEndpoint	
DISCON1	All MsClient nodes have a LOOKS_UP relation	PiggyMetrics, Graph Proc. , Lakeside
DBCON1	A Microservice uses a Repository in the same package (depth=3).	FTGO
DBCON2	Microservice and Repository belong to the same jar (SpringBoot or Microprofile Application).	
DBCON3	A Repository is not shared with other Microservices	FGTO, Delivery, Lakeside

Table 39 - Constraints found in the whitebox OS projects

10.2 MSA patterns in ten blackbox OS projects

We scanned ten additional blackbox OS projects with JQAssistant and loaded them into a Neo4J database for validation purposes. Table 40 shows an overview of all concepts per project detected by the static code analysis tool. The first number is the type of concept as defined in the previous paragraphs. The second number is the number of occurrences of that type. As an example, CAS has two implementations of type 6 for the concept microservice.

Concept / Project	CAS		Freddy bbq		Genie		micro Service		ms book		Movie recom		Netflix micro		Share bike		Task track		Ali Baba	
Microservice	6	2	1	1	14 61		1	13			1	4	1	2	1	3	5	7	14	289
MsEndpoint	11	2	1	2	23 45 68 9	33 16 17 3 25 1 1	1	32	16	3	12	122	1	2	24 416	441	1	46	12 48	964 69
MsClient					4	3	24	64	4	2	24	61	1	1	4	6	4	4	14	2211
MsRepository									1	3					1	1				
Repository			1	2	13	91			1	3	23	21			13	43			1	1
Gateway									1	1	1	1	1	1	1	1			1	1
ExtEndpoint													1	2						
MessageBus													3	1	3	1				
MsSubscriber																				
MsPublisher																				
Circuit breaker			2	10			124 631		2	3	12	61	124	1111	2	6			13	221
DiscoveryClient			1	5			12	61	1	6	12	61	12	24	1	8			13	291
DiscoveryService			1	1			1	1	1	1	1	1	1	1	1	1				
CQRS write																				
CQRS read																				
ConfigService							1	1			1	1	1	1					1	1
Caching																				
DB (Mongo)																				
DB (Redis)																				
DB (Neo4J)																				
DB (H2)			2	2																
DB (MySQL)																				
DB (Dynamo)																				

Table 40 - MSA concepts and patterns in blackbox OS projects

Table 41 shows the constraints, which were hit for the validation projects. The DBCON3 is directly related to the database per service pattern and shows the projects where multiple microservices use the same repository.

Constraint	Description	Projects
MSCON1	All MsClient nodes belong to a Microservice or MsRepository node	Movie rec., Share bike, Alibaba
MSCON2	All Microservice nodes are Spring Boot or Microprofile applications	Genie, Task track, Alibaba
MSCON3	Only one microservice per jar-file	Genie, microService, Task track, Alibaba
GWCON1	A Gateway does not have a relation (relation depth=2) to a (Ms)Repository	
GWCON2	An ExtEndpoint should not route directly to a MsEndpoint	
DISCON1	All MsClient nodes have a LOOKS_UP relation	Genie, microService, ms book, Movie recom., Netflix micro, Share bike, Task track, Alibaba
DBCON1	A Microservice uses a Repository in the same package (depth=3).	
DBCON2	Microservice and Repository belong to the same jar (SpringBootApplication).	
DBCON3	A Repository is not shared with other Microservices	Freddy's BBQ, Movie recom., Share bike, Alibaba

Table 41 - Constraints found in validation projects

10.3 Summary of MSA patterns in all OS projects

Table 42 shows an overview of the patterns found in the whitebox and blackbox projects.
Five of the seven patterns are frequently (>60%) used in the whitebox projects. The score of the blackbox projects is lower but shows the same distribution, except for the messaging pattern. We will discuss the result of Phase 1 and Phase 2 in more detail in Chapter 10.4.

Category	MSA pattern	Usage whitebox projects	Usage blackbox projects	Remarks about the nodes or relations found via the static code analysis tool.
Communi-cation	API gateway	80%	50%	The project has node Gateway. Most implementations use ZuulProxy from Netflix via Spring Cloud.
	Messaging	60%	20%	The project has nodes MessageBus, MsPublisher, or MsSubscriber.
	Circuit breaker	70%	70%	The project has a relation Circuitbreaker.
	Service discovery	100%	70%	The project has nodes DiscoveryClient or DiscoveryService. Most projects use the Netflix implementation via Spring Cloud.
Data layer	Database per server	70%	60%	The project has not hit constraint DBCON3.
	Cache	20%	0%	The project has relation Caching.
	CQRS	20%	0%	The project has nodes MsPublisher or MsSubscriber with CQRS type

Table 42 - Summary of MSA patterns in selected OS projects

10.4 Investigation of the OS blackbox results

The result of the blackbox projects is lower for most of the patterns. Therefore, we searched in these projects for MSA pattern keywords mentioned in Chapter 6 and recorded our findings in Table 43.

Category	MSA pattern	Remarks about the blackbox projects
Communi-cation	API gateway	We only encountered that project Task has its own implementation of a Gateway and is not part of one of our queries.
	Messaging	Project Alibaba mentions RocketMQ as messaging platform on its GitHub page. We did not include that in our queries.
	Circuit breaker	Project Task track uses Hystrix for Circuit breaker but did not use the standard Spring annotation for it, but an import to the old Netflix library.
	Service discovery	The score is good; no extra investigation was done.
Data layer	Database per server	The score is good; no extra investigation was done.
	Cache	Searching for the word Caching reveals that project Genie uses an implementation of AWS ElastiCache[31]. We did not include that in our queries.
	CQRS	The score CQRS is also low for the whitebox projects, and searching for the word CQRS in the blackbox projects did not give any hit.

Table 43 - Remarks about the MSA patterns in the OS blackbox projects

We investigated one extra project CAS, because it was one with a very low score on Microservice concepts and MSA patterns. The microservice's clients are based on an (and not in our static code analysis tool included) Apache REST client implementation, which needs much coding compared with Spring's FeignClient. It has no Circuitbreaker, no Service Discovery, no Messaging, no Gateway, no Repository. So we can conclude that it is not a mature MSA project, which was also indicated by our overview in 10.2

[31] https://aws.amazon.com/elasticache/

11 Discussion

In this chapter, we will compare the usage of the MSA patterns in the literature sources of Phase 1 and the OS projects of Phase 2 and we will come back to the Agile Manifesto mentioned in the introduction. Finally, at the end of this chapter, we will discuss the threats to the validity related to the found results.

11.1 Usage of selected MSA patterns

Table 44 shows the two categories, communication and data-layer, with the seven MSA patterns. It summarizes the percentages of sources describing the MSA patterns for the five literature sources (see 5.3) and the percentages of projects implementing the MSA pattern in the ten OS whitebox and ten OS blackbox projects (see 10.3). For example, the percentages for the API gateway mean that all (100%) selected literature sources mention the API gateway, eight out of ten (80%) whitebox projects and five out of ten (50%) blackbox projects implemented the pattern.

Category	MSA pattern	Lite-ratury sources	Usage whitebox projects	Usage blackbox projects	Remarks
Communi-cation	API gateway	100%	80%	50%	Important MSA pattern for communication with external systems.
	Messaging	60%	60%	20%	Messaging implementations are very diverse.
	Circuit breaker	60%	70%	70%	Circuitbreaker is used in synchronous communication.
	Service discovery	100%	100%	70%	Important MSA pattern for communication with the external systems.
Data layer	Database per server	100%	70%	60%	A key aspect of microservices, but database implementations are diverse.
	Cache	20%	20%	0%	Not often used at application level.
	CQRS	20%	20%	0%	A newer pattern, not often used in small projects.

Table 44 - Usage of MSA patterns in selected Literature sources and OS projects

The following subchapters will discuss the MSA patterns usage for the Communication and Data-layer categories in more detail.

11.1.1 MSA patterns of the communication category

We see for the communication category that most sources and OS whitebox projects have described or implemented the API gateway and service discovery patterns. So we could say that these two are key patterns for MSA.

The messaging and circuit breaker patterns have a lower score, for which we give two reasons. The first reason is that MSA does not introduce these patterns, as Márquez et al. [3] mentioned, and thus left out by some sources. We disagree with Márquez et al. [3] that these patterns are the same as the SOA patterns because the implementations are different, like the circuit breaker donated by Netflix and the usage of a lightweight message bus instead of an enterprise service bus.

The second reason for a lower score for the circuit breaker and messaging implementation in the OS projects is that a project which uses messaging as the key communication method (e.g. FTGO application) does not need the circuit breaker, and a project that uses synchronous communication without messaging will use the circuit breaker as a means for resilience and error handling.

The OS blackbox projects' score is similar except for the messaging, which is much lower. An explanation is that the messaging pattern's implementation is diverse and dependent on configuration files, which are harder to analyze.
The static code analysis tool for MSA patterns (API gateway, Circuit breaker and Service Discovery), which depend on source code annotations, is more reliable than the tool is for the MSA patterns, which depend on external configuration files.

For (one of the more mature) FTGO application, we noticed that the MsClient, DiscoveryClient and Ciricuitbreaker nodes are missing in Table 38. We assume that the message-driven nature of this application does not require the MsClient and Circuit breaker concepts. DiscoveryClient is not part of their source code but is configured during deployment.

11.1.2 MSA patterns of the Data-layer category

The Database per Service is a key pattern, as we can see from the percentages score of the literature study and the implementations in the OS projects.
Although they are not MSA patterns, we cannot detect many database types (e.g. like H2 of MySQL) at the moment as part of the Database per Service pattern. One of the reasons is that there is a lot configured in deployment scripts, which are not part of the current scanning by JQAssistant for this thesis. Of course, it is possible to scan these files with JQAssistant, but there is much diversity in these files, and it will not contribute a lot to analyzing patterns. In the previous subchapter, we also mentioned that detecting concepts via annotations is more reliable than via external configuration files.
We see that the Caching and CQRS patterns are not often described or implemented for the data-layer category. For Caching, a reason could be that the Java projects are server-side implementations and the pattern description has more focus on the client-side. CQRS is a newer pattern, as mentioned in 5.2, and not always necessary in a small project, hence the few implementations.

11.2 MSA patterns relating to the Reactive Manifesto

Looking back at the Reactive Manifesto mentioned in the introduction, we think the seven MSA patterns described in this thesis can be linked to the manifesto, as shown in Table 45.

Key level characteristic	MSA pattern
Responsive	Caching, CQRS
Elastic	API gateway, Service Discovery, Database per service
Resilient	Circuit breaker
Message-driven	Messaging, CQRS

Table 45 - Reactive Manifesto and MSA patterns

Caching (avoiding extra round trip delay) and CQRS (model optimized for reading) will contribute to the characteristic Responsive. However, only Márquez et al. [3] name Caching as an architectural pattern and the description of the pattern is not very comprehensive. Furthermore, Caching has a relation with CQRS, for which a replica is created for the read model, which can be seen as a form of Caching (see 8.8.1).
The container hosting infrastructure for MSA enables the elasticity (scaling) requirements as mentioned in Subchapter 3.1. In addition, the patterns API gateway, Service Discovery, and Database per Service will ensure that MSA applications effectively use this underlying hosting environment.
The Circuit breaker pattern is implemented to deal with failures in synchronous communication and contributes to the resilient characteristic.
Messaging and CQRS (with commands and events) contribute to the Message-driven characteristic.

11.3 Implementations of other MSA patterns

MSA and its patterns are much broader than we could address in this thesis, which was already mentioned in Chapter 4, where we looked into a wide range of architectural patterns in the literature. We acknowledge this during the analysis of the MSA implementations in Chapter 8. We encountered a lot of coding and configuration used for the deployment and management of microservices in production. The related patterns are used for implementing scaling, deployment, administration, fault detection, health monitoring and traceability. Netflix Hystrix, which is used for the circuit breaker patterns, also supports monitoring (e.g. see projects PiggyMetrics, Blog post, Tap and Eat, Micro Company) with a Hystrix or Turbine dashboard. Each microservice supporting Hystrix can stream data to a microservice annotated with EnableHystrixDashboard, EnableTurbineStream or EnableTurbineAmqp, which creates a ready-made dashboard for monitoring the errors detected by Hystrix. Microprofile also supports many features for health monitoring and metrics. Some projects (Lakeside, MicroCompany) use a microservice with annotation EnableAdminServer, enabling a ready-made dashboard showing all Spring Boot applications' status.

11.4 Threats to validity

This subchapter described the threats to the internal and external validity of the results of this thesis.

11.4.1 Internal validity

11.4.1.1 Limited number of literature sources

Appendix A and Appendix B show the papers used in the two steps of this thesis's literature study. We eventually selected four academic articles and one website to create the descriptions of the MSA patterns. However, three of these papers [1, 3, 4] also include a mapping study and have many references to other papers (see Appendix B). This means that many more papers were used than the selected five sources.

11.4.1.2 Limited number of OS projects

The thesis is restricted to twenty Open-Source Java projects in GitHub, and most of the used projects are related to learning and demonstration purposes (except SiteWhere). Although the thesis was already limited to seven architectural patterns, we see a significant variation in implementations, as shown in the number of Cypher queries used together with the Neo4J Graph database.
We used the selection criterium of two sources [3] [45] as described in Subchapter 7.2.1.
Table 46 gives an overview of the no. of microservices per project, and it is comparable with the no. of microservices found in the literature study by Francesco et al. [1]. Therefore, the no. of microservices per project used in this thesis is in line with the cited literature study.

No. of microservices	Project(s)
1 (3 projects)	Tap-And-Eat, Graph Processing, Freddy's bbq
2 (4 projects)	E-Commerce App, Delivery system, Cas, Netflix microservice
3 (4 projects)	Blog post, E-Commerce App, ms book, Share bike
4 (2 projects)	Piggy Metrics, Movie recommendation
7 (3 projects)	FTGO Application, Genie, Task track support
13 (1 project)	microService
15 (1 project)	Lakeside
37 (1 project)	AliBaba
42 (1 project)	SiteWhere

Table 46 - No. of microservices in the projects

Analyzing Open-Source projects has the disadvantage of concentrating less on non-functionals (error handling, performance) required for commercial applications. Therefore, this thesis does not include feedback on the described implementation of the MSA patterns in this area.

11.4.1.3 Results of OS blackbox projects

The results of the OS blackbox projects are not verified by analyzing the complete source code of these projects.
We do not think that running the static analysis tool will lead to false positives for these projects. The terms used in the Cypher queries are not very general and if needed the fully qualified name (e.g. fqn=org.springframework.cloud.gateway.route.RouteLocator) was specified.

We described some remarks about the reason for a lower score compared with the OS whitebox projects in Subchapter 10.4. We already mentioned that we think that the current queries will have false negatives. The enormous variations in configuration files used for detecting e.g. Messaging and Database types could cause these false negatives.

11.4.1.4 Limitations of JQAssistant for code analyzing

JQAssistant cannot read the parameters of a method. We raised a question on a JQAssistant forum, and the answer was that it requires much extra work, and they want to focus on the architectural level. The usage of a method's parameters can improve Cypher queries' usability, but it was not a major issue for this thesis.

11.4.2 External validity

We have created files to set up an environment with JQAssistant and Neo4J and loaded the projects into the Neo4J database. The Cypher queries are stored in a separate file for each microservice concept and pattern. The files are listed in Chapter 17 (JQAssistant analysis programs) and are available as an appendix to this thesis.
These files enable other people to quickly execute the JQAssistant queries and verify the results of this thesis.

12 Conclusion and future work

12.1 Conclusion

This thesis investigates architectural patterns related to communication and data-layer, as described in the literature and implemented in Open-Source Java projects.
The patterns API gateway, Service Discovery, Messaging, Circuitbreaker and Database per Service are a solid foundation for an application based on microservices. In addition, enhancements in performance can be achieved by patterns Caching and CQRS.
We confirmed that the API Gateway, Service Discovery and Database as Service are the three key MSA patterns in literature and implementation. Messaging and Circuit breaker are patterns for supporting respectively asynchronous and synchronous communication and are not always used in combination. CQRS and the briefly mentioned Domain-Driven Design (DDD) are newer developments supporting MSA.
Patterns are frequently used in the software by putting annotations in the source code and are implemented by the (Open-Source) frameworks defined in the build files of maven and gradle. JQAssiatant and Neo4J as a static code analysis tool make it possible to create a graph overview of an MSA application by drawing the concepts (e.g. microservice, repository, MSA pattern) described as nodes and relations and checking constraints for an MSA pattern.
We encountered that detecting MSA patterns via standard annotations is easier and more widely applicable than via a project's own coding implementations or all kinds of external configuration files. The Java language and additional frameworks should support this approach to implement MSA patterns for developers quickly. The wide adaption of the Spring frameworks with the Netflix donation and Microprofile frameworks in the OS projects shows that this is very successful for MSA.
Patterns in the other areas, like those mentioned in Subchapter 11.3 (scaling, deployment administration, fault detection, health monitoring, traceability), should be added to design an MSA reference architecture. (IBM provided such a reference architecture[32], which shows the importance of the seven MSA patterns described in this thesis.)

12.2 Future work

The querying for MSA patterns in several OS projects using a Neo4J Graph database is a good combination. The JQAssistant scan and analysis phases performed well with all ten projects loaded into the database. This offers opportunities to investigate whether the combination JQAssistant and Neo4J can be used to analyse a very large number of Github projects.

More attention can also be put on the outcome of the constraints. We only used DBCON3 as the database per service pattern criteria, but GWCON1 is a candidate for further analysis. It is against the pattern description that a gateway should not have a repository. The constraints can also be extended to check if the URL configuration is valid, e.g. between MsClient or Gateway and MsEndpoint.

Another topic that can be researched further is integrating rules and constraints in a software project, which supports junior developers to implement MSA patterns in their software. A company can fine-tune the implementation of MSA patterns and limit it to only a few implementations. The rules and conditions are based on these few implementations and not so wide as investigated in this thesis. JQAssistant can be enriched with a new plugin to support the validation of these MSA patterns, which contains specific rules and constraints for a company, similar to the current JQAssistant DDD plugin[33].

[32] https://www.ibm.com/cloud/architecture/architectures/microservices/reference-architecture
[33] https://github.com/jqassistant-contrib/jqassistant-java-ddd-plugin

13 References

1. Di Francesco, P., P. Lago, and I. Malavolta, *Architecting with microservices: A systematic mapping study.* Journal of Systems and Software, 2019. **150**: p. 77-97.
2. Lewis J., F.M. *Microservices Guide*. 2014 [cited 2020 Jan 3]; Available from: https://martinfowler.com/microservices.
3. Márquez, G. and H. Astudillo. *Actual use of architectural patterns in microservices-based open source projects*. in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. 2018. IEEE.
4. Taibi, D., V. Lenarduzzi, and C. Pahl, *Architectural patterns for microservices: a systematic mapping study*. 2018, SCITEPRESS.
5. Messina, A., R. Rizzo, P. Storniolo, M. Tripiciano, and A. Urso. *The database-is-the-service pattern for microservice architectures*. in *International Conference on Information Technology in Bio-and Medical Informatics*. 2016. Springer.
6. Richardson, C. *Microservice Architecture*. 2014 [cited 2020 June 1]; Available from: http://microservices.io.
7. Universiteit, O., *Workbook Software architecture*. 2nd ed. 2018.
8. Zeiner, H., M. Goller, V.J.E. Jiménez, F. Salmhofer, and W. Haas, *Secos: Web of things platform based on a microservices architecture and support of time-awareness.* e & i Elektrotechnik und Informationstechnik, 2016. **133**(3): p. 158-162.
9. Krylovskiy, A., M. Jahn, and E. Patti. *Designing a smart city internet of things platform with microservice architecture*. in *2015 3rd International Conference on Future Internet of Things and Cloud*. 2015. IEEE.
10. Arévalo, M., C. Escobar, P. Monasse, N. Monzón, and M. Colom. *The IPOL Demo system: a scalable architecture of microservices for reproducible research*. in *International Workshop on Reproducible Research in Pattern Recognition*. 2016. Springer.
11. Vianden, M., H. Lichter, and A. Steffens. *Experience on a microservice-based reference architecture for measurement systems*. in *2014 21st Asia-Pacific Software Engineering Conference*. 2014. IEEE.
12. Villamizar, M., O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil. *Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud*. in *2015 10th Computing Colombian Conference (10CCC)*. 2015. IEEE.
13. Balalaie, A., A. Heydarnoori, and P. Jamshidi. *Migrating to cloud-native architectures using microservices: an experience report*. in *European Conference on Service-Oriented and Cloud Computing*. 2015. Springer.
14. Gadea, C., M. Trifan, D. Ionescu, and B. Ionescu. *A reference architecture for real-time microservice api consumption*. in *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*. 2016.
15. Rademacher, F., S. Sachweh, and A. Zündorf. *Differences between model-driven development of service-oriented and microservice architecture*. in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017. IEEE.
16. Lysne, O., K.J. Hole, C. Otterstad, Ø. Ytrehus, R. Aarseth, and J. Tellnes, *Vendor malware: detection limits and mitigation.* Computer, 2016. **49**(8): p. 62-69.
17. Brown, K. and B. Woolf. *Implementation patterns for microservices architectures*. in *Proceedings of the 23rd Conference on Pattern Languages of Programs*. 2016.
18. Newman, S., *Building microservices: designing fine-grained systems*. 2015: O'Reilly Media, Inc.
19. Butzin, B., F. Golatowski, and D. Timmermann. *Microservices approach for the internet of things*. in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2016. IEEE.
20. Microsoft. [cited 2020 June 1]; Available from: https://docs.microsoft.com/en-gb/azure/architecture/patterns/circuit-breaker.

21. Montesi, F. and J. Weber, *Circuit breakers, discovery, and API gateways in microservices.* arXiv preprint arXiv:1609.05830, 2016.
22. Heorhiadi, V., S. Rajagopalan, H. Jamjoom, M.K. Reiter, and V. Sekar. *Gremlin: Systematic resilience testing of microservices*. in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. 2016. IEEE.
23. Namiot, D. and M. Sneps-Sneppe, *On micro-services architecture.* International Journal of Open Information Technologies, 2014. **2**(9): p. 24-27.
24. Del Esposte, A.M., F. Kon, F.M. Costa, and N. Lago. *InterSCity: A Scalable Microservice-based Open Source Platform for Smart Cities*. in *SMARTGREENS*. 2017.
25. Marru, S., M. Pierce, S. Pamidighantam, and C. Wimalasena. *Apache airavata as a laboratory: architecture and case study for component-based gateway middleware*. in *Proceedings of the 1st Workshop on The Science of Cyberinfrastructure: Research, Experience, Applications and Models*. 2015.
26. Vinoski, S., *Advanced message queuing protocol.* IEEE Internet Computing, 2006. **10**(6): p. 87-89.
27. Leymann, F., C. Fehling, S. Wagner, and J. Wettinger. *Native cloud applications: Why virtual machines, images and containers miss*. in *Proceedings of the 6th International Conference on Cloud Computing and*. SciTePress.
28. Oberhauser, R. *Microflows: automated planning and enactment of dynamic workflows comprising semantically-annotated microservices*. in *International Symposium on Business Modeling and Software Design*. 2016. Springer.
29. Stubbs, J., W. Moreira, and R. Dooley. *Distributed systems of microservices using docker and serfnode*. in *2015 7th International Workshop on Science Gateways*. 2015. IEEE.
30. Versteden, A., E. Pauwels, and A. Papantoniou, *An Ecosystem of User-facing Microservices Supported by Semantic Models.* USEWOD-PROFILES@ ESWC, 2015. **1362**: p. 12-21.
31. Potvin, P., M. Nabaee, F. Labeau, K.-K. Nguyen, and M. Cheriet, *Micro service cloud computing pattern for next generation networks*, in *Smart city 360°*. 2016, Springer. p. 263-274.
32. Fauscette, M. *ERP in the Cloud and the Modern Business*. 2013 2020-03-25]; Available from: http://resources.idgenterprise.com/original/AST-0111292_ERP_US_EN_WP_IDCERPInTheCloud.pdf
33. da Silva, V.G., M. Kirikova, and G. Alksnis, *Containers for virtualization: An overview.* Applied Computer Systems, 2018. **23**(1): p. 21-27.
34. Jamshidi, P., C. Pahl, N.C. Mendonça, J. Lewis, and S. Tilkov, *Microservices: The journey so far and challenges ahead.* IEEE Software, 2018. **35**(3): p. 24-35.
35. 42010–, I.I.I., *Systems and Software Engineering—Architecture Description*. 2011, International Standardization Organization Switzerland.
36. Bass, L., P. Clements, and R. Kazman, *Software architecture in practice*. 2003: Addison-Wesley Professional.
37. Clements, P., F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond , Second.* cit. on, 2011: p. 9.
38. Shalloway, A. and J.R. Trott, *Design patterns explained: A new perspective on object-oriented design, 2/E*. 2005: Pearson Education India.
39. Gamma, E., R. Helm, R. Johnson, J. Vlissides, and D. Patterns, *Elements of reusable object-oriented software.* Reading: Addison-Wesley, 1995.
40. Dobbelaere, P. and K.S. Esmaili. *Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper*. in *Proceedings of the 11th ACM international conference on distributed and event-based systems*. 2017.
41. Richardson, C., *Microservices patterns*. 2018: Manning Publications Company.
42. Ismail, A.A., H.S. Hamza, and A.M. Kotb. *Performance evaluation of open source iot platforms*. in *2018 IEEE Global Conference on Internet of Things (GCIoT)*. 2018. IEEE.

43. SiteWhere. [cited 2020 October 1]; Available from: https://sitewhere.io/docs/2.1.0/platform/architecture.html#microservice-approach.
44. Aderaldo, C.M., N.C. Mendonça, C. Pahl, and P. Jamshidi. *Benchmark requirements for microservices architecture research*. in *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*. 2017. IEEE.
45. Rahman, M.I., S. Panichella, and D. Taibi, *A curated dataset of microservices-based systems.* Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution, 2019.
46. Celińska, D. and E. Kopczyński. *Programming languages in github: a visualization in hyperbolic plane*. in *Eleventh International AAAI Conference on Web and Social Media*. 2017.
47. Frederickson, B. *Ranking Programming Languages by GithHub Users*. 2020; Available from: https://www.benfrederickson.com/ranking-programming-languages-by-github-users/.
48. Jung, W., E. Lee, and C. Wu, *A survey on mining software repositories.* IEICE TRANSACTIONS on Information and Systems, 2012. **95**(5): p. 1384-1406.
49. Kalliamvakou, E., G. Gousios, K. Blincoe, L. Singer, D.M. German, and D. Damian, *An in-depth study of the promises and perils of mining GitHub.* Empirical Software Engineering, 2016. **21**(5): p. 2035-2071.
50. Lenarduzzi, V., A. Sillitti, and D. Taibi. *A survey on code analysis tools for software maintenance prediction*. in *International Conference in Software Engineering for Defence Applications*. 2018. Springer.
51. Robinson, I., J. Webber, and E. Eifrem, *Graph databases: new opportunities for connected data*. 2015: O'Reilly Media, Inc.
52. Štefanko, M., O. Chaloupka, B. Rossi, M. van Sinderen, and L. Maciaszek. *The Saga pattern in a reactive microservices environment*. in *Proc. 14th Int. Conf. Softw. Technologies (ICSOFT 2019)*. 2019. SciTePress.
53. Schröder, S. and G. Buchgeher. *Discovering architectural rules in practice*. in *Proceedings of the 13th European Conference on Software Architecture-Volume 2*. 2019.
54. Ben-Kiki, O., C. Evans, and B. Ingerson, *Yaml ain't markup language (yaml™) version 1.1.* Working Draft 2008-05, 2009. **11**.

14 Glossary

API	Application Programming Interface
CD / CI	Continuous Delivery / Continuous Integration
CQRS	Command Query Responsibility Segregation
CRUD	Create Read Update Delete
CS / SS	Client Side / Server Side
DDD	Domain-Driven Design
EE	Enterprise Edition (Java or Jakarta)
HTTP	HyperText Transfer Protocol
IOT	Internet Of Things
JSON	JavaScript Object Notation
MQ	Message Queue
MQTT	Message Queue Telemetry Transport
MSA	Microservices Architecture
MVC	Model View Controller
OSS	Open-Source Software
REST	Representational State Transfer
RPC	Remote Procedure Call
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
XML	eXtensible Markup Language
YAML	YAML Ain't Markup Language (Yet Another Markup Language)

15 Appendix A – Literature study for MSA patterns

The literature study for **Architectural patterns and microservices** was done via google scholar with the search terms [3]: (architectural) AND ((micro service) OR (microservi) OR (micro-servi)) AND (pattern) AND (design). See table below, P. means no. of patterns mentioned in the article, C. means no. of citation of the article, R. means that it is used as a reference in this paper, see Chapter 13.

Year of publication / Title	Authors	P.	C.	R.
2019				
Architecting with microservices: A systematic **mapping study**	Di Francesco et al.	8	11	[1]
A curated Dataset of Microservices-based systems	Pahl et al.	0	3	
Qualitative Evaluation of Dependency Graph Representativeness	Nurmela et al.	1	-	
From Monolithic Systems to Microservices: An Assessment Framework	Taibi et al.	0	1	
2018				
Actual Use of Architectural Patterns in Microservices-based Open Source Projects	Márquez et al.	17	4	[3]
An-**Exploratory-Study**-of-Academic-Architectural-Tactics-and-Patterns-in-Microservices-A-systematic-literature-review	Márquez et al.	44	-	
A pattern language for scalable microservices-based systems (refers to [6])	Márquez et al.	38	3	
An Empirical Study of Scalability Frameworks in Open Source Microservices-based Systems (refers to [6])	Márquez et al.	9	-	
Review of Architectural Patterns and Tactics for Microservices in Academic and Industrial Literatur (Spanish)	Márquez et al.	-	6	
Architectural Patterns for Microservices: A Systematic **Mapping Study**	Taibi et al. (Pahl)	7	52	[4]
Towards Micro Service Architecture Recovery: An Empirical Study (refers to [6])	Alshuqayran et al.	0	6	
Microservices: The journey so far and challenges ahead	Jamshide et al. (Pahl)	2	65	
On the definition of microservice bad smells (refers to [6])	Taibi et al.	1	44	
Security Strategies for Microservices-based Application Systems (has a link about inter-process communication)	R. Chandramouli	7	1	
Microservices Migration Patterns	Balalaie et al.	15	54	
microservices-pains-gains (refers to [6])	Soldani et al.	5	38	
Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture	C. Rudrabhatla	2	1	
SPARQL Micro-Services: Lightweight Integration of Web APIs and Linked Data	Michel et al.	2	8	
Continuous-Architecting-With-Microservices-and-DevOps-a-Systematic-Mapping-Study	Taibi et al. (Pahl)	7	1	
Design of Microservice Architecture for IOT	Pushpalatha et al.	0	-	
A Mapping Study on Microservice Architectures of Internet of Things and Cloud Computing Solutions (overview no. of studies)	G. Campeanu	0	5	
2017				
A Systematic Literature Review on Microservices	Vural et al.	0	34	
Understanding cloud-native applications after 10 years of cloud computing – A systematic mapping study	Kratzke et al.	4	86	
Ensuring and Assessing Architecture Conformance to Microservice Decomposition Patterns (refers to [6])	Zdun et al.	0	8	
Microservices in Practice, Part 1 and 2	Pautasso et al.	6	98	
Towards a Taxonomy of Microservices Architectures	Di Elettronica er al.	3	17	
Overview of a Domain-Driven Design Approach to Build Microservice-Based Applications	Steinegger et al.	4	5	
Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption	Di Francesco et al.	5	115	
Towards Recovering the Software Architecture of Microservice-based Systems	Di Francesco et al.	0	38	
MicroART: A Software Architecture Recovery Tool for Maintaining Microservice-based Systems	Di Francesco et al.	0	22	
2016				
The database-is-the-service pattern for microservice architectures	Messina et al.	5	19	[5]
A Simplified Database Pattern for the Microservice Architecture	Messina et al.	5	13	
Microservices: A Systematic Mapping Study (overview of 2015!) -	Pahl et al. (Jamshidi)	7	143	
Microservices approach for the internet of things	Butzin et al.	7	86	
microservices-in-practice-key-architectural-concepts-of-an-msa	K. Indrasi	6	1	
A Systematic Mapping Study in Microservice Architecture	Alshuqayran et al.	2	160	

16 Appendix B – Literature study of selected MSA patterns

The list of references was retrieved from five selected sources. The duplicate documents mentioned in more than one source per pattern were removed. Some documents were not available and removed from the list because they were not accessible for free (mainly via IEEE).

Architectural pattern	Category	Papers and references mentioned in the five selected sources
API gateway (Aggregator, Proxy, Backend for frontend)	Communication layer	[1] P5, P34, P36, P38, P45, P48, P63, P74, P77, P84, P90 [3] R2, R17, R35, AP62, IPA2, IPA57, IPA67 [4] S2, S3, S11, S14, S31, S34 [5] [6]
Messaging (Publish/ subscribe)		[1] P2, P3, P48, P100, P103 [3] R39, IPA2, IPA60 [6]
Circuit breaker		[1] P15, P29, P42, P44, P63, P89 [3] R35, AP12, AP69, IPA2, IPA63 [6]
Discovery (Registry)		[1] P1, P48, P63, P85, P91 [3] R17, R35, R36, AP62, AP69, IPA2 [4] S1, S2, S7, S9, S10, S13, S16, S24, S25, S36 [5] [6]
Database per server (Shared, Cluster)	Data layer	[1] P48, P98, P101, P103 [3] R18, AP31, AP62, IPA2, IPA60 [4] S11, S13, S15, S16, S18, S23, S25, S27, S36 [5] [6]
Caching		[3] R17, AP24
CQRS		[6]

17 Appendix C – JQAssistant programs for analyzing MSA patterns

Program	Remarks
msa-projects-load-scripts.adoc	Scripts to build and load the projects-jars into Neo4J.
msa-projects-result-scripts.adoc	Scripts to query Neo4J database to retrieve results.
msa-patterns-microservice.adoc	Node and relations for microservices.
msa-patterns-repository.adoc	Nodes and relations for repositories.
msa-patterns-configuration.adoc	Nodes and relations for configuration.
msa-patterns-gateway.adoc	Nodes and relations for gateway pattern.
msa-patterns-messaging.adoc	Nodes and relations for messaging pattern.
msa-patterns-circuitbreaker.adoc	Nodes and relations for circuit breaker pattern.
msa-patterns-servicediscovery.adoc	Nodes and relations for service discovery pattern.
msa-patterns-caching.adoc	Nodes and relations for caching pattern.
msa-patterns-database-per-service.adoc	Nodes and relations for database per service pattern.
msa-patterns-database.adoc	Nodes and relations for databases.
msa-patterns-cqrs.adoc	Nodes and relations for CQRS pattern.
msa-patterns-constraints.adoc	All constraints
style.grass	Colour and size styles for nodes in the Neo4J browser
scan_keywords.txt	Grep commands for text-based search for patterns
<see directory pictures>	Visual graphs of some projects
<see directory output>	csv-output files of result-scripts

18 Appendix D – Visualization of project PiggyMetrics

The following figure shows the Neo4J visualization of project PiggyMetrics (see Table 38) after loading the project's jars and scanning them with JQAssistant.

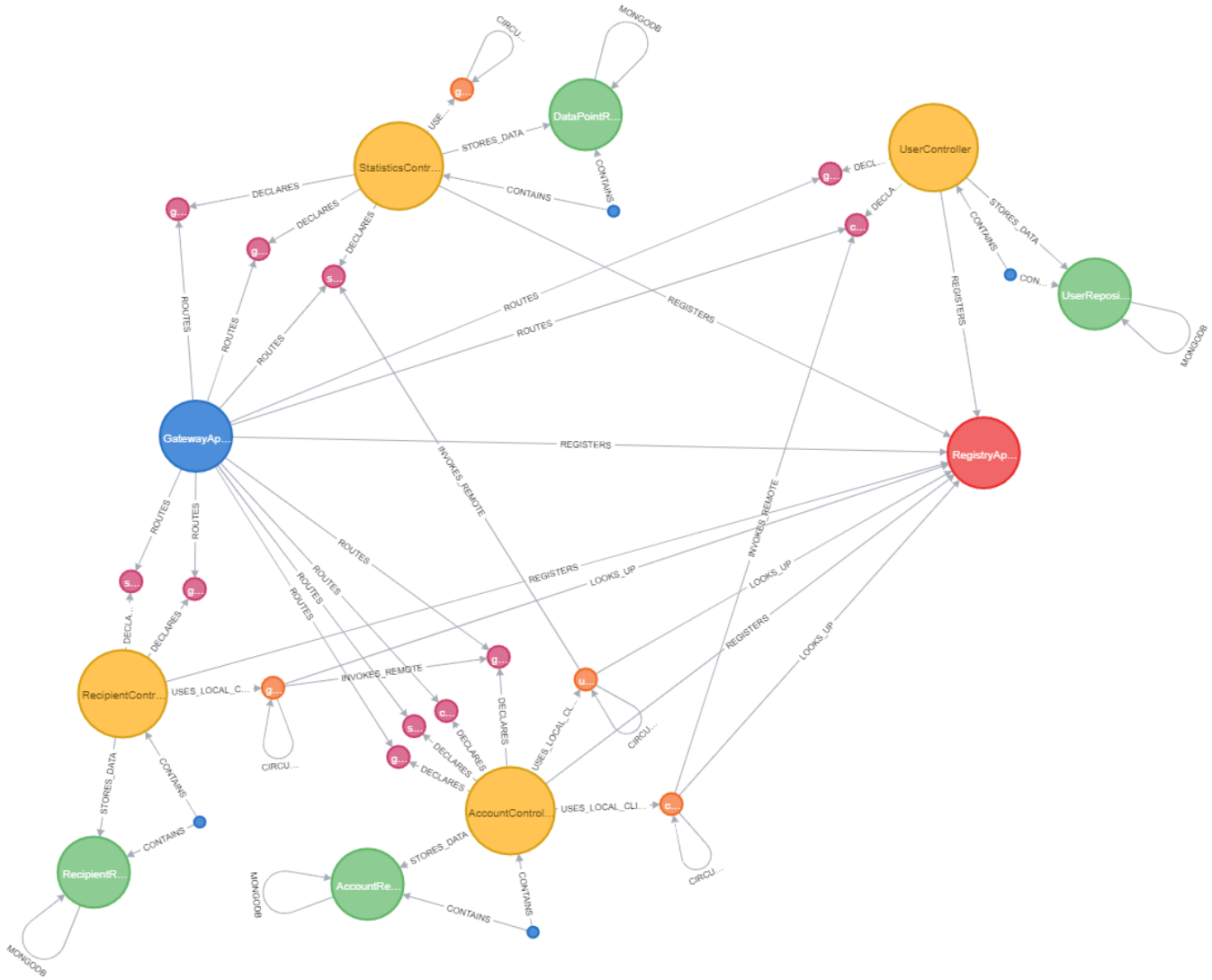


Figure 27 - Visualization of project PiggyMetrics with JQAssistant and Neo4J