Linköping University | Department of Computer and Information Science
Bachelor thesis | Programming
Spring term 2017 | LIU-IDA/LITH-EX-G--17/012--SE

An analysis of microservice frameworks

Erik Edling
Emil Östergren

Tutor, Peter Dalenius
Examinator, Rita Kovordanyi

LINKÖPINGS UNIVERSITET

Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Erik Edling and Emil Östergren

An analysis of microservice frameworks

Erik Edling
Linköping University
Linköping, Sweden
Eried975@student.liu.se

Emil Östergren
Linköping University
Linköping, Sweden
Emios530@student.liu.se

ABSTRACT
Microservice architecture has entered the industry to solve some of the problems with the monolithic architecture. However, this architecture comes with its own set of problems. In order to solve the microservice architecture problems while also providing additional functionalities, microservice frameworks have been developed. In this thesis, microservice frameworks were compared and thereafter two were chosen to implement a small part of a large monolithic system as microservices. This was done in order to see how well they could implement the different functionalities that the frameworks provided in relation to the benefits and the cross-cutting concerns of the microservice architecture which are concerns that is applicable to the entire system. The results showed that the frameworks embraced the benefits of the microservice architecture in the aspects of maintainability and scalability. However, in the terms of being able to change frameworks in the pursuit of newer technologies there were problems. Some functionalities such as service discovery requires all of the new services created to use the same mechanism in order to create a unified system. There were also problems caused by the load balancing mechanism provided by the frameworks used in this thesis. The load balancing mechanism made the system unable to send large data files which was crucial for the system that was to be implemented as a microservice system.

INTRODUCTION
Microservices has emerged in the recent years to tackle the problems with monolithic systems. In a monolithic system, all components are closely bound together into a single unit and as the requirements of the systems has increased over the years, so has the size required by them. A huge problem with ever growing monolithic systems is that it makes continuous development and continuous delivery very difficult as the entire application has to be tested and redeployed even with small changes to the codebase[1][2]. Monolithic systems also have the downside of not being able to scale properly and decreased modularity[1]. Decreased modularity creates problems such as less flexibility to switch out or reuse components in a system[3]. Other problems with the monolithic architecture is that the growing systems can cause problems such as introducing new employees to the large codebase, reduction in productivity, increasing the need of coordination for the development and the ability to change frameworks in pursuit of newer technologies[1].

Microservices is one of the solutions to the heavier monolithic systems. Instead of deploying the system as a single unit, the system is divided into multiple small services that can communicate through network calls. Each service handles typically one specific task and is independent from the rest of the system[4]. The services can be deployed either on multiple or a single machine while the latter should be avoided to prevent a single point of failure according to Sam Newman[5].

Microservices are however not without their own share of problems. Dividing the system into smaller services introduces new security risks[6]. Since services requires communication through network calls, they are more exposed to threats. Dividing the system also decentralizes functionalities such as logging and distributed tracing. When the system is divided across multiple servers, tracking errors can be very hard, especially if there are two or more servers for each service. Another big problem with microservices is the service discovery problem. Since microservices are scalable, locating services can be complex and needs to be dealt with in real time[7]. Implementing functionalities to solve these problems that is applicable to the entire system, also referred to as cross-cutting concerns, can take up a lot of development time. Microservice systems can contain hundreds of services and each service would have to implement these cross-cutting concern functionalities which is not feasible.[8]

Microservice frameworks can be of a big help countering these cross-cutting concerns by providing the system with certain functionalities. These can be categorized into two groups, the necessary and the beneficial functionalities. The necessary functionalities are those that are needed in the system in order for it to work correctly. The beneficial functionalities can be seen as lower prioritized functionalities that might be of use in future development or higher prioritized functionalities that the companies particularly want. These can be either from the cross-cutting concern functionalities or the functionalities that are not directly linked to the microservice architecture. The system that is to be implemented as a microservice architecture needs to be analyzed in order to create a working solution as well as to get the most out of using microservice frameworks.

PURPOSE
The goal of this thesis is to examine and compare different microservice frameworks in order to find the most promising ones and then use them to implement a small part

of an existing monolithic system written in Java. After the system is implemented as microservices, the workflow and the solution will be analyzed in order to find out how well the microservice frameworks can implement the different functionalities that are beneficial or necessary from a microservice architecture perspective.

RESEARCH QUESTIONS

To help choose frameworks to use for an implementation the following question will be answered:

What functionalities do the microservice frameworks provide that can be beneficial or necessary for an implementation of a microservice system?

When the implementations have been made we will analyze the following question:

How well can the chosen frameworks implement the beneficial or necessary functionalities in relation to the benefits and the cross-cutting concerns of the microservice architecture?

DELIMITATIONS

The implementations that were created for this thesis were based on a smaller independent part of a monolithic system. This part was written in Groovy while the rest of the monolithic system was mostly written in Java. Because of this, the implementations had to support at least Java as it supports Groovy by default. It also means that some functionalities were required for the implementations to work correctly such as handling of large data chunks. An alternative way of providing us with needed or wanted functionalities were to use pure third-party libraries, however as this thesis was focusing on the capabilities of the microservice frameworks the libraries was not taken into consideration when choosing frameworks.

THEORY

In this chapter, we will first go through the background information regarding the system that is to be implemented as microservices. Afterwards we will explain the theory for microservice architecture and its origin, benefits, problems and how microservice frameworks comes into the picture. This is to give a clear explanation to the different technologies this thesis revolves around.

Background

The task for this thesis was provided by Ida Infront and was mainly to investigate different microservice frameworks and which one that suited their system the most. They also provided some requirements for the system together with some more prioritized functionalities.

When it comes to microservices there are many problems and there is next to nothing scientific written about microservice frameworks. This thesis was created in order to find out if the frameworks could support the developer handling those cross-cutting concerns while additionally helping the developer implementing extra functionalities that could also be beneficial or necessary.

Short background about the targeted system

The targeted system that was to be migrated ta a microservice architecture is used to handle data that will be archived and was written in Groovy. Before the data can be stored it has to go through different processes such as file converting, generation of thumbnails and file validation. The system was already divided with its own RESTful interface but is still hosted on a single server.

Microservices

To explain microservices we must first go back to its origin which is service-oriented architecture. Service-oriented architecture has been around for a long time. The idea behind it is to separate different systems into their own smaller services[10]. However, a problem was that there were no standards on how to create a service-oriented system. This meant that it was up to the developer to interpret the architecture, for example which part that should be put into which service and the determination of how big each service should be. Microservice architecture emerged to handle the problems with service-oriented architecture and thereby provide developers with more standards on how microservices should be created. In other words, microservice architecture is a specialization of the service-oriented architecture with more well defined standards.[5]

What is a microservice?

According to Thönes “A microservice, in my mind, is a small application that can be deployed independently, scaled independently, and tested independently and that has a single responsibility”[4]. With this explanation, we can see that the core concept of microservices is that each microservice should only have one responsibility. In a larger system, there will be many of these services communicating with each other as shown in figure 1. This entails strong cohesion and loose coupling[9].

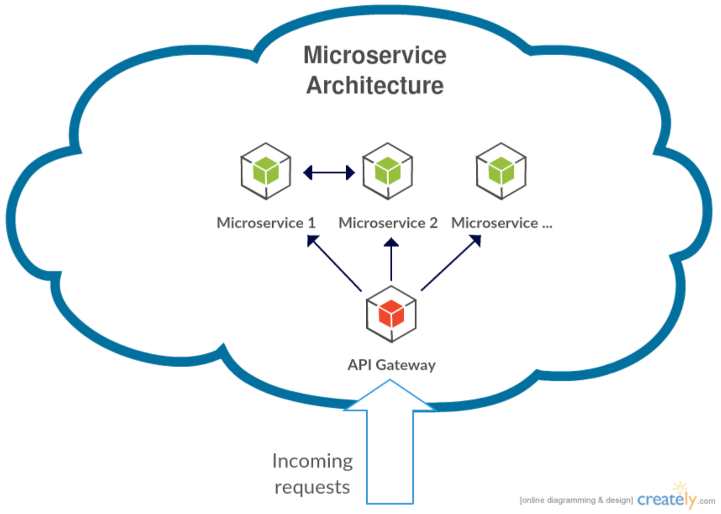


Figure 1: An example of a microservice architecture.

What are the key benefits of microservices?

When it comes to microservice architecture there are several benefits to be found when comparing it to the old monolithic architecture. These benefits are the following:

Scaling - One of the big flaws with a monolithic system is that it does not scale very well since even the smallest changes requires a new deployment of the entire system[1]. With microservices, only the actual service that needs to be scaled requires a new instance to be deployed[9].

Easier to maintain - When a monolithic system gets bigger, it becomes more complex thus harder to maintain compared to a system with microservices where each service is kept small and precise[9].

Flexibility - As each service in a microservice architecture system is run and works independently, each one can be changed without it affecting the others, allowing gradual updating or upgrading to take place[7].

Problems with microservices

Just like other architectures, microservices does not only have benefits but comes with its own set of problems. These problems include:

Service Discovery - In a system built with a microservice architecture, the deployed microservices might be running several instances of a single service. These services might encounter failures or upgrades and since the microservices are dynamically changing, their address locations as well as the number of instances might change. This makes it hard to keep track and provide the correct service that currently is in good health to the end user.[7]

Security - As communication between various parts of the system now is handled by sending network calls instead of function calls within the system, it introduces a new security risk. The communication can now be more easily intercepted and either read or even manipulated if security measures are not implemented.[6]

Decentralization of functionality - When the system is divided into multiple services, functionalities such as logging now becomes more complex to handle[7]. In a microservice system with hundreds of services you might not want to retrieve logs from every single service manually.

Load balancing - Load balancing is a problem for all distributed systems. As the traffic increases the system needs to evenly distribute the traffic among the healthy instances of a service in order to keep a high performance on the system.[11]

Fault tolerance - Circuit breaking is a technique for fault tolerance in a microservice system. It is built upon a similar principle as the physical one. As a service becomes unavailable or nonfunctional, the circuit breaker makes sure that the call is not repeated to the same service. Instead the call is made to another instance.[12] Fault tolerance can be very complex and needs to be dealt with.

Countering the problems and cross-cutting concerns using Microservice Frameworks

The basic idea of a framework is to provide tools which helps developers create applications in certain domains. When it comes to microservices, it is possible to use frameworks that provides certain functionalities that can be useful when creating a system with a microservice architecture. We will call these frameworks microservice frameworks. With a good microservice framework the developer can focus on the continued development of the microservice system and let the framework handle the general cross-cutting concerns of microservices.

Related work

When it comes to monolithic, service-oriented and to some extent microservice architecture a lot of research has been done (see chapter: Theory). However, the subject of microservice frameworks is relatively untouched by the scientific community and mostly written about in places such as blogs and the like. Scientific work directly related to the subject at hand could not be found.

The research that has been conducted about microservices has mostly been about defining the overall concept, problems of microservices and how to deploy them.[1][9][7]

METHOD

To reach a conclusion for the research questions asked in this thesis, three different steps were performed. First, a comparison was made in order to choose suitable frameworks. Then an implementation of a small part of a huge monolithic system was done using two different frameworks. Lastly, an evaluation and analysis was made in order to find out how well these frameworks could handle the implementation of the earlier stated functionalities that are beneficial or necessary.

Choosing frameworks

Currently on the market there are many frameworks at different development stages and every framework differs to fill a specific need in the market. Depending on what type of application that is being developed, there can be diverse types of functionalities that can be classified either as beneficial or even necessary.

First, the different frameworks were identified and explored. This was done by looking through three different kind of web pages.

- Various community pages involving microservices, distributed systems and service-oriented architecture frameworks.

- Google searches with combinations of the words microservice, service-oriented architecture, distributed system frameworks.
- Various tutorial websites and blogs for creating microservices.

We examined what types of functionalities these frameworks provided and created a list of all these combined in a union.

Before the system can be implemented, the developer should know about the different requirements of the system that is to be implemented as microservices. In the case for this thesis, all the requirements that the system needed in order to work correctly were given by the contractors. If this would not be the case, the system should be analyzed in order to find them.

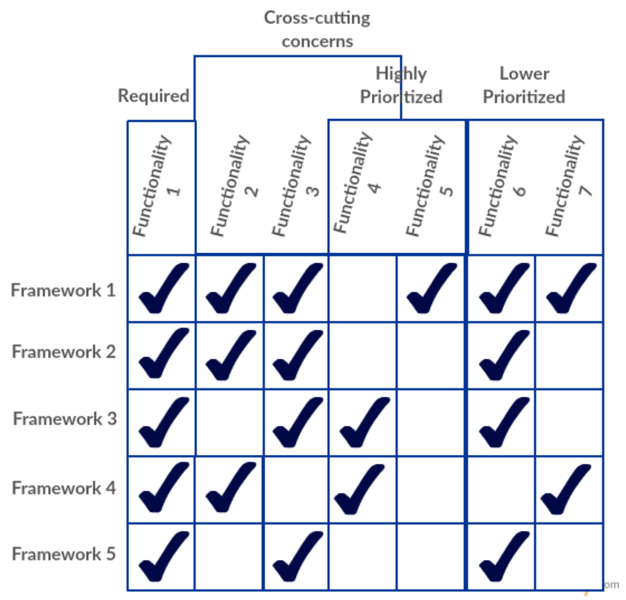
When all the functionalities had been identified, they were categorized into four groups in order to create a matrix. These groups were:

- Requirements for the microservices that the system needed when transferred from monolithic architecture to microservice architecture
- Functionalities that could help countering the cross-cutting concerns of microservices.
- Highly prioritized functionalities that were wanted, in this case by Ida Infront, for the system that was to be implemented as microservices.
- Lower prioritized functionalities.

With these categorized groups, a matrix was constructed with functionalities as one axis and the frameworks as the other. When this was done the matrix was to be filled with data on whether the framework had the listed functionality or not. This is illustrated in figure 2.

With this data, the different frameworks could be compared against each other in order to finally select two of them to implement the system in.

The comparison of the frameworks involved a discussion based on the four categorized groups. The first category was the most important one seeing as these functionalities were necessary for the system to work. The next priority is the highly prioritized functionalities followed by the functionalities that handles cross-cutting concerns. The last category was functionalities that are or could in the future come to be useful for the system, so they were lower prioritized. If there were multiple frameworks that nearly had the same number of functionalities, a subjective analysis of the community and official documentation had to be made to see which framework that seemed to be most mature and easiest to develop with. This was in some part due to the threat that is explained later in the Discussion – Method chapter.



	Required	Cross-cutting concerns		Highly Prioritized		Lower Prioritized	
	Functionality 1	Functionality 2	Functionality 3	Functionality 4	Functionality 5	Functionality 6	Functionality 7
Framework 1	✓	✓	✓		✓	✓	✓
Framework 2	✓	✓	✓			✓	
Framework 3	✓		✓	✓		✓	
Framework 4	✓	✓		✓			✓
Framework 5	✓		✓			✓	

Figure 2: Matrix showing what functionalities different framework has and what category they are categorized in.

With the frameworks selected, the implementation of the system could begin.

Implementation

The implementation process of the microservices consisted of two steps. The first step was the process of making the system compatible with microservices. The second step was to implement the different parts as microservices together with all the wanted functionalities.

Making the system microservice compatible

Constructing microservices out of a monolithic system is no simple task. There is a lot to think about and to take into consideration depending on how the monolithic system is built. We found an interesting approach to help us with the migration. An article series was posted by Kyle Brown at IBM and it shows the first steps involved in the migration process[13]. These steps only described how to repackage the monolithic system into a structure that could easily be translated into microservices. It did not describe how to implement the system as microservices, this step was different depending on what microservice framework that was used for the implementation.

The article series provided by IBM very deeply explains the process of migrating the system. In this thesis however, each step will only be given a shallower explanation. For a more in depth read, each step can be found in the IBM article series[13].

Step 1: Repackaging the application

Before splitting the application, it was most likely to be packaged as an EAR file which is a package containing multiple modules which is illustrated in figure 3. The ultimate goal with this step was to split the application into several WAR or JAR files, which basically are files that contains all necessary resources for a web application. These were to be deployed, each file as its own service. [13]

Step 2: Refactoring code

In step 1, the application was split up into multiple WAR files. In this step, each WAR was analyzed and if possible, split up into even more parts. There could be several patterns to explore in order to find parts that could be untangled and put into their own service. For example, if some parts of the system were already using REST API it was as simple as dividing each of these into their own services. However, if there were parts that needed to be divided but were communicating through function calls, a new interface would be created for those parts[13]

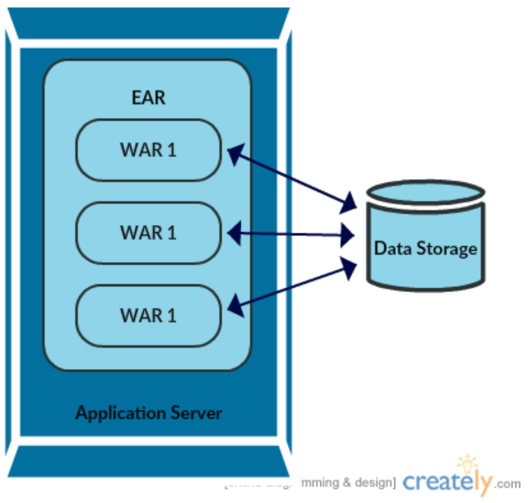


Figure 3: An illustration of how an entire system is deployed as an EAR file on a single application server.

Step 3: Refactoring the data

When the application was split up into small services, the data was to be refactored if applicable. This entailed that for each service, the goal was to have as little data transfers with other services as possible. To reach this goal the data structure from the monolithic system would be split up onto each service when applicable.[13]

Implementing the system as microservices

Once the system was properly divided into smaller parts, it was time to implement them as microservices using the chosen frameworks. This process would be different depending on what frameworks that were chosen in the previous step. This was because of the following two reasons:

Framework differences - The first reason was the most obvious one. The frameworks themselves had different approaches on how to handle the implementation.

Different functionalities - As previously stated, each microservice framework tried to differentiate itself by focusing on various aspects of microservices. Some of the microservice frameworks might have had support for some functionalities while others did not.

When the needs for all the required functionalities were covered and the microservices were up and running correctly, the different functionalities that were supposed to handle the cross-cutting concerns of the microservice architecture and the prioritized functionalities were implemented one by one.

Analyzing the framework based on the implementation

Based on the implementation phase and the results, several questions were examined in order to help answer the second research question. These questions were the following:

1. How did the implementation process or solution for each functionality implemented stand out to its expectations?
2. Was there any cross-cutting concern or prioritized functionality that the framework could not handle well?
    a. Was it because the framework did not include all the functionalities that was needed?
    b. Was it because it did not handle the advertised functionality well enough for the targeted systems standards and if so, why not?
3. For the cross-cutting concerns or prioritized functionality that the framework could handle, did the framework help reduce the complexity in the development process? Was it easy to use?

RESULT

This chapter will show the results of the thesis and has the same structure as the Method chapter. However, each framework implemented will have its own implementation chapter.

Choosing framework

By searching through the internet using the different tactics explained in the method chapter, 15 different frameworks were found which can be seen in appendix A. From these frameworks and their functionalities, a matrix was created. The functionalities were limited to those that are of particular interest in a microservice architecture and was gathered by searching through the official documentations and user guides. The functionalities that were found were grouped into the categories mentioned in the method chapter. The matrix can be seen in appendix B.

The matrix showed that there was a large variation when it comes to the number of functionalities each framework had. While there were some that had a high number of functionalities, there were many that only had few with the goal of letting developers quickly and easily get started with deploying multiple microservices. However, there were three frameworks that stood out to the rest of them, Spring cloud, Wildfly Swarm and VertX. While Spring cloud satisfied our needs the most with only one missing lower prioritized functionality, Wildfly Swarm and VertX were not far behind. Therefore, a subjective evaluation of the community and documentation was made.

Spring Cloud was the framework with most in depth guides and had overall more detailed documentation. Spring covered all the functionalities except one lower prioritized while Wildfly missed one highly prioritized and VertX missed one from the cross-cutting concerns. There was also a lot more community threads about Spring Cloud. With this in mind, the conclusion was to use Spring Cloud for the implementation.

For the second framework to implement the system in, VertX was chosen. It had more in-depth documentation while Wildfly Swarm had significantly less so. VertX also supported all the higher prioritized functionalities while Wildfly Swarm was missing one highly prioritized functionality.

Implementation

The implementation chapter is divided into the two frameworks. A walkthrough of the implementation for each framework is described to give a clear picture of what functionalities and techniques that were used for the specific framework.

Making the system microservice compatible

As described in the Background chapter, the system that was to be implemented as microservices was already relatively divided. However, each of the three steps previously explained was thoroughly performed to make sure no important part was missed. The following steps were performed in the same manner for both chosen frameworks.

Step 1: Repackaging the application

The system that was to be implemented as microservices was originally built into a single JAR file. Each of the RESTful interfaces were split up into their own independent service which resulted in several JAR files when built.

Step 2: Refactoring code

Since the system already was created as a RESTful application it was an easy process migrating the current interface to the different frameworks REST interfaces. The current interface was created as methods instead of the previous RESTful endpoints that the newly created interface would call.

Step 3: Refactoring the data

A thorough investigation of the system showed that it did not use any database for saving data thus making the third step redundant in this thesis. However, in other systems with databases, this step is very important in order to make each microservice independent.

Implementing the system as microservices

When the system was divided into their own independent services, the implementation of the microservice functionalities began. The process of implementing each functionality implemented will be explained briefly.

Spring Cloud

Spring Cloud is a framework that is built upon its predecessor Spring Boot which is a framework created with the principle of making it easy to create an application. Spring Cloud integrates Spring Boot with many external libraries/frameworks such as Netflix components, Consul and Zipkin. Each of these libraries/frameworks provides different functionalities such as load balancing and service discovery and are implemented by writing minor configurations in the Spring Boot application.[14]

Security - All REST endpoints are by default protected by Basic Authentication. Every single request in Basic Authentication requires a username and password. For a more advanced authentication OAuth2 could be used instead which provides the system with a single sign in token based authentication. For this thesis, Basic Authentication satisfies the basic needs for secure endpoints while also leaving more time to implement the rest of the functionalities.

Service Discovery - Service discovery was created using Consul. Consul runs as its own standalone server that dynamically keeps a record of all services and their ip addresses. Each microservice that is to be registered in Consul simply adds some basic configurations in the Spring Boot application. When each microservice is started, they will register themselves with their IP, port and service name to Consul.

Distributed Logging - Spring Cloud does not officially support any type of distributed logging functionality. However, Spring cloud integrates Sleuth which gives the system distributed tracing using Zipkin which can include data logs. A Zipkin service is created as its own service with some minor configurations. The Zipkin service acts as remote collector of the logs and provides a user interface for viewing them. Each service sending logs also requires some minor configurations and will after this send logs through an intermediate server called RabbitMQ that simply needs to be installed on the server. RabbitMQ is a message broker that handles the messages sent between each service and the remote collector service Zipkin.

Metrics and Diagnostics - Metrics and diagnostics are by default provided in a Spring Boot application. An administration service was created so that metrics and information of the system could be observed. This service could be created without any external integrated library. It was possible for each microservice to create their own set of custom metrics. Spring Boot applications also supports health check endpoints which can be used together with Consul in order to see which services that are healthy.

Remote Configuration - Remote configuration was implemented using Spring cloud config. The configuration architecture consists of a configuration service that handles the connection between normal services that wants to retrieve configurations and the configuration repository where all configurations are stored. When a service starts up and has remote configuration enabled the service initially uses its local configuration. It then makes a request to the remote configuration microservice. If there are any remote configuration for the specific microservice it will be used instead of the local configurations.

Load Balancing and Circuit Breaker - Load balancing and circuit breaker was handled by Spring Cloud Ribbon. Ribbon utilizes the service discovery, in this case Consul to find a healthy instance of a microservice. Ribbon has a built-in functionality for circuit breaking which filters out the services that fails to respond.

Large Data Handling - Receiving large data, typically 200GB files worked smoothly. This could be done by doing some minor configurations of maximum file size received in Spring. However, when the implementation used load balancing with Ribbon, a solution to send large data could not be found. 1 GB files worked smoothly while 50GB files did not.

VertX

VertX is a lightweight microservice toolkit that provides a wide variety of functionalities, both built in functionalities as well as third party functionalities such as service discovery with Consul and metrics with Dropwizard. VertX also support multiple languages making the developer able to choose the best language for the given situation. In order to make a better comparison between the chosen frameworks, the language chosen for this implementation was in Java.

Security - VertX supports multiple authentication and authorization functionalities such as JWT auth(Json Web Tokens) and OAuth2. For this implementation, JWT auth was chosen as it gave an easy way of handling authentication and authorization. An auth server was created to handle the authentication and authorization of the services so that all calls between microservices were authorized.

Service Discovery - Vertx allowed for many different ways of implementing service discovery. It supports for example consul for service discovery however, Vertx also supports Hazelcast which is a cluster manager. Each service registers itself into a cluster and each call for another service inside the cluster is handled by a deeply integrated event bus. When a service publishes a request on the event bus, the affected service that should handle the request consumes it and returns the result asynchronously. This means that a true service discovery was not needed as this was automatically handled by the cluster manager together with the event bus.

Metrics and Diagnostics - Metrics were handled with Dropwizard metrics. In order to display metrics, Hawtio was used which is a modular web console for managing applications. In order to expose the metrics a Jolokia agent was needed for each microservice that wants to expose the data. Jolokia acts as a middleware to make sure that Hawtio can access the metrics. Consul was used to implement health checks as no documented solution for exposing the health status to Hawtio could be found.

Remote Configuration - VertX supports a native way of collecting remote configurations from different kinds of configuration stores such as Git and Kubernetes Config map store. For this implementation, git was chosen as it required only a few lines of code in order to work. As the services starts up, the configurations were collected asynchronously and when the services had collected all configurations, the service was started.

Load Balancing and Circuit Breaker - Load balancing was not found in VertX until the implementation phase of VertX had begun. This was due to it not being mentioned as load balancing in the documentation. Load balancing in VertX was handled automatically when using the event bus for sending messages between microservices. However, normal REST calls were not load balanced and no supported way of doing this was found. Circuit breaking however only needed a few lines of configuration together with a few lines of code for each endpoint that was to be protected with the circuit breaker.

Large Data Handling - When it came to receiving large files there was no problem. However, when it came to sending large data there was more to it. Since VertX is asynchronous by nature, it limited the implementation to use their built-in functionalities to send data. No solution was found to make them handle large data files.

Analyzing the framework based on the implementation

By analyzing the frameworks using the questions provided in the chapter "Method – Analyzing the framework based on the implementation" the following results were found.

Spring Cloud

The implementation with Spring Cloud took approximately two and a half weeks. Most of the functionalities were easy to set up, some services only required a few lines of configuration to get them up and running. Service discovery, monitoring, load balancing and circuit breaking also needed a separate central service which was explained in the implementation chapter. These were also easily set up with some minor configurations. Spring Cloud services consisted of about 150 lines of code excluding the code representing the monolithic system. Even though each functionality was easily set up it was still necessary to read and understand the documentations which explained each functionality thoroughly. Without the thorough documentation, even the simplest configuration would have been immensely harder.

For centralized logging and remote configuration more work was needed. Even with a good documentation it was hard to know exactly how to configure them to work correctly with a system that differed any from the official guides. A lot of experimentation with configurations was needed in order to finally get them working correctly. Compared to the other functionalities, these took approximately five to six times longer to implement.

Centralized logging and handling of large data chunks did not work as well as expected. For centralized logging, the built-in solution only supported distributed tracing which does not give all of the logging capabilities wanted in the microservice system. It was not possible to save logs at different logging levels apart from errors and non-errors.

The framework supports handling of large data chunks however, the method used when making a request with load balancing did not support large enough files. It was possible to skip the built-in load balancing and instead make a custom implementation but that would increase the complexity immensely, be too time consuming for this thesis and would no longer be considered to be a built-in feature which this thesis focuses on.

VertX

The implementation with VertX took approximately three weeks. In VertX, all functionalities except for remote configuration and metrics were a bit tricky to implement. Even though VertX has a huge documentation and numerous examples, it did not fully explain every single part that was needed to implement the functionalities.

Coming blind into the framework, it was very hard understanding the correlation between the event bus and the cluster. It was hard finding examples or community threads about creating a system with both the cluster and the event bus combined. When the cluster and the event bus were implemented however, other pieces such as load balancing and service discovery was easy to get in place.

In VertX, most functionalities required more than just a few configuration lines. For example, to get health checks working the mechanism needed to be written by hand although with strong tools provided by VertX. To get a clearer picture of this, each service consisted of about 350 lines of code excluding the code representing the monolithic system.

Implementing security by authentication and authorization was a simple task in Vertx. However, when it came to the cluster and event bus, which are deeply integrated functionalities of VertX, we could find no proper information in the VertX documentation or tutorials. To find this information it was necessary to dive into Hazelcast which documentation rivals the size of VertX's. To get this working, more time than available would have been needed to be invested.

DISCUSSION

Result - Choosing framework

The matrix summarizing the features offered in different frameworks showed that some of the different frameworks were very similar. This made it hard choosing a framework without either doing a subjective analysis of various aspects such as maturity, community, documentation and guides or creating some form of prototype with each framework to see how it stands out. The latter were in this case not an option as it would have taken up too much time.

While the implementation phase was in progress, extra functionalities were found in VertX such as load balancing which was not marketed in the documentation. This clearly shows that it can be hard finding out every aspect of a framework without getting into each framework on a deeper level.

Result - Implementation and analysis

Microservice architecture has a lot of benefits compared to its predecessor but how well could the system created in this thesis uphold to them? The implementation phase showed a lot of promising results. If there would be more traffic on any part of the system, scaling the system was as simple as deploying another instance of that particular service. With Spring Cloud this service would automatically register itself into the service registry and can thereby be contacted by the rest of the services via the load balancing mechanism. In VertX this was done by using Hazelcast clustering together with the event bus. Another big benefit is the maintainability of the system. In a monolithic architecture, it could take a very long time to build the system making it hard to quickly release new functionalities[1][2]. In this implementation, we only had to rebuild the specific service that was updated making the build process easy and fast. This made it very suitable for continuous development and delivery. This was true for both frameworks.

As previously stated microservice frameworks is supposed to handle the cross-cuttings concerns of the microservice architecture. The implementation phase of the thesis showed a lot of promising results. The entire system with nearly all cross-cutting concern functionalities was implemented in a relatively short amount of time. However, it also gave more insight into some problems while implementing the system. Even though the framework handled the cross-cutting concerns, it could sometimes be hard to find a correct, stable solution for a specific system solution. In both Spring Cloud and VertX documentation, some aspects were missing on how to configure and implement our system. This could have been due to the many different possible combinations of functionalities and external frameworks that were available. In order to get a secure and stable system, it was needed to get involved with all those functionalities and external frameworks. Getting into these functionalities and frameworks could require effort similar to the amount needed for the base framework. Time that did not exist in this thesis. One of the biggest issues was connected to the load balancing mechanism. To use load balancing, the system was forced to use the built-in mechanism provided by the frameworks. This caused problems for the most prioritized functionality large data handling which was not acceptable in the targeted system.

A big advantage with microservices compared to the monolithic system that was previously stated is that the system is able to pursue newer technologies as each microservice is supposed to be independent from the rest of the system. However, as can be observed in the implementation chapter, some functionalities in these frameworks were closely bound to some specific framework. In Spring Cloud for example, Consul was used as the service discovery mechanism. If another microservice was to be created using another type of framework, it still had to be registered with Consul in order to keep the system working together as a single unit. The same thing could be observed with VertX clustering and event bus. New microservices would have to talk through the same event bus as well as using (in this case) Hazelcast as the clustering method. This can be a complex solution if the framework would not provide a way of doing this. Taking this into consideration it could mean that it might not be that easy to switch out components depending on what type functionalities is used from the various frameworks.

Method

Choosing frameworks

The first step in this thesis was to choose suitable frameworks to use for an implementation of the microservice system. As there was no official regulated place of gathered information about these frameworks we had to find these by ourselves. We used many different types of sources in order to find as many frameworks as possible. Using this method, some frameworks could have been missed. This has to be taken into consideration when looking at the matrix as it can be incomplete.

When it comes to finding what functionalities each framework provided, there was a limitation using the chosen method. As stated in the Discussion - result chapter, an implementation and more understanding of each framework was needed in order to know exactly what functionalities they provided. Because of the limited time, it was not possible to give each found framework the attention needed to make sure that it had a specific functionality or not. In order to reduce the number of missing functionalities, each frameworks documentation was revised twice.

Choosing the framework had to take the previous flaws into consideration. As there could be missing functionalities in the matrix, it was not possible to rule out frameworks simply because they had one less functionality. If this would be the case, a subjective analysis had to be made. A subjective analysis is not very scientific but it was the only method within the time limitation. A better way would have been to implement a prototype of each framework and then decide which framework that suits the implementation the most, although it would require significantly more time.

Implementation

The three steps provided by the IBM article[13] were performed in order to make the monolithic system microservice compatible. Since the system that was to be implemented as microservices was relatively small, each step was not needed to be performed. However, each step was carried out thoroughly as it was highly important to make sure that the thesis can be replicated in the highest possible degree.

Analyzing the framework based on the implementation

Analyzing the frameworks and the implementations was a central piece of this thesis as it provides the insights of how well the system could be implemented using them. The questions were set up in order to create a structured way of analyzing the frameworks and the implementations as well as to make the thesis more replicable.

A better solution with the entire thesis in mind would be to involve more developers in the process of implementing and analyzing the frameworks to get a more accurate representation of the result. However, this was not possible as it would require the participants to be present through these phases.

CONCLUSIONS

The different frameworks provided a lot of different functionalities that can help reduce the complexity of the microservice system. These functionalities can be seen in appendix B. It was also shown that some of the frameworks nearly covered all of the functionalities that was found. With this in mind, a combination of different frameworks or libraries is not truly needed in order to cover the functionalities. However, it should be mentioned that some functionalities are integrated from other frameworks or libraries directly into the frameworks themselves.

When it came to getting a system up and running with all functionalities there weren’t many problems and went swiftly although in order to implement all the functionalities for a specific system, it could be hard to find a good and stable solution. The problems were often connected to the load balancing mechanism as it limited how the services could interact with each other.

The results also showed that the frameworks embraced the benefits of the microservice architecture in some aspects such as maintainability and scalability. When it came to the ability to change frameworks in the pursuit of newer technologies, the frameworks caused some issues as their functionalities could force some solutions such as service discovery to be universal, and thereby required across the entire microservice cluster.

FUTURE WORK

This thesis showed that there were some problems correlated with the load balancing mechanism and the event bus. Another way of handling the clustering and load balancing is by using container orchestrators such as Kubernetes[15]. These frameworks handle clustering and load balancing on a different level than the Microservice frameworks. Therefore, a study can be conducted in order to find out if for example Kubernetes is a solution to load balancing issues faced in this thesis.

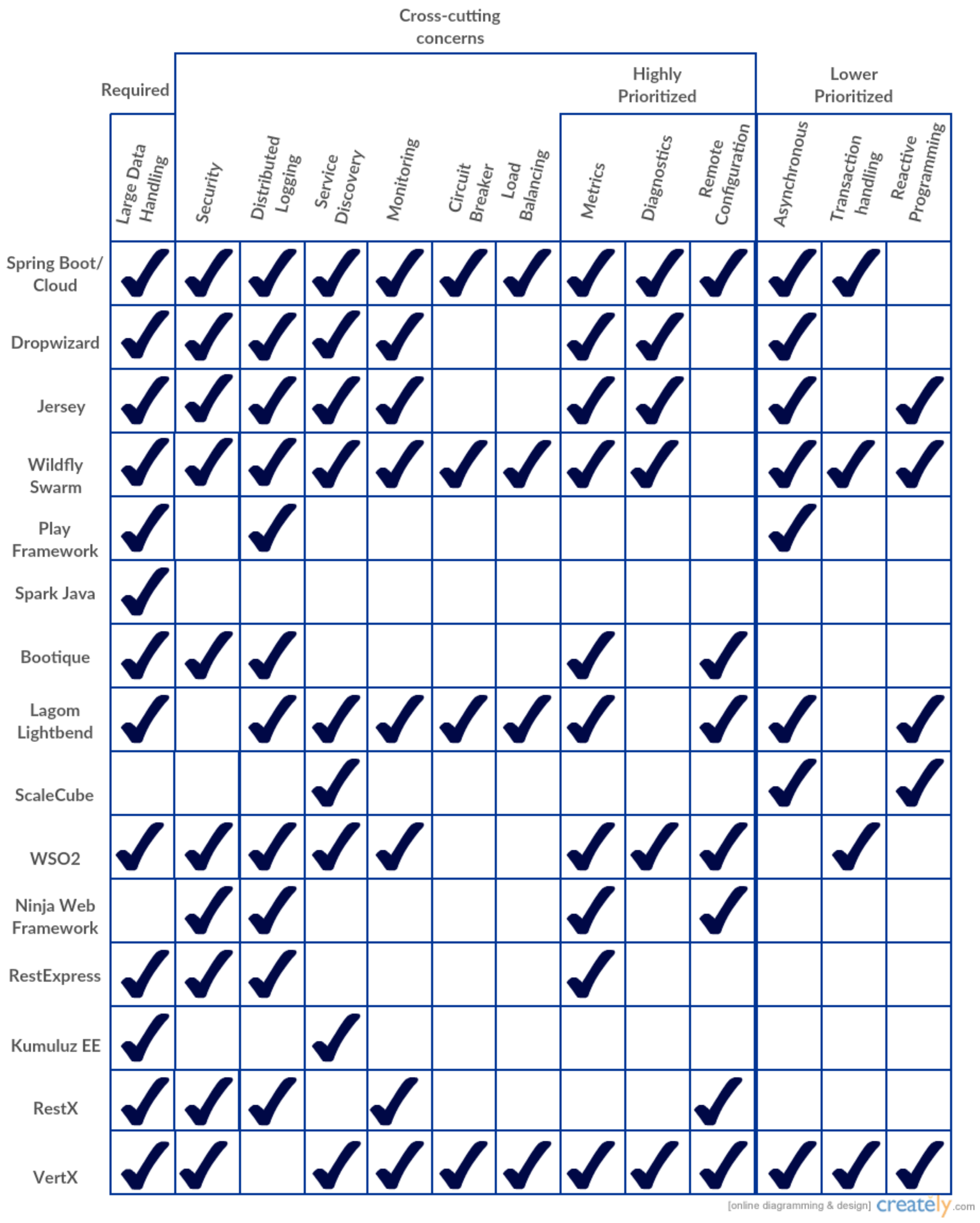
REFERENCES

1. Namiot, Dmitry, and Manfred Sneps-Sneppe. "On micro-services architecture." *International Journal of Open Information Technologies* 2.9 (2014).
2. M, Fowler. (2015, June, 1). Microservice Trade-Offs [Online] Available: https://martinfowler.com/articles/microservice-trade-offs.html
3. Schilling, Melissa A. "Toward a general modular systems theory and its application to interfirm product modularity," in *Academy of management review* 25.2, 2000, pp. 312-334.
4. Thönes, Johannes. "Microservices." *IEEE Software* 32.1 (2015): 113-116.
5. Newman, Sam. “Microservices,” in *Building microservices*, first ed. Sebastopol: O'Reilly Media, Inc., 2015, pp, 1-11.
6. Saleem, Muhammad Qaiser, Jafreezal Jaafar, and Mohd Fadzil Hassan. "Model driven security frameworks for addressing security problems of Service Oriented Architecture." *Information Technology (ITSim), 2010 International Symposium in*. Vol. 3. IEEE, 2010.
7. Stubbs, Joe, Walter Moreira, and Rion Dooley. "Distributed systems of microservices using docker and serfnode," in *Science Gateways (IWSG), 2015 7th International Workshop on*. IEEE, 2015.
8. C, Richardson. (2015, June, 1). Pattern: Microservice chassis [Online] Available: http://microservices.io/patterns/microservice-chassis.html [Accessed: 05-May-2017]
9. Dragoni, Nicola, et al. "Microservices: yesterday, today, and tomorrow." *arXiv preprint arXiv:1606.04036*, 2016.
10. Papazoglou, Mike P. "Service-oriented computing: Concepts, characteristics and directions," in *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*. IEEE, 2003.
11. Cardellini, Valeria, Michele Colajanni, and Philip S. Yu. "Dynamic load balancing on web-server systems," in *IEEE Internet computing 3.3*. IEEE, 1999.
12. M. Nygard, *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007.
13. Refactoring to microservices, Part 1: What to consider when migrating from a monolith. https://www.ibm.com/developerworks/cloud/library/cl-refactor-microservices-bluemix-trs-1/
14. “Spring Cloud,” *Spring Cloud.* [Online]. Available: http://projects.spring.io/spring-cloud/. [Accessed: 20-Mar-2017].
15. “Production-Grade Container Orchestration,” *Kubernetes*. [Online. Available:]https://kubernetes.io/ [Accessed: 11-May-2017].

Appendix A - List of Microservice frameworks

All data for the frameworks was collected during the time period 2017-03-20 to 2017-03-31.
Spring Boot/Cloud https://spring.io/ version Camden SR6
Dropwizard http://www.dropwizard.io/ version 1.1.0
Jersey https://jersey.java.net/ version 2.17
Wildfly Swarm http://wildfly-swarm.io/ version 2017.3.3
Play Framework https://www.playframework.com/ version 2.5.13
Spark Java http://sparkjava.com/ version 2.5.5
Bootique http://bootique.io/ version 0.21
Lagom Lightbend https://www.lightbend.com/platform/development/lagom-framework version 1.3.1
ScaleCube https://github.com/scalecube/scalecube version 1.0.3
WSO2 http://wso2.com/products/microservices-framework-for-java/ version 2.0.0
Ninja Web Framework http://www.ninjaframework.org/ version 6.0.0-rc1
RestExpress https://github.com/RestExpress version 0.11.3
Kumuluz EE https://ee.kumuluz.com/ version 2.1.1
RestX http://restx.io/ version 0.34
VertXhttp://vertx.io/ version 3.4.1

Appendix B - Matrix, frameworks-functionalities



	Required	Cross-cutting concerns											
								Highly Prioritized			Lower Prioritized		
	Large Data Handling	Security	Distributed Logging	Service Discovery	Monitoring	Circuit Breaker	Load Balancing	Metrics	Diagnostics	Remote Configuration	Asynchronous	Transaction handling	Reactive Programming
Spring Boot/ Cloud	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Dropwizard	✓	✓	✓	✓	✓			✓	✓		✓		
Jersey	✓	✓	✓	✓	✓			✓	✓		✓		✓
Wildfly Swarm	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
Play Framework	✓		✓								✓		
Spark Java	✓												
Bootique	✓	✓	✓					✓		✓			
Lagom Lightbend	✓		✓	✓	✓	✓	✓	✓		✓	✓		✓
ScaleCube				✓							✓		✓
WSO2	✓	✓	✓	✓	✓			✓	✓	✓		✓	
Ninja Web Framework		✓	✓					✓		✓			
RestExpress	✓	✓	✓					✓					
Kumuluz EE	✓			✓									
RestX	✓	✓	✓		✓					✓			
VertX	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

[online diagramming & design] creately.com