Department of Computer Science

Distributed and Self-organizing Systems Group

<TODO: Art der Arbeit>

<TODO: Titel der Arbeit>

<TODO: Vorname Autor> mpc

Chemnitz, 9 January 2019

**Examiner:** <TODO: Prüfer>

**Supervisor:** <TODO: Betreuer>

**mpc, <TODO: Vorname Autor>**

<TODO: Titel der Arbeit>

<TODO: Art der Arbeit>, Department of Computer Science

Chemnitz University of Technology, 9 January 2019

Aufgabenstellung

Many web applications grow over time to accommodate the new requirements of their users. With every newly added feature, the application becomes more interconnected and harder to scale and maintain. Microservices architecture was created to help developers scale their applications with ease without added complications for modifications or maintenance.

This new architecture suits both ends of the application, the Frontend and the Backend as well. Backend requirements will be handled by small tasks and each task will be performed by a microservice. On the other side, the Frontend will be divided into different parts and each part will be rendered by one micro frontend. As this implies communication between all micro parts, trust plays especially with parts of different parties a central role. The objective of this thesis is to research the workflow, tools and guidelines involved in creating a web application based on this architecture, while solving trust concerns via embedded content trust. To achieve this, a Blog will be developed out of micro frontends and microservices. The relationship among micro apps will be addressed regarding their content trust. A solution will be created to help the different parts of the application to establish a context-wise trust.

The objective of this master thesis is to find an approach or a combination of approaches to solve the previously mentioned problem in the context Microservices and Content Trust. This particularly includes the state of the art regarding microservices and trust in computer science with reference to Content Trust. The demonstration of feasibility with an implementation prototype of the concept is part of this thesis as well as a suitable evaluation with exemplary use cases.

Table of Contents

[List of Figures vii](#_Toc534297102)

[List of Tables ix](#_Toc534297103)

[List of Listings xi](#_Toc534297104)

[List of Abbreviations xiii](#_Toc534297105)

[1 Introduction 1](#_Toc534297106)

[1.1 Current situation 2](#_Toc534297107)

[1.2 Motivation 3](#_Toc534297108)

[1.3 Problem 5](#_Toc534297109)

[1.4 Objective 9](#_Toc534297110)

[1.5 Outline 10](#_Toc534297111)

[2 State of The Art 11](#_Toc534297112)

[2.1 Requirements 11](#_Toc534297113)

[2.1.1 Requirements of Microservices and Micro frontends 12](#_Toc534297114)

[2.1.2 Requirements of Content trust 17](#_Toc534297115)

[2.1.3 Requirements of Developers and Users 21](#_Toc534297116)

[2.2 Literature review 22](#_Toc534297117)

[2.2.1 Microservices and Micro frontends literature review 22](#_Toc534297118)

[2.2.2 Trust literature review 29](#_Toc534297119)

[2.3 Analysis 36](#_Toc534297120)

[2.3.1 Microservices analysis 37](#_Toc534297121)

[2.3.2 Content trust analysis 39](#_Toc534297122)

[3 Concept 42](#_Toc534297123)

[3.1 Concept of microservices 43](#_Toc534297124)

[3.2 Concept of content trust 46](#_Toc534297125)

[3.3 Overall structure 52](#_Toc534297126)

[Bibliography 55](#_Toc534297127)

[Appendix A Bezeichner für Anhang A 57](#_Toc534297128)

[A.1 Bezeichner für Anhang A.1 57](#_Toc534297129)

[Glossary LIX](#_Toc534297130)

[Index LXI](#_Toc534297131)

# List of Figures

[Figure ‎2‑1 Scale Cube [35] 27](#_Toc534297190)

[Figure ‎3‑1 Overall structure 52](#_Toc534297191)

# List of Tables

[Table ‎2.1 Comparing monolith and microservices [34] 28](#_Toc534297181)

# List of Listings

[Listing 2.1 einfaches XML-Beispiel **Error! Bookmark not defined.**](#_Toc332900301)

[Listing A.1 Mit Alt+ F9 bearbeiten (hängt von Heading 4 statt von 1 ab) 57](#_Toc332900302)

# List of Abbreviations

|  |  |
| --- | --- |
| **ABC** | Alphabet |
| **123** | Zahlenreihe |
| **HTML** | Hypertext Markup Language |
|  |  |

# Introduction

Traditional web applications are a software that comprises several parts, all those parts come together to form the final product. But at the end this product will look like as it is made out of one big unit. This unit is composed of few different layers on top of each other but each layer is tightly coupled with the other layers [46]. This architecture is sometimes called three-layered architecture that has presentation, business and data as its three layers.

Once the application is ready to be deployed, developers will have to approach it as if it is, essentially, composed of two parts that can’t be divided into smaller units. Such division can be very useful when there’s a failure and the application is not running as it is supposed to. In this case, developers could isolate the malfunctioned parts. Hence the system will keep providing some of its services to the clients while also being maintained. Moreover, such possibility of parts isolation gives developers more flexibility and features when trying to find solutions for the problems.

Furthermore, updating a three-layered-based application and adding new features to it becomes harder the bigger the application is [34]. Application parts will be more interconnected and making changes to one part could result in needs to make changes to other parts. Hence developers might have to modify many parts of the system to allow for the new updates to take place.

Microservices architecture tries to overcome the challenges that are imposed when the application is created as a three-layered application. The idea of microservices is to have the system composed of many independent small parts that work together to form the final web application. This concept can also be projected into the frontend part of the application resulting into the micro frontends architecture. In essence, having the frontend as a combination of many small independent micro frontend apps.

## Current situation

Microservices is still a new concept, although some companies have already migrated to the microservices architecture such as Amazon and Netflix [2]. There are companies facing many issues that could be solved by this new architecture. Yet those companies are hesitant in moving forward and migrating their application to the microservices architecture.

The idea of refactoring an existing application into a new one built using another architecture is not easy and brings with it many challenges [1]. Some of those challenges are still waiting to be addressed.

There are still many questions to be asked when thinking about the microservices architecture, for example: when thinking about the functional requirements of the application, developers should think about how their application can be divided into smaller parts. And whether following certain standards can be helpful in making a final better product. There can be other questions regarding the nature of the microservice itself. Such as the size of each microservice. Moreover, the communications between microservices are another challenge that deserves thoughtful consideration.

On the other hand, some concerns related to the frontend of the application can also be thought about. In principle, the frontend is what the user see and interact with. As an example for these concerns what if the frontend becomes also complex and needs to be divided into micro frontends. The concept of microservices can further be projected into the other side of the web application. Microservices is not just concerned with the backend side of the system, but it can be also applied to the frontend as well. When microservices concept is applied to the frontend it is called micro frontends [3].

Developers should find ways to help micro frontends exchange data between each other without violating the isolation and independence of each micro frontend. Since each micro frontend should be able to interact with other micro frontends while keeping the application loosely-coupled.

Furthermore, a mechanism should be in place to help microservices trust each other. Trust here doesn’t just mean believing each microservice to what it claims it is. But having a system that helps microservices to trust the behavior of each other.

There is still no standard definition of microservice architecture and there’s no clear guidelines of how an application based on microservices should be built [32]. Over the last few years some characteristics for a microservices-based application have been developed and some general basic outlines are now commonly used [1-2,31-34].

* Many small units: A microservices-based application should consist of more than one component: Unlike three-layered applications, a system built using microservices architecture should be composed of multiple components, each component is self-contained. This way the application can be changed, updated and modified whenever is needed. In this case, each change will be applied to only the concerned component itself and not the entire application.
* Simple Routing: Components in a microservice-based application will have a simple workflow. They will take an input, process it and then forwards the result.
* Decentralization vs centralization: An application based on microservice architecture is built out of many different components and each has its role. Sometimes there’s a central body that manages the interactions between the services. While other times, no one big unit moderating the communications between microservices.
* Different technology stack: The development cycle of a microservices-based application involves having different teams working on different microservices. Each team can then choose development technologies and tools that are most suitable for their own microservice.

## Motivation

The current architecture that’s used heavily in building web applications is composed of layers built on top of each other. Each layer is responsible for a part of the application [34]. As mentioned in the first section of this chapter, it usually consists of three layers on top of each other’s [2] [46]. Moreover, some applications could end up having their logic layers divided further into more layers.

Although a three-layered web application is divided into layers, yet the application is still very tightly coupled [16]. There’s a great amount of connections between the layers. As a consequence, the system, of course, will be hard to maintain and update [34]. Each problem in the Backend could cause failure of the system, where no information could be processed or stored in the database. On the other hand, each problem in the User Interface could cascade to prevent data from flowing to the backend.

In some cases, fixing the issues could cost developers to make some modification. This leads to the problems of updating the system. Where Updating a traditional layered application is another big problem faces developers and business owners [34]. When needing new features or updates to fix the newly discovered issues, developers could find themselves facing two situations: Not being able to achieve the wanted updates and fixes as much as they’re supposed to. And on the other hand, having to perform huge system modifications and changes to accommodate the new desired features and changes.

These situations happen frequently and usually cost money, time and lots of efforts. Microservices architecture was developed to make applications more flexible. With microservice-based application, the system is now more accepting for changes. Developers don’t need to make great modification for the system to adapt a new feature. When a system failure happens or when a problem is discovered, developers have the ability to isolate the problem and fix it quickly.

With micro frontends, the frontend of the application is composed of many different small self-contained applications. Instead of having the frontend as a one unit written in one framework such as Angular or React JS. The frontend can be written and developed as a sum of smaller frontends. Each small frontend performs a very specific task and can be called or rendered whenever it is needed.

In micro frontends architecture, each part of the application is developed independently, as a standalone application. This means that developers could choose Angular for one micro frontend and then choose React JS or Vue.js for another part. Each development team can focus on their micro frontend. This division also helps with separation of concerns. Designers of the system will be able to divide it into small tasks. Then assign the development of those small tasks to different teams. At this point, each team will not have to bother with interacting with other teams. They could focus on their task, its functionality, problems, and its deadline.

Although microservices and micro frontends architecture helps to solve many pressing problems for the development of web application. Yet such architecture brings its own challenges and problems.

Putting all the small parts together is not an easy task. Developers should think of the best way of having microservices interact with each other. Not just in the sense of sending and receiving information but also microservices should be able to exchange sensitive data. A system of trust between microservices could help make the interaction more secure.

Such trust is very important, especially, when there is a need for using third-party microservices. In this case microservices might end up exchanging sensitive data such as user logins and passwords or maybe even bank details. In such situation microservices should not start exchanging this type of data without verifying and knowing more details about the microservices on the receiving end. Here a system of content trust can play a great role. It can insure that exchanging data between microservices only happens after the behaviour of each microservice has been verified. Once it is known that no microservices have any harmful code, or bad intentions then data exchange should take place smoothly between microservices.

## Problem

Having microservices architecture in its early days means that not many resources are available. Moreover, not enough research is done yet to help developers find answers for their problems [32]. When dealing with microservices architecture two types of scenarios could be happening:

One case would be that; the Web application already exists using monolithic architecture. But there is a need to migrate it into microservices architecture for many reasons including:

* The application is not scalable anymore: New features are needed but scaling the application to fit those new features in is not possible without making big alternations to the application itself.
* The application has some serious problems: Developers are not able to find suitable solutions for those problems without creating many additional parts or performing huge changes to the base code.
* Some parts of the application fail constantly causing the whole system to fail: In this case developers are not able to isolate the failing parts. Thus whenever a failure happens the system does not continue to run while the problem is being fixed.

The other case would be that; developers want to develop the required system from the beginning based on the microservices architecture. One reason for this can be because the application is expected to grow. With microservices architecture, it is easier to scale the system as much as needed compared to monolithic architecture [34]. Another reason would be that the application has a complex nature and requires different technology stack for its various parts. Hence building it using microservices architecture will help developers to use different tools and technologies for different parts. This situation also applies for the frontend. In this case, the frontend could be complex and many special requirements are needed thus developers can divide it into smaller apps. With each app they can use different tools, frameworks and languages instead of having to use one toolset or one framework for developing the whole frontend.

Microservices architecture is basically one variant of Service Oriented Architecture (SOA) [32], but operations conditions are not quite the same as with traditional SOA [39]. With microservices architecture there are many small independent parts operating. Each part is providing or performing one small task. Sometimes microservices need to exchange data with each other and of course with the frontend.

Developers have to decide what kind of communication methods and protocols should be used among microservices. Representational State Transfer (REST) is one architecture that helps developers when creating web services. Another option is Simple Object Access Protocol (SOAP) which is a protocol for exchanging messages in a web services environment. There are many technologies that could be used in a SOA Including:

* Web Processing Service (WPS)
* WSCL - Web Services Conversation Language
* XML-RPC - XML Remote Procedure Call
* JSON-RPC

The most famous ones and most used is REST and SOAP [40]. The decision for using a specific protocol depends on the developer and how they want data to flow between the different parts of the application.

On the other hand, when developers decide to use micro frontends to render the frontend of the application. They should also think of how data is going to transfer between the different micro frontends and how communication between the many frontends and the microservices is happening.

In essence, micro frontends are autonomous independents parts [1]. These parts are not divided according to how they’re going to render on the screen. They’re divided according to their functionality or even the business they support. For example, in an online store, the distribution of micro frontends could be as follows: One micro frontend for the displaying of the products, another would be for the product details, a third micro frontend could be for adding items to the cart. And maybe a fourth for the check out and payment. Other micro frontend can be focused on other functionalities of the web site such as handling user data. For example, registration and creating a new account. This task can be assigned to a micro frontend, or updating user information. One micro frontend can also be created to handle security concerns such as when a user forgets his passport.

At the end once each micro frontend is developed, the final frontend should be able to contain all the micro frontends when they are needed. There are different technologies for stitching micro frontends together. Two approaches are available for putting all the parts together, either client-side or server-side. For each method there are different tools and technologies available, each has its own good points and negative ones. On the server side, such technologies include:

* NGINX as reverse proxy
* Varnish with ESI module enabled

While on the client-side, developers could use one of the following technologies:

* Single SPA library
* Web Components
* iFrame

For both methods there are more technologies and tools than the previously mentioned ones, but those are the most famous so far. Choosing the right technology depends on the application being developed and if there’s a need for communication between the micro frontends or not.

Moreover, security concerns are a big challenge under microservices architecture. How would microservices exchange particular sensitive information such as passwords or bank details? This concern would be stronger if microservices were not all developed inside the same company. Such scenario could rise when small companies want to build their applications using microservices architecture. In this case, when having a small team of available developers, one could think of using ready-made solutions. Developers could use third-party microservices to save time and money. But doing this imposes more security concerns. Hence microservices need to establish a trust mechanism to trust each other before exchanging data. Moreover, when adding new microservices to an already running system. The existing microservices need to be able to trust the newly added ready-made solutions. The kind of trust discussed here is related to the behaviour of each microservice. After all, malicious or harmful microservices could hide their true intentions by expressing different behaviour while a harmful one is practiced behind the scenes.

There should be a way to help microservices trust each other without having a human intervention. When the application is getting bigger and embracing hundreds of microservices. Which in turn might also be depending on other microservices to run, then checking the behaviour for each one of them by the developers could end up being an endless task. Developers could start checking the microservices they adapted into their applications. But then shortly find themselves checking microservices that are used by the microservices they used. Hence keep moving backwards in the string of microservices.

Each microservice should be doing its own task of verifying its behaviour with other microservices. When one microservice fails in doing so, then developers should be alerted.

## Objective

This thesis will try to fill the gap that still exists in the literature regarding some of the mentioned concerns. The focus will be on building a system out of microservices and micro frontends while providing a solution for security concerns, in particular content-trust among microservices.

Building a solution that’s ready to be deployed based on microservices and micro frontends architecture is still missing in the literature. Moreover, having a practical example of content trust between applications is also absent. Hence building a content trust system between microservices will fill a gap in the literature. And could add to the work done in microservices as well as content trust.

This research will try to find answers to some of the questions mentioned in the previous sections. While there are many questions and uncertainty to explore and research, this thesis will not try to find answers for every possible problem resulting from building microservices-based web applications. The workflow will be building of a Blog based on the microservices architecture, the development will involve using the latest technologies and tools to build the Blog. Solutions for faced challenges will be documented which then can be used by researchers, students or anyone interested in the microservices architecture. It could also be used by developers who are building a microservices-based web application to help them overcome some of the challenges that could face them.

Moreover, this research will also focus on providing a solution for security concerns, more specifically, the problems of content trust among microservices. A method will be created to help microservices trust each other context-wise. This trust is not just about verifying each microservice its identity to the other microservices, but it is also about having a mean or way of verifying the behaviour of each microservice. Such method could be very useful when developers have to add different microservices from other sources and have to check their behaviour and trust it.

On the other hand, this thesis will not try to provide a full workflow and complete guidelines for building microservices-based web applications. Such attempts require years of research and will most likely be a never ending one. Since tools, frameworks and technologies are ever changing and developing and so are the ways of developing web applications.

## Outline

The following chapters will talk more in depth about the microservices architecture and micro front ends.

The next chapter of this thesis will be State-of-The-Art, in this part, discussion of requirements for microservices-based application and content trust will be presented. Moreover, a literature review and analysis of the requirements against the literature review will be provided.

Third chapter will focus on the concept of building microservices and securing their behaviour with content trust. This part of the thesis will try to weight the benefits as well as the negative sides of building applications based on the microservices architecture. It’ll investigate how a system with many moving parts can work and offer a stable and seamless experience to its users. At the same time have very clear division and separation of functionalities into small autonomous collaborating tasks. How content trust can play a role in the microservices architecture will be provided. A method of content trust among microservices will be discussed and inspired from the content trust of the web. The discussion will also pay attention to the principles of micro frontends and the different methods, server-side as well as client-side, of combining micro frontends will be presented.

Chapter four will focus more on the practical side of the research. This chapter will discuss the development and building of a Blog based on the microservices architecture. The workflow will be presented and mistakes that have been done during the development will be discussed to help make other researches aware of them. The used tools will be explored and the reasons behind using such tools will be made clear.

The developed method of content trust among microservices and micro frontends will be presented, how this method is developed will also be outlined and discussed.

Last chapter is number five, in this part of the thesis, an evaluation of the development and carried out research will be performed. This chapter will show the negative as well as the positive sides of the research and the implementation. It’ll also discuss the difference in the productivity when using specific tools or technologies. It’ll show the similarities and dissimilarities when selecting specific tools over others

# State of The Art

This chapter will be composed of three parts:

1. Requirements
2. Literature/State of the Art Review
3. Analysis

The first part will discuss and analyse the requirements for an application based on the microservices architecture. It’ll also discuss the requirements of content trust between microservices. Hence it’ll be composed of two sub sections. One for microservices and the other for content trust.

The second part will layout the literature review for microservices and also for content trust. As with the first section, this one will be mainly divided into two sections as well. One for microservices and the other for trust in general and content trust.

Last part of this chapter will be connecting the first two parts together. It’ll check the literature review against the requirements analysis.

## Requirements

This part of the thesis will present requirements analysis for a microservices-based architecture that relies on content trust to verify the behaviour of its different components. The discussion will be split into three parts:

1. Requirements of Microservices and micro frontends
2. Requirements of Content trust between microservices
3. Requirements of Developers and Users

The reason for such division is that, microservices and micro frontends are architecture. Hence they have their own set of rules and requirements when it comes to building an application based on this new architecture.

On the other hand, content trust is one of the security concerns, and with just like any application, security concerns will be addressed throughout the development of the proposed Blog. Yet the focus here is on the trust among different microservices, especially those coming from different sources/developers. Thus, this chapter will allocate a subsection in the requirements analysis to discuss the requirements of the content trust of the application.

A third section will discuss from the prospects of developers and users.

### Requirements of Microservices and Micro frontends

A system based on microservices architecture consists of different small pieces of code. Each small piece is an application that can be deployed independently. It can also be updated and modified while no need for any modifications of the other small apps because such an update took place for one or more microservices. Such architecture, in theory, makes the system loosely coupled. Thus different system parts and components are easy to change, update, modify or even replace. As long as the interface of the new introduced microservices respects the old interface or keeps using the same communications protocols, then the system will continue to function.

Since micro frontends are basically microservices architecture applied to the frontend part of the application. Some requirements should be respected when developing frontends as a group of small independent micro frontends.

Microservices as described by Sam Newman in his book building microservices [1] are basically small independent services, that work together. From this definition, the basic requirements of microservices can be derived.

* Small
* Autonomous
* Has an Interface

Moreover, in [1][5][29] the description continues and more requirements can also be derived

* Resilience
* Scalable
* Easy to deploy
* Reusable
* Automation
* Replicable
* Low influence if the interface changes

The following pages will go in details about each one of the requirements. For each requirement, a discussion of the micro frontends requirements is presented when applicable.

**Small**

The idea of microservices architecture is that the application will be composed of small services. In order to get the most out of microservices, each service should be doing one task. Such focus is tied to the functionality requirements of the business. With having each microservice handling only one task, developers can increase the chances of developing an application that respects other requirements of microservices, such as resilience and scalability.

Each microservice is supposed to be small, the size of each service should be scaled down until it can’t be scaled anymore. Such approach will help to magnify the gains but also adds more overhead [1]. Once each service is very small then it can easily be replaced, isolated, updated or deleted while the rest of the system is still running. On the other hand, having many small dynamic parts in the system will make it harder to manage and could add extra complexity.

The size of each micro frontend is also supposed to be small, where the frontend will be decomposed into small apps and each app will handle a portion of it. For example, one micro frontend for the navigation bar, another for the footer, and more micro frontends will handle the body and other functionalities of the page.

**Autonomous**

Each service should have the possibility of getting deployed independently. Such feature gives applications more flexibility. It’ll help to make the application more loosely coupled. When one or more microservices need updates or changes. Hence the application can benefit from the concept of microservices. Moreover, the isolation of each microservice is important to make the whole system scalable [30].

Furthermore, each microservice is independent enough to allow developers to develop each microservice independently. This includes giving each team the freedom of choosing which toolsets to be used for the development of this particular microservice. Such freedom will help developers to choose the most suitable tool for each service depending on the service itself and regardless of other parts of the application.

This also applies to the micro frontends. Each micro frontend should be developed in a way that makes it as an independent app.

**Has an Interface**

Since the application consists of many small independent parts that work together. This means that each part of the system should be able to provide some form of communication channels to other parts in order to be able to work with other parts and not to be isolated. Hence each microservice should provide an Application Programming Interface (API) that enables other microservices to talk to it and exchange data with it [33]. Such API will enable microservices to support the principle of encapsulation. Where each microservice will have the freedom to hide its internal implementation and expose only a channel of communication.

Furthermore, having each microservice offering an API will help to make the system adheres more to the principles of microservices. Hence allowing each microservice to be updated or changed without affecting the rest of the system as long as it keeps respecting its old API.

**Resilience**

One of the features offered by microservices architecture over monolithic architecture is having more resistance to system failures. When one part of the system, in monolithic application, fails the whole system will suffer from this failure. Such situation could easily deteriorate and in the end might lead to the complete failure of the system. On the other hand, when a service fails in microservices architecture, the system will be able to handle this failure much better [1].

Having each microservice as an independent entity means that each service can be isolated from the rest of the system once it shows signs of failures. Hence the system will be able to keep running and offers its services to clients except for the microservice that failed and currently isolated [5]. This division of functionality leads to even more benefits, as in this case, the system will keep running while developers are working on fixing the failing part.

As for the frontend, in case of a failure in one or more of its micro frontends, other micro frontends should still function. Any problem happening in a micro frontend should not affect the rest of the micro frontends.

**Scalable**

One of the key features of microservice architecture is having systems that can be scaled easily. When the system needs new features in monolithic architecture, developers could end up scaling the whole system [5]. In order for it to handle the new additions to support the new features, where it is difficult to scale an application under monolithic architecture [30]. Such case means developers will spend more time worrying about how the system can be modified to support the new features. Thus more time and resources are spent. On the other hand, adding new features for a microservices-based application is much easier.

It could simply mean adding more services to the system. And so, other parts of the system will not be affected and will not need any intervention from the developers. Such case, means developers don’t have to spend time thinking about how the system can accept the new additions. Hence saving more resources.

The overall frontend should allow for modifications and adjustments. At any time adding new micro frontends should be possible without affecting other micro frontends.

**Easy to deploy**

Developers under monolithic architecture have to redeploy the whole system when they add new features or make changes. Meaning they would need to release new versions of the system with each update. And in case they try to gather each few updates together before releasing a new version then there are more risks in having errors in the released versions [4]. But having the system composed of small independent parts means that adding or modifying is easier. Such modifications will only concern certain parts of the system. Hence developers will be able to have continuous deployment of the system without the need to release new versions each time [1].

Moreover, when something goes wrong after a new modification, it’ll be easier to isolate the new changes and fix the problem. Or even rollback to the old service while fixing the mistakes in the newly updated parts.

**Reusable**

Since an application built using microservices architecture will consists of many different small services, then other system might share some of the same functionality as another one. Hence developers could be able to use some of the services that’s built for one system in another one [5]. This of course means saving more resources.

Furthermore, some services can be reused on different platforms [1]. After building a web application using microservices, the same services can be used to help power a mobile application. Since microservices will have an interface thus making it easy to communicate with each service as long as the communication respects the provided interface. Such feature will help developers release more combatable applications on different platform, faster than ever.

Each micro frontend is an independent small application. As a result, reusing each micro frontend in different applications is possible. Sometimes, few modifications to the styling are required to make the micro frontend more suitable with its new application.

**Automation**

Since services are independent entities that can be deployed on separate machines, developers need to be able to deploy the service and all its runtime environment automatically. It would make sense to deploy the services and their runtime environment manually when the number of services is still relatively small [5]. But when the number of services increases over the development time then developers should be able to deploy services and their runtime environments quickly [4].

**Replicable**

The nature of microservices help in making each one of them a replicable entity. Each microservice has an interface that helps it in communicating with other microservices. Hence replacing one service with another is possible as long as the new service interface respects the old service interface [29].

**Low influence if the interface changes**

Since microservices only method of communication is through API then system developers should strive to keep each interface unchanged. In case a change is absolutely necessary, the influence of such change must be keep minimal [30].

### Requirements of Content trust

This section will discuss the requirements for a content trust system that will be implemented to help microservices verify the behaviour of other microservices. The following pages will try to give more details about what it should be included in content trust system.

Microservices themselves need to have clear rules about how to securely communicate with other microservices especially ones coming from different developers. This communication should allow microservices to make a judgment of whether to trust the other microservices or not.

On the other sides, there are no requirements for the user, the one who is going to use the overall application. Because, content trust is something to be handled between microservices, or a protocol and set of rules to be read and understood by the microservices themselves. And so the user has no role in how the content trust should be and what it should include.

Content trust has more dynamic nature than other types of trust such as policy-based trust. While when performing an identity check the outcome could be one of two. Either the identity has been proven, or the identity failed to prove itself. But with content trust several characteristics, measures and aspects should be taken into account to come to a decision of trust or distrust. Those measures are inspired from web content trust [17].

Those measures differ in how hard it is to obtain the needed information. It all depends on the kind of information that is being retrieved. In order for microservices to be able to check the content trust of one another, each microservice should be able check the following:

* The identity of a microservice
* The type of service
* Service sensitivity check
* Check the number of interacting services
* The evaluation by other microservices
* Age of the microservice
* Trust transitivity
* The nature of the connection with the microservice

On the other hand, content trust can have other more factors to be evaluated. Not all of those factors can be used between services in a SOA. The reason for this, is because some of those factors require real-time human analysis such as: User expertise, bias, incentive, and deception. Moreover, other factors are mostly concerned with evaluating information online and can’t be projected into microservices such as: Agreement, recency, authority, specificity, and likelihood. Furthermore, factors such as limited resources.

**The identity of a microservice**

When an application starts using third-party microservices then there should be a need for verifying the identity of microservices.

A system must be in place to help microservices verify each other and make sure that each microservice is what it claims to be.

If a verification mechanism is implemented as a part of the content trust system, then the evaluation process will be handled sequentially. The system at first must allow for microservices to verify themselves to other microservices. If the verification process was successful, then microservices move on to the next step. In case the verification step failed then the involved microservices could stop. And any planned communication may not take place. This of course depends on how developers want to handle such failure. Moreover, such scenario could actually lead to further steps. The system could include a notification mechanism. Those notifications will signal the ID of the involved microservices and which one failed in authorizing itself. Hence developers can take notes of what happened and make sure that all the microservices of the system are secure.

It is not mandatory that microservices stop any communication when identity check fails. Microservices can always proceed to the rest of content trust check without going through any identity verifications. But having a successful identity verification could increase the positive evaluation of the trust for the concerned service.

**The type of service**

Microservices should be able to check the service that’s provided by any microservices trying to connect with them for the first time. The more sensitive the provided service is the more important such check. There can be different ways that allow one microservice to check the service of the requesting microservice. For example, there could be a record that lists the services provided by each microservice with the identity of each microservice. Once a request has been made, the requested microservice can check the request information of the requestor microservice against this record. It will then be able to verify that the service of the requestor microservice is what it claims to be.

Another method could be done by running a simple test with the requestor microservice using dummy data to first check that it’ll run as it claims to be. This might need some form of artificial intelligence. It will enable each microservice to truly verify if the requestor behaviour is as it is supposed to be or not.

**Service sensitivity check**

The services provided by the different microservice will vary in nature, some will offer simple service, while others will offer sensitive services. For example, some services could offer routing to help the user navigate from one page to another. On the other hand, another microservice will offer a log in service to the user. Such service has a higher sensitivity than the other.

**Check the number of interacting services**

Each microservice should be able to know if there are other microservices that used the service of the requestor microservice or replied positively to its requests. Also each microservice should be able to know if the requests of the requestor microservices have been denied or not. And which microservices denied the requests of the requestor microservice. In case of failing or negative replies to the requestor microservice, then it’ll also be helpful to know which microservices responded negatively to the requestor. And what kind of relationship exist between the requested microservice and those other microservices that responded negatively to the requestor.

**The evaluation by other microservices**

When a microservice receives a request from another microservice, it should be able to see what evaluation the requestor microservice received from other microservices. This evaluation will be the degree of trust each microservice gave to the requestor. By seeing what other microservice thinks of the behaviour of the requestor microservice it’ll then help the requested microservice to increase its own positive or negative evaluation of the requestor.

**Age of the microservice**

Microservice in the system might have different operation age, this difference come from the nature of the architecture of microservices itself. Microservices are added gradually to the system. Hence some will be added in the early stages while other will be added at a later stage. Moreover, some microservices will be replaced by new microservices. And some new microservices will also be added to fulfil new requirements or fix a newly discovered bug. When making requests to other microservices, the requested microservice should be able to check the age of the requestor. The older the requestor the ore trust the requested microservice could have about the requestor. Of course, there should be a very trusted method of checking the age of each microservice by the other microservices.

**Trust transitivity**

The relationships between microservices could play an important role when deciding which microservices to trust or not. A transitive system could be in place to help transit the trust of a microservice from one to another. For example, three microservices A, B and C. the content trust relationship between them is as follows: A trust B, and B trusts C. the two microservices A and C have never interacted before. When C makes a first request to A. the microservice A should be able to check the microservices that it already trusts and their trust decision regarding C. Since A already trusts B, and B already trusts C, then A could come to a positive conclusion and increase its evaluation of trust to C based on its trust to B which in turn trusts C.

**The nature of the connection with the microservice**

Microservices have interfaces that help them interact with each other and exchange information, this can be in a form of an API calls over a protocol. Such interface could be helpful when deciding to trust a microservice or not. When microservices have similar methods for API calls and use the same data type and containers for exchanging data such as JSON or XML then this could be considered a positive point when deciding to trust the behaviour or a microservice or not. The reason for this is because if the microservice was installed by the same developers then it is more likely that they will try to unite the way microservices interact with each other inside their system.

When microservices have a system that helps them in identifying other microservices and also helps them in making a decision about the behaviour of other microservices. Then the system will exhibit an intelligent behaviour that will help in detecting and eliminating security threats. In order to obtain a decision about the behaviour of other microservices, each microservice must rely on different set of factors and guidelines. These factors and norms will help microservices come to a decision of whether they should trust another microservice making a request and trying to exchange information with them.

As can be seen, content trust requirements of microservices are a combination or policy-based trust and reputation-based trust. Microservices have to make a decision of trust the content and behaviour or other microservices based on a combination trust system.

### Requirements of Developers and Users

Developers are the person or the group of people who are creating the application. From their point of view when trying to handle content trust between microservices, there can be two cases:

* All microservices are developed in-house
* Some microservices are developed by a third-part

When having all the microservices as an internal product, something developed by the same company then trusting the behaviour of different microservice is not as important. The reason for this is that when developers develop a microservice they’ll be able to trust its behaviour directly. They will be sure that no hidden intentions are implemented or any harmful behaviour intentionally in place. Furthermore, some concerns should be taken into consideration if in the future some microservices will be introduced from a third-party.

On the other hand, when some microservices are developed by a third-party, then developers must make sure that microservices of both sides will be able to communicate with each other to verify their behaviour and build their trust. Developers should be able to adapt any third-party microservices. In such a way make it able to provide the requested information. Such information as identity verification, age of operation, the type of service provided, and so on. This information will help microservices to make a decision of whether they should trust the behaviour of a certain microservice or not. Failing to do so while having a content trust system implemented could create many problems.

On the other hand, users of the system don’t have any content trust requirements. The reason for this is because content trust is something related to the interaction between microservices themselves. It is all internal behaviour.

## Literature review

This section presents a literature review for microservices as well as trust in computer system.

### Microservices and Micro frontends literature review

Micro frontends architecture is a new concept. Not many resources are available about it and not much discussion in the literature so far. Nevertheless, some resources online have discussed it and the following lines will present definitions and discussions about micro frontends.

Micro frontends are similar to microservices; the difference is that microservices are applied to the backend part of the system while micro frontends are applied to the frontend of the system. Many of the same concept of microservices can be applied to the frontend of an application and that would result in micro frontends. As explained in [43]. Moreover, in [42] micro frontends are defined as “independent frontend components”. It goes on to explain that the system can be split into parts and each part could have its micro frontend, a microservice, and maybe a database according to [42].

Instead of writing the application as a monolithic one, it can be divided into small parts. “an approach to developing web application as a composition of small frontend apps” as defined in [44].

The writer in [45] agrees with [43] where it mentions that, micro frontends is the concept of microservices applied to the frontend. Furthermore, [45] presents an important feature of micro frontends which is developing each part using the right technology. It is described as “use different technology for different services”. Services here refers to micro frontends.

On the other hand, microservices architecture as defined in [30] is a way to develop an application that’s composed of a group of small independent services. Similar definition is given in [33] where it says: “Microservices is an architecture style, in which large complex software applications are composed of one or more services”. Furthermore, microservices are also referred to as small independent services that work together [1]. The definition given in [5] also agrees with the above mentioned definitions, it states: “Microservices are relatively small, autonomous services that work collaboratively together”. Writers in [32] go on explaining that microservices is a product of Service Oriented architecture (SOA). The same is also mentioned in [31]

The size of each microservice has also gained a lot of attention when discussing microservices. “A microservice is a separate entity” that can’t get any smaller in size [1]. While [5] mentions that each microservice must implement only one business requirement. And it’s been argued that no rules have been given to how small each service should be. [33] Also agrees with [5] where each microservice should try to represent one business functionality.

Moreover how small each service is can be different from one system to another [5]. Other researchers also suggest that counting how many lines of code each service is should give developers a good measure on how big each microservice is [29]. It is also simply mentioned in [29] that services should be small, “The name “microservices” conveys the fact that the size of the service matters; obviously, microservices are supposed to be small”. Furthermore, counting the number of days each service takes to be developed is also another measure. In [1] it is advised that each microservice should not take more than two weeks to be developed. Just like other researchers [32] describes applications built with microservices architecture as a composition of small services. [33] Mentions that each microservice should only be concerned with implementing one task.

The literature also discuss how microservices should communicate between each other. At first it is mentioned in the definition of microservices architecture, that microservices collaborate with each other [1] [5]. This implies that microservices should find a method or mechanism to turn such collaboration into an actual exchange of data. [33] Mentions that each microservice should adopt an API where “They communicate with each other using language-neutral application programming interfaces (APIs)”. It is also mentioned that microservices only communicate with each other using network calls [1]. Moreover, [31] agrees that microservices should implement APIs interfaces to help communicate with each other, it states: “They are characterized by well-defined and explicitly published interfaces”. However. [29] Doesn’t go in details about how each microservice can communicate with other microservices, it is only mentioned that “Communication between microservices is distributed communication”. Researchers in [30] agree that microservices should implement API and that “Services can communicate with some lightweight mechanisms”.

The characteristics of microservices are also discussed by many writers. Many of them agree that each service should be small as in [1] [5] [29] [31] [32] [33]. The same agreement happens when talking about the independence of each microservice. For example, regarding the independence of each microservice [30] explains: “each of the services is running in its own independent process”. The same thought is mentioned in [1] and [31] where both of them mention the word autonomous to describe the independence of each microservice. [31] “Each service is fully autonomous”. While [1] says: “Microservices are small, autonomous services”. Same with [5] where the word “autonomous” is also mentioned, it is stated: “Microservices are relatively small, autonomous services”. Also [33] agree that microservices should be independent from each other.

The relationships and the effects each microservice has on other microservices is also discussed. [31] Mentions that when changing the implementation of a microservice other microservices should not be affected. [33] Agrees that microservices should not affect each other, the term “loosely coupled” is mentioned to describe the nature of the relationship between microservices. The word “isolation” is mentioned in [1] to describe how microservices should not affect each other when changes happen. On the other hand, such isolation could introduce “overhead” [1]. The write goes on and describe that microservices should be able to change independently from each other. It is stated: “The golden rule: can you make a change to a service and deploy it by itself without changing anything else?” [1]. Researchers in [5] agree with the concept of having microservices independent from each other, they say: “Loose coupling is critical to a microservices-based system”.

Furthermore, some researchers [29] suggest that microservices are supposed to be easily-replaced components. The same is suggested in [1] where microservices should have the possibility of being isolated from the rest of the system or even getting replaced completely. Such replacement and changes should not create complications for the system [1].

Moreover the term “bounded context” has been mentioned. [33] Explains it as microservices doesn’t need to know anything about how each microservice was developed. A similar idea is also mentioned by other researchers. [30] Mentions that microservices can communicate using their API and the way each microservice is implemented should not have an impact on the communication. Furthermore, they explains that this property gives more freedom to developers to use different tools for different microservice. This same concept is explained in [1] where the write states that such freedom in choosing different tools for different microservices could help developers in choosing the right tool for the right task.

Many researchers also agree that if one service fails, the system should still be able to operate normally [1] [29] [32]. Moreover since microservices support the promise of loosely coupling, at the times of failures, the failed service can be isolated and fixed while the rest of the system is still operating [1] [30].

**Microservices vs monolithic**

This section will give a comparison overview of microservices vs monolithic architecture in literature. The comparison will focus on the following:

* Size
* Scalability
* Loose coupling
* Maintainability

**Size**

Microservices architecture is composed of different services each service is collaborating with the one or more other services. This whole dynamic will come together to form the final system. One of the key characteristics of microservices architecture is the size of each service. Almost every paper suggests that the size of the service should be relatively small [1][5][29][34]. Even some researchers suggested counting lines of code as a measure to decide on the size of the service [29]. While many other researchers suggested that each service should be concerned in handling one task [33]. This task should be derived from the business requirements.

Such small service size helps managers to assign a small team of developers to each service hence making the development faster and more efficient. Moreover, each team can decide to use different tools for different services, depending on which toolset is the best for each service.

On the other hand, monolithic applications are divided into three layers: Backend, logic layer and frontend [34]. Each layer is covering many functionalities of the system. Hence the size of each layer could end up getting bigger and bigger with more business requirements. Furthermore, assigning small team of developers will be harder and each team needs to collaborate with other teams during the development. Moreover, usually, teams don’t have the freedom of choosing their own toolsets for development. They are bounded to what other teams are using and whether their toolset is compatible with the rest or not.

**Scalability**

Applications based on microservices have the chance to grow when there is a need for new features in the application. Adding such new features means adding new services to the system. And adding new services to the system is usually an easy task.

On the other hand, at some point monolithic applications grow to the point where they are hard to scale. The codebase becomes very big and complicated and each additions of new features require a good amount of work to allow the application to accept the new features.

In the book The Art of Scalability [35] the scale cube is introduced. It can be seen in figure 2.1. The Scale Cube has 3 axes: X-axis, Y-axis, and Z-axis [35]

* Horizontal Duplication and Cloning (X-Axis)
* Functional Decomposition and Segmentation (Y-Axis)
* Horizontal Data Partitioning - Shards (Z-Axis)

Commonly, monolithic-based application can scale only on one axis, that’s (X-Axis). On the other hand, a microservice-based application have the ability to scale over all three axes [36].



Figure ‎2‑1 Scale Cube [35]

**Loose coupling**

One of the important characteristics of microservices architecture is that services should be isolated from each other [33]. The connection between services is only achieved via a well-defined interfaces. And each service can be modified as long as it is still respecting its interface. Hence making the application components loosely coupled. On the other hand, monolithic applications developers have to take extra measures to make sure that the parts of their applications don’t overlap which costs them more time and work. The importance of loose coupling is stated clearly in [5] as “Loose coupling is critical to a microservices-based system”.

**Maintainability**

Since microservices are independent entities, then isolating each service in case of a failure is possible while the rest of the system continues operating [1]. However, the same can’t be true for monolithic-based applications and in worst cases one failure in the system could cascade to stop the whole system from operating [1].

In [34] researchers use different points to compare monolithic-based applications to microservices-based application. Table 2.1 presents the comparison. They conclude that both architecture styles have positive and negative points. In general microservice architecture is more suitable for projects with big codebase. But once the project is small building it with microservices architecture could bring an additional overhead [34].

|  |  |  |
| --- | --- | --- |
| Category | Monolith | Microservices |
| Time to market | Fast in the beginning, slower  Later as codebase grows. | Slower in the beginning because  of the technical challenges that microservices have. Faster later |
| Refactoring | Hard to do, as changes can affect multiple places. | Easier and safe because changes are contained inside the microservice. |
| Deployment | The whole monolith has to be deployed always. | Can be deployed in small parts, only one service at a time. |
| Coding language | Hard to change. As codebase is large. Requires big rewriting. | Language and tools can be selected per service. Services are small so changing is easy. |
| Scaling | Scaling means deploying the whole monolith. | Scaling can be done per service. |
| DevOps skills | Doesn’t require much as the number of technologies is limited. | Multiple different technologies a lot of DevOps skills required. |
| Understandability | Hard to understand as complexity is high. A lot of moving parts. | Easy to understand as codebase  is strictly modular and services use SRP. |
| Performance | No communicational overhead.  Technology stack might not support performance. | Communication adds overhead.  Possible performance gains because of technology choices |

Table ‎2.1 Comparing monolith and microservices [34]

### Trust literature review

This part of the thesis tries to present a literature review of trust and the different mechanisms used to help secure the behavior of a software.

First, a definition of trust is illustrated, to give the reader a broader understanding of trust and its use in the literature. The next step is presenting the different techniques of trust as used by researchers and software developers.

**Definition of trust**

The word trust has been a subject of many studies, many researchers tried to form a definition of trust and what it means. The reason for this is because trust plays an important part of people’s life and is involved in many scientific fields such as philosophy, psychology, economy and recently in computer science. In his famous PhD thesis Marsh [9] mentions that many efforts have been spent trying to discuss trust and putting a definition to it, especially in the second half of the last century. His research was an attempt to create a model that can offer a mathematical way to measure trust.

In [6] concludes that there’s no one agreed upon definition of trust, “little consensus has formed on what trust means”. In his research, he agrees with [9] that many discussions have been written about trust. Furthermore, he adds that different definition of trust are used in the literature,” significantly diverse definitions of trust continue to be used in the interdisciplinary research literature”. In an attempts to giving a definition for the word trust, [6] first tries to show the importance of trust. It is mentioned that “trust leads to cooperation”. First, presents what some other researches have provided as a definition or an understanding of trust. On the other hand researchers in[18] try to give a definition or an explanation of how trust can be evaluated. Their idea is that the trust between two parties is a variable with many dependencies. They say: “The trust of an entity with other entity is not a fixed value but can change dynamically depending on the behavior of the entity and context in the environment”

In [6] a distinguishes between six types of trust is made “we define six related types of trust”. While [18] makes a distinction between two types of trust, those are execution trust and code trust. It explains them as “Execution trust exists from subject’s side to service provider’s side that service provider will correctly and faithfully allocate resources for the efficient execution of job with respect to established policies”. While according to [18] code trust “exists from service provider’s side to subject’s side that subject will generate a legitimate request consisting of virus free code and will not produce malicious results and does not temper other results/information/code”

Moreover [18] gives other distinctions for the trust, they explain this as trust could have other types. This extended distinctions of trust types is composed of seven types of trust. Namely “direct trust, indirect trust, full trust, partial trust, recommended trust, authentication trust and finally privacy trust”.

The first type of trust defined in [6] is the Trusting Intention. This type of trust means that one is able to depend on others. [6] Argues that this type of trust is a “situation-specific”. On the contrary to this definition, [10] thinks that this type of trust is not a situation specific. While [6] presents the difference in such understanding as “This makes the other person the object of trust, rather than the person in one situation”

The second type of trust is the “trusting behavior” as defined [6]. The definition for trusting behavior is also given in [11] “is the extent to which one person voluntarily depends on another person in a specific situation with a feeling of relative security, even though negative consequences are possible “. [6] Goes further and tries to decompose trusting behavior into different subconstruct forms. Namely are: “cooperation, information sharing, informal agreements, decreasing controls, accepting influence, granting autonomy, and transacting business “. Researchers in [12] studied the trusting behavior in their work, named “Belief in others’ trustworthiness and trusting behavior”. They show that many factors play a role in the trusting behavior, and it is not just about “maximizing the profit”.

The third type of trust in [6] is “trusting Beliefs”. The given explanation is “the extent to which one believes (and feels confident in believing) that the other person is trustworthy in the situation” other researches have also studied trusting beliefs, for example [13] explain that trusting beliefs are “perceptions of the competence, benevolence, and integrity”. In their explanation they give an example of a vendor-consumer relationship. “Willingness to depend (that is, a decision to make oneself vulnerable to the vendor)”.

Trusting beliefs is also used as one of the conceptual definitions of trust in [7]. Besides trusting beliefs, two more definitions are given: Disposition to Trust and Institution-based Trust.

In [37] a discussion about the importance of perceived information and its consequences. When low-quality information is provided but such information where not of a big importance then the consequences of such false information is relatively low. However, when the nature provided low-quality information has high importance then the negative consequences of trusting this information could be high.

Moving on to the fourth type of trust that was distinguished by [6], this type is called system trust. It is explained as “means the extent to which one believes that proper impersonal structures are in place to enable one to anticipate a successful future endeavor”. Researchers in [14] give an example of system trust, the use ecommerce system as such example. They concluded that system trust has an impact on the intentions of customers to decide whether to buy or not, “system trust plays an important role in the nomological network by directly affecting trust in vendors and indirectly affecting attitudes and intentions to purchase.”

Dispositional trust is the fifth type of trust in [6], explained as “, if one believes that others are generally trustworthy (Belief-in-People), then one will have Trusting Beliefs (which in turn lead to Trusting Intention).” Dispositional trust is also noted in [15], in his research about situational uncertainty, he concludes that dispositional trust can foresee trusting phenomenon.

Lastly, the sixth type of trust according to [6] is the situational decision to trust. Explained as “the extent to which one intends to depend on a non-specific other party in a given situation. “. Although it is recognized as a different type of trust, but it doesn’t exhibit much difference from the first type of trust which is Trusting Intention.

[8] In his paper about the concept of trust defines trust as “Trust as a ‘leap of faith’ or willingness to be vulnerable”. He argues that trust is a tool learnt at as earl age as infancy. Where people use this tool to approach uncertain situations “trust is learned in infancy and enables the individual to deal with the unknowable in the social context”. In his explanation for the term ‘leap of faith’ he presents this term as an important part of the trust where it “involves the trustor experiencing a lack of expertise in a particular area of their life and acknowledging that the expertise they require to address this lack is held by another individual or system. They either consciously “

However, another definition of trust is also presented in [8] trust is seen as a “social capital” writer argues about the role trust plays for individuals in society and hence the role each individual plays in the society.

Lastly [8] also presents trust as a part in the “power-knowledge” theory where knowledge leads to power and trust is an important component to acquire knowledge.

As can be seen that there’s no one definition of trust in the literature and many researchers have come to conclude different means and concepts of trust. Some have given examples from the real world such as [14] where he talks about trusting a system. The same concept of trust is agreed upon by [6] where he gives an example of trust in a system of doing a purchase via the credit card. Where both the buyer and the seller trust the system. In case the system rejects the credit card of the buyer, both parties will not have less trust of the system but the seller will only be suspecting the buyer and most likely never the system.

**Authentication and Authorization**

Authentication and authorization are important for the trust. When a service is able to identify itself to other services, it helps to add points to the overall evaluation of the trust. Authorization will help to have the requestor gains access to the resources such as data. In [18] a definition of authorization is given as “deals with issues like who can access which resources/services under which conditions”. Hence once a microservice is authorized, it will be able to make requests to other microservices and exchange data with them.

An explanation for an authorization system is given in [18] by describing it as a system that provides access rights. They explain “An authorization system can be defined as a system that grants specific type of access to specific requesters based on their authentication”

Furthermore, [19] Describe authentication as “allows identity verification of any entity.” Moreover, the authentication of users as “the basic feature of protecting data from computer system intruders”. Wallace [20] agrees to this definition, he states an authentication protocol as “its purpose is to authenticate entities wishing to communicate securely. “

Importance of authentication is described by [21] as a very important aspect of computer systems security, they mention “authentication in computer systems has been a cornerstone of computer security for decades” such statement shows without doubt the importance of authentication in computer system and applications and agrees with the rest of the researchers about its importance and role. Writer in [20] continues to explain authentication as “Authentication is a simple function where one party presents a set of credentials to a system. If the credentials match a given set on the system, the system returns a value that represents authorization; otherwise it does not.” In the case of microservices, this attempts of authentication happen between microservices. Where one is trying to identify itself to the others. Similar to [19], researcher in [20] continues to present the importance of authentication by stating that it is the very first step a requestor has to take before it is granted further access, “verifying the identity of an entity is the basis for all future rights and privileges granted to the entity”.

Furthermore, [21] tries to give a simple example of what an authentication process could look like. Their example uses the ID of a user for achieving an identity verification. They say: “In the basic authentication process, the entity desiring authentication presents credentials, usually an account ID and some additional information, to prove that the request is coming from a legitimate owner of the ID”. [19] Shows the importance of authentication as it is the proceeding step before authorization “Authentication is critical for sending our data over the internet, as well as for ensuring that authorizing is done properly allowing access to systems and services.” Writers in [21] agree with [19] to the importance of verification where “Identifying a user is essential for the application of security in the form of permissions to various objects, processes and access to resources”

In the book “Information Security: Principles and Practices” [22] enclose the goals of security in three point: “All information security measures try to address at least one of three goals” those points are stated to be “Protect the confidentiality of data, preserve the integrity of data, promote the availability of data for authorized use”. On the other l [19] states similar points as the task of authentication and authorization. They state that “Authentication and authorization is implemented to ensure confidentiality, integrity, availability, authenticity, and accountability.”

An indication to the importance of identity check in computer system is made in [20]. It explains that having identity verification helps in making the system more secure against attacks. “Validating users before allowing them access is an easy way to catch an intruder from stepping into boundaries that they shouldn't cross”.

Moving on, [19] continues to explain how authentication is achieved. Where they explain that “Authenticating users is carried out by the series of identification and verification stages. At the identification stage, access to a security system is defined, and the binding between an entity and an identifier is done during the verification step.”

From the definitions and explanations given by different researchers, it is clear that authentication is an important step in giving access rights to a requestor that’s trying to access one or more resources.

On the other hand, [21] tries to provide a more practical view on authentication by presenting a simple mechanism which uses a combination of a username and a password. They state “The concept of a user id and password is a cost effective and efficient method of maintaining a shared secret between a user and a computer system”. It moves on explaining that many computer systems use the simple known identification method. This method is composed of a username and a password, “Many modern systems have adopted a simple id/password method of achieving the goals associated with the identification and authentication function”. The same thing that [19] agree upon, it saies “the old-fashioned technique which requires a username and password remains the prevailing measure of securing computers, email accounts, or online transactions.” They also explain that despite all the advances that took place in both hardware and software, the combination of username and password is still the most dominant way of verification in computer systems. Saying: “Even with the adoption of new technologies in hardware and software, password authentication is still not completely replaced by the existing alternative authentication methods.”

On the other hand, [18] gives more in depth definition of an authentication and authorization system. They define each entity in the process from the requestor to the requestee including the resources and the action to be taken upon these resources. They describe the requestor as “Subject is an entity that wants to access services/resources. It can be a user, a service or any other entity on behalf of user/service”. And then move on to describe the service by explaining that “Service is a piece of software that provides some functionality and can be accessed by Subjects or other Services”.

After that, [18] have described the requestor and the service they continue by giving a definition to the resource that’s being requested. They say: “Resource is an object that is accessed by Subjects. It can be a CPU, a storage device, software, data”

Another interesting definition given in [18] is about the requirements given by each service in order to be accessed. This is called Service Policy, explained as “Service Policy refers to the set of rules/requirements associated with the Service. A Subject must conform to Service Policy in order to Access that Service”. Furthermore, the access that’s granted to reach the requested service is also explained. Where “Access is an operation that a Subject performs on Service/Resource. The access is provided based on conformance to Service Policy that is associated with that Service/Resource. Hence it can be clearly seen by definitions in [18] that access to the service is not granted unless the service policy of the service is respected. While Policy itself is also described by Singh et al [18] as “a set of rules/requirements”. Where this set of rules can be linked to the Subject, the Service or even the Domain according to [18].

**Reputation based trust**

Reputation based trust started as a review made by users for others, one of the earliest examples of it was adopted by eBay. As [23] refers “Reputation systems are already being used in successful commercial online applications”. A similar idea is referred to in [28], it says: “Reputation-based trust systems were mainly used in electronic markets, as a way of assessing the participants”.

However, trust has been divided into two distinctions one as “strong and crisp” while the other as “soft and social” [24] the “soft and social” is concerned with reputation based trust. Where according to [24] “reputation-based trust relies on a “soft computational” approach “. In this case, trust is computed from two sources: First based on own experience, second based on experiences of others as referred to by [24]. Moreover, trust depends on other factors such as the time and the settings [26].

The same concept for computing trust is used in [25], it agrees that reputation-based trust is computed from two sources. It says: “Reputation information is divided into two categories: First-hand experiences are gained from the trustor monitoring the outcomes of actions it has engaged in itself, and are generally considered to be error-free within the limits of observed- ability. External experiences are gained from third-party recommenders based on their own first-hand experiences”.

On the other hand, in [26] uses the term “behavioral trust” instead of “reputation trust”. It is defined as “fulfilling the expectation of others”. And it is classified into two categories: Direct trust and indirect trust. Where direct trust means the experiences gain from own direct interaction. While indirect trust means the experiences of other’s interactions. It is obvious that [26] agrees with [25] where it uses first-hand experiences and external-experiences instead of direct trust and indirect trust. In both definition, the resulted trust is variable and never constant, where its value changes after each interaction. While in policy-based trust the resulting decision is either positive or negative [24]. Where such trust depends on well-defined measures such as certificates and is referred to as “strong security” [24].

[27] Also agrees with the mentioned studies, it states that “reputation serves as the basis for trust”. Hence giving an important value for the experiences of other entities in the system.

A distinction between entity trust and content trust is given in [17]. Entity trust is given as an evaluation of an entity based on its ID and behavior. While content trust is defined as “A trust judgment on a particular piece of information in a given context “. Both types of trust are related to each other’s.

## Analysis

This section provides an analysis of the content trust as well as microservices architecture in regards to the requirements that were presented in the first section of this chapter. Those requirements will be analyzed against the presented literature review in the second section of this chapter. The analysis will be discussed under two titles:

* Microservices
* Content trust

Besides going through the previously presented literature review, the following pages will also compare the presented requirements against some well-known implementations of microservices as well as content trust.

### Microservices analysis

The following pages provide analysis for the literature review of microservies while comparing it to the requirements of microservices, mentioned in the first section of this chapter.

Microservice-based applications can be built using either one of two architecture styles, that’s is orchestration and choreography [41]. In orchestration, the work flow between services is managed centrally. One or more service is directing the calls to their intended destination. Hence, the application has a central part to manage the traffic and help services communicate with each other. On the other hand, services in an application based on the, choreography style should handle any calls by themselves. They should identify the destination service whether its address or the type of service it offers. Unlike orchestration, choreography offers more loosely-coupled architecture since no centralization [41]. As a result, an application could benefit the most of microservice architecture.

According to the requirments, services are supposed to be small. When one tries to have a very abstract understanding of the name microservices. The word micro reflects the size of the service. The size of the service was discussed heavily in the literature [1] [3] [5]. Some writers tried to come up with a way to measure how small each service should be. Others were pointing out that the decomposition of tasks should be according business requirements and each service should be handling only one task [5] [33].

Each microservice is supposed to be independent, can be deployed independently and can be changed without affecting other services. Such requirement was also discussed in the literature. In the definition given in [1] it is clearly stated that microservices should be autonomous. Writes in [31] [33] also emphasis that in order to get the most out of microservices architecture, services should be made as independent as possible. However, having services as independent entities, doesn’t mean that service can’t interact together. On the contrary as mentioned in the definition in [1] services are supposed to collaborate together. In order to keep services independent but at the same time having the chance to work together when needed, each microservice should provide an interface. Writers in [1] [5] [29] [33] al agree that microservices should have interfaces to help other services interact with them. While at the same time giving each service a great deal of independence.

The previous point bring the discussion to the argument that microservices-based application should be resilient. Where the system have the ability to handle failures. In such a way that the rest of the system will keep working and only the failed parts will stop. Moreover, the failed parts can be isolated easily to be fixed or replaced. This was also mentioned in [1] [31]. Moreover, in [33] it is stated that each microservice should be encapsulated. Furthermore, microservices don’t need to know anything about how each service is implemented. All what microservices need to know is how to communicate with the target microservice. This gives developers more freedom in how to implement each service, or even how to update or modify the implementation of an existing service. As long as the interface of the service is respected, other services will still be able to communicate with it. Hence the system will continue to function despite the updates and modifications.

Since each microservice has a well-defined interface [1][5], then replacing one service with another is easily achieved as long as the interface for the new service is the same as the old one. Moreover, reusing microservice in other application is easier. One of the requirements of microservices is having the services as reusable entities. In [1] and [29] writers point out that microservices should be designed in order to be used by more than one application. Since each service is handling one task, then using such service in other application is also possible. If the new application can adhere to the service interface then no changes are needed, and the service can be used easily.

Microservices architecture was developed to overcome the shortcoming of monolithic architecture. One of these shortcoming appears more clearly the more the project gets bigger. At some point adding more features to a monolithic-based application will be a difficult task [34]. At some point an application could require huge modifications in order to accommodate new features. On the other hand, microservices-based applications are supposed to be scalable [5] [30]. Such applications are composed of many small interacting parts, each part has a well-defined interface. Hence the system will be able to accommodate new services with ease.

This leads the discussion on how easily developers can release their solutions. And how possible it is to offer continuous deployment and updates. Unlike monolithic-based applications, microservices-based applications have more flexibility after adding new features to the system. Developers can offer continuous delivery in a way that they can easily keep adding new services to the system without the need to release new versions with each newly added feature. In [1] as well as in [34], easy deployment of microservices-based applications was discussed. In [34] a comparison was made between microservices and monolithic architecture where deployment was one of the compared points.

### Content trust analysis

Different definitions of trust were presented by different researchers. Where some of the definitions intersect with one or more of the provided requirements of content trust. One definition for trust distinguish between direct trust and indirect trust. Direct trust is established after own direct interaction with other entities. While indirect trust information is gathered from other entities experiences with the entity in concern. This definition can be seen in at least one of the requirements of content trust. Microservices will have their own evaluation of trust once they interact with a certain microservice. And also will be depending on the evaluation of other microservices for the concerned microservice.

After each interaction, the trust they already have about the other microservice could be affected positively or negatively. Moreover their new evaluation of trust could also play a role in how other microservices evaluate their trust with the concerned microservice. This is perceived from the indirect interaction, hence the indirect trust.

One of the widely cited study about trust [6] mentioned “trust leads to cooperation” this understanding is also exhibited in the requirements of content trust. The point of adopting content trust in microservices is to make collaborating microservices trust each other’s behavior and exchange data safely. In such scenario, having high evolution of content trust among microservices will lead to more exchanged data and cooperation.

Trusting the behavior of others was also mentioned by some researchers [6][11]. One definition is presented as “Belief in others’ trustworthiness” this definition is reflected in the evaluation of trust each microservice will have. Believe is reflected as a dynamic value that can grow or shrink depending on the how positive or negative each interaction is.

The relationship among services is present in more than one requirement. On the other hand, the relationships among collaborating entities was mentioned by different researchers. Such collaboration between concerned entities is mentioned in the sixth definitions of trust presented in [6] aswell as in [15].

The age of microservices is taken into account when deciding about the content trust of a microservice. It is mentioned as one of the requirements of the content trust system. However, the age doesn’t have much influence on any description provided for the trust or any of its contrasts or sub definitions. Yet the age is mentioned specifically in [17]. It is stated that the age of the content could play a role in helping the readers of the content on deciding whether the content is trustworthy or not.

The sensitivity of the service is also presented in the requirements of the content trust. In [37] the importance of the provided information is discussed. In the requirements of content trust, the sensitivity of the service can be projected into the importance of information presented in [37]. In such case when there’s a trust among microservices that are exchanging sensitive information, but the information where not of high degree of integrity. Then as indicated in [37] the consequences could be more serious, than if the exchanged information where of low importance.

The identity of a microservice has a weight in deciding of trusting the behavior of a microservice or not. In the requirements of microservices, each microservice should be able to authenticate itself to other microservices. Failing to authenticate itself, could result in having a decreased evolution of the service by one or more other services. In [21] a model for verifying the identity of a requestor is presented as a combination of User ID and a Password. Such combination is also used by other researchers [19]. In the case of microservices, proving each service what it claims to be is important as it establish a first level of trust. Such ground could be used to move on and try to establish other forms of trust. Having the identity of the service verified will help in increasing the evaluation of it by one or more services positively.

Regarding the implementation of content trust. The most common one is used by Docker and it is called Docker content trust. Docker is basically a container for processes. One can think of it as a virtual machine but much lightweight and faster to boot. This lightweight virtual machine is called a container and one host can have more than one container running at the same time and sharing the host resources. Docker containers are actually used widely to deploy services for applications built on the microservice architecture. Docker content trust is used to help in trusting the images of the containers released by software providers. According to the official documentation [38], the point of Docker content trust is to ensure the integrity of Docker images and also verify the identity of the publishers of the image. This explanation only satisfies a portion of the requirements of content trust presented in the first section of this chapter. But it doesn’t go any further, for example: users of Docker images can’t provide an evaluation of their experience after using a specific image. Hence other users can’t use such information in helping them to decide of whether it is reasonable to put one’s trust a specific image or not.

To conclude this chapter, an application based on microservices could be described as a big piece made of many small blocks, each block is an independent reusable entity. It can be reused to develop other applications. While content trust will add a security layer on top of the tradition security measures. A system of content trust could be helpful once the blocks of the application is developed by different sources. Hence it’ll be a mechanism that helps each service to trust the behavior of other services. This process is done automatically, without a human intervention.

Next chapter in this thesis will discuss the concept behind building the Blog. It’ll give an overview of how, micro frontends, microservices, and content trust system can all work together to produce a more secure, flexible and robust system.

# Concept

Web applications usually consists of two parts, a frontend and a backend. The frontend is the part that’s responsible about what the user can see and interact with. The backend is responsible for processing the data and storing it in a database.

Moreover, those two parts of the application are divided into three layers, that’s the user interface, application logic and the database. This monolithic three layers architecture has some problems associated with it. Where it is hard to scale the application when new functionalities and features are required. The bigger the application the harder to scale []. Maintenance also becomes harder, the three mentioned layers overlap and becomes more tightly coupled. When a problem arises in the system, sometimes it may not be easy to isolate the problem and keep the system running. The application is tightly coupled and isolating any parts of it could lead to the stopping of many of its functionalities.

Furthermore, continuous development under monolithic architecture is hard [], developers has to either releases new version with every bug fixed or with every new feature added to the system. Hence new versions of the application will be released in short periods. Or developers have to wait until the changes that’s made to the application deserve a new release. Thus some problems in the application could exist of a good amount of time before a new version is released that helps in fixing such problems.

Microservices architecture was developed to overcome the problems that’s imposed by monolithic architecture. Basically, under microservices architecture, the logic layer is divided into small applications, those applications work together to deliver the final product. Each application is basically a service, and each service handles one task. Hence the logic layer could consists of many services. Since each service handles only one task then it is advised that the size of each service is kept as small as possible []. Each service is called a microservice.

Microservices architecture can also be applied to the frontend part of the application. That’s the user interface. In such scenario, the user will be getting the same output as if the application is based on the monolithic architecture. The difference is that the user interface will be composed of more than one frontend. Those frontends collaborate together to render the final output to the user. Each frontend is called a micro frontend.

Microservices architecture helps developers create loosely-coupled applications. Although each small service is collaborating with other services but the collaboration happens under a well-defined interfaces. Each service offers an interface that helps other services make a request to it and receive a response. So designing an application that’s loosely coupled is easier for developers and becomes more like a default behavior.

Such loose coupling helps developers isolate any problems appear in the application after delivery. This isolation will only affect the service that host the problems and any other services that want to interact with this service. In other cases the isolation of the problem could not affect the system at all. Developers could create more than one service to handle each task. When a service is isolated the other services that perform the same task can handle the coming requests. In this case the system will continue to function as usual while the problem is being fixed.

Moreover, continuous deliver is much smoother under microservices architecture, developers don’t need to release a new version for each change they make. They can simply replace one service by a new one and the system will continue to run as usual.

## Concept of microservices

This thesis will implement a Blog built using microservices architecture. The Blog will use many services collaborating together to deliver its services to clients. The services in the Blog will have certain features that make them suitable to be used in a microservices-based application. Such services adheres to the requirements mentioned in the second chapter. The services will be:

**Small**

Each service in the Blog should have a small size where it handles one task. For example, one service would handle storing a new post in the database. Another service handles bringing posts from the database.

The reason for making services small is to be able to get the most benefit possible from using microservices architecture. When services are as small as possible it becomes easy to replace them with new services, or isolate a service when it is not running as it is supposed to. Also it helps to make the system more scalable since adding new features means adding more small services to the Blog. This would be easier than adding a big service that handles many tasks and have to communicate with many other services.

**Independent**

Each service in the Blog is independent where it doesn’t rely on other services to perform its task. Services of course would need an input to start processing the data, but handling the data is something a service doesn’t need help with. The more independent each service is, the easier it is to form a loosely-coupled application.

When a service only needs the required input to operate then such service can easily be modified or updated without affecting other parts of the Blog. The only concern here is to keep the interface as it is so other services can still deliver data to it and receive the output.

**Has an interface**

Services in the Blog should be independent that performs one task only. But this doesn’t mean that services will act as isolated islands where no communication is happening among them. In fact, without such communication the overall functionality of the system can’t be achieved. Hence most services in the Blog will offer an interface where other services can communicate with them. It is important that all services that need to exchange data with other services be able to do it through a unified well-defined interfaces. Changes that happen to a services should not affect the interface that the service exhibit to the outer world.

**Reusable**

Since each service is performing a small specific one task then there’s a high chance that the same functionality will be needed in other applications. For example, a service that’s responsible for registering new users in the Blog, could be reused in other applications where users registration is required. Such concerns will be taken into account when designing each service. Because when most services are designed from the beginning as reusable entities, it would be easier than taking each service and adapting it to other applications.

The five previous features represent the highlights or how services are going to be. Some differences might occur from one service to another but it will be discussed in the following chapter which is the implementation.

The Blog itself should also have few features that comes from using microservices architecture. After all, if those features don’t exist then the benefits of using such architecture are not reached. On the contrary, microservices architecture brings its own challenges. Hence using such architecture without getting the most of it is just an added overhead in the development. Thus the Blog must be:

**Scalable**

Where new features can be added easily. This scalability comes from the possibility of being able to add new services to the Blog. When there’s a need for new features, the existing services can’t be changed to accommodate the new features. Because it means that one or more of the existing services will be handling more than one task which in turns break one of the main characteristics of a microservice. That’s each service should be small and handles one task only. For this reason, services in the Blog should have the flexibility to accommodate new services and be able to communicate with them.

**Resilience**

The Blog must be able to handle failures where they don’t cascade in a way that affects other services and stop the Blog from operating. The Blog must be flexible in a way that allow for failures isolation where the malfunction services are isolated from the rest of the Blog. Temporary replaced by other services until the failure is handled.

This is a very important feature of any microservices-based application. Such application should exhibit a better behavior when dealing with failures compared to a monolithic application.

**Loosely-coupled**

The Blog should have its services as independent services that can operate without the need of other services. Each service in the Blog is a small application by itself. Some services might need an input or have an output but it is all performed via the interfaces of the services.

Such loosely-coupled structure of the application helps the Blog to be more flexiable when facing problems. Or when some services should be replaced by others. It also helps when performing updates on the Blog.

## Concept of content trust

Content trust as defined in [17] is not an isolated judgment but it is related to the context in which the judgment is taking place. Hence the surrounding environment and the time of making the decision play a role in the final judgment.

Content trust and reputation trust are related but they are not the same. From the requirements provided in chapter 2, it can be seen that the reputation of the involved entities will play a role about the trust of each one of them. It is, however, not the only deciding factor. Many other factors influence the decision of trust. For example: Verifying the identity of each entity has also a negative or positive influence depending on the outcome of the verifying process. Such influence means that identity verification is also related to content trust.

In a microservices environment where many services are trying to work together, content trust will play a role in helping each service to make a judgment of trust the other service or not. Thus making the system more secure. On the other hand, such system of content trust must be designed with care, otherwise the system could behave in an unpredicted way. When such system is not given a thoughtful design and enough preparation and testing then sometimes services could end up making negative judgments about each other. Such negative judgment could happen when a positive judgment is the most probable decision to be made. In this case, services will reject the incoming requests and operations will not take place. Thus clients of the application will be denied the services for not valid reasons.

For example in an online banking system, a user is trying to start a transaction from one account to another. The request goes from the frontend to the services responsible for handling such transactions. Before going any further, the involved services will try to evaluate the trust each one has on the other. If one of the services decides that it can’t trust at least one of the other services then the transaction may not take place. The system eventually will refuse to complete the transaction leaving the client with unhandled request. Hence clients could end up leaving such system and never using its services because of its unpredictable behavior.

The previous scenario, raises many design questions, one of them is whether a system should have more than one services providing the same service. In the previous example, if one service can’t trust another one, then the transaction can still takes place if another microservice was available providing the same service as the untrusted one. Such duplication of services could be useful where each service has more than one option. On the other hand, such design can be redundant. It’ll take more time to design a system that has more than one service handling the same task. Hence the deployment of the application will be postponed and further loses could takes place.

The proposed system will help microservices have an evaluation system of trust that can help microservices decide of whether exchanging data with another service is something secure or not. By secure, it is not meant the medium of communication but how the involved entities will handle the received data.

Building on the requirements mentioned in the second chapter, the system of content trust should follow the mentioned requirements. The following pages will discuss the concept behind building a content trust system based on those requirements. The following points will be discussed:

* Identity verification
* Services classifications
* Service sensitivity measure
* Past experience
* Number of interacting services
* Check the evaluation of other microservices (indirect trust)
* Check the age of microservices
* Passing of trust
* Developers of the service
* Limited services

The proposed system should allow each microservice to be able to check the above characteristics of other microservices.

**Identity verification**

Each microservice has a unique ID in the system. Hence the installed services should be able to read all the IDs of the other services. Once two services want to interact with each other. An Identification process will take place. It will help the two involved microservices to check the ID of the other microservice and make sure that it is part of the installed services. All the involved services will have the ability to express their ID in a similar way. If one of the services fails in doing so then this service will raise more concern. Should the whole communication stops or should it only affect the trust negatively??

Identity check can take several forms and follows different algorithms. The important point here is that microservices should be able to have a unified way of announcing themselves to any microservices that are making a request about the identity of a certain microservice.

**Services classifications**

The involved microservices will offer different types of services. Some could offer routing services that helps users navigate through the website, while other services handle user bank details, for example. The two mentioned services have different nature and different sensitivity level. The system of content trust should be able to identify that services have different sensitivity levels and also classify each service according to its sensitivity.

This classification of sensitivity is not achieved by services themselves. It is one of the characteristics of each service. Developers should be able to give each service this classification.

**Service sensitivity measure**

The classification mentioned in the previous point helps microservices making a better decision. Each service should be able to know how sensitive the other service is. If the service has a low sensitivity classification then its trust evaluation doesn’t need to be high. On the other hand, when a service is classified as a sensitive one, then its trust evaluation by other microservices should be high before deciding to exchange data with it.

**Past experience**

Each microservice interacting with another microservice, could have past experience(s) with one another. The past reactions will have an influence on the decision of whether the two services will interact together this time or not. This means that the system of content trust will record the past overall evaluation of trust of each service. Once a service wants to interact with another service it could look into the past record of its interaction to see if it had an interaction with the microservice in mind and what kind of an overall evaluation it had.

Moreover, past experiences can be designed in more than one way. For instance: one way to approach this point is to keep record of only the last interaction happened between the two involved microservices. Another approach would be to keep record of all the interactions that happened and take the average of all the interactions.

This past experience doesn’t have an influence when two microservices wants to exchange data for the first time. In this case there’ll be no weighting taken into account for past experiences.

**Number of interacting services**

Each microservice trying to initiate a connection with another microservice should be able to know how many other services successfully interacted with it and how many other services failed to interact with it. The difference in the two numbers helps each services to either add a positive value to the trust evaluation or a negative one.

The number of interacting services is variable. It’ll change with any interaction attempt. If the interaction was successful then the number of interacting services as well as the number of successful interaction will increase. Otherwise, only the number of interacting services will increase.

Of course, the more successful interactions each microservice has, the higher its trust value is. This number should be taken into consideration in relation to the overall number of interaction each microservice has.

**Check the evaluation of other microservices (indirect trust)**

Indirect trust plays an important role in content trust. The reputation of an entity is related to the content trust [17]. Each microservice should be able to see the evaluation of other services to the microservice in concern. When two services are trying to connect with each other, both of them should be able to read the trust evaulation of other services about the service on the other side of the connection. This evaluation whether it has a positive sum or a negative sum will have an influence on the decision of whether to trust the service or not.

**Check the age of microservices**

Some services will be added together to the system. While other services will be added at a later time. Moreover, other services could be replaced by new ones. This dynamic means that services will have a different operation timespan. This timespan can play a role in the system of content trust. The older the microservice is the more positive value it should get regarding this factor. Hence microservices should be able to check the age of other microsrvices where such age could influence their decision of trusting the service or not.

**Passing of trust**

Transitivity of trust could also play a role when interacting with a new microservice. When two microservices want to interact with each other for the first time, any indirect trust between the two could be helpful. Each microservice should be able to check the services that it already trusts. If one or more of those services also trust the new service in concern, then such trust could be transformed to the original service and influence its decision positivly of whether to interact with the microservice or not.

Transitivity of trust doesn’t imply an automatic trust. If three services named A, B and C have the following trust relationship: A trust B with high evaluation. B trusts C with high evaluation also. And then A and C are trying to cnnect with each other, they have one highly trusted service in common. Thus both will end up increasing positivly their trust about each other. But this doesn’t mean that they’ll actually trust each other and start exchanging data. This simply means the positive evaluation will increase.

**Developers of the service**

Who is behind the development of the service should give a good indication of whether the service should be trusted or not. Developers of the system could develop most of the services themselves. Yet they might need to use third-party services. In this case, developers can give a trust value for each service based on the source of its development. In-house developed services should have a high estimation. Services developed by well-known companies should also have a high estimation. On the other hand, services that were developed by unknown developers should have an estimation that helps services be more aware of the situation. Hence a low estimation is expected.

**Limited services**

A system composed of many services interacting together could at some point have two services providing the same service. This is also the decision of the designers of the system, of whether they want to have only one service handling each task. Or more than one service handling the same task. The later decision could be due to maintenance. In case one of the services fail the other one could handle the coming requests immediately.

This point is important for content trust. If the system has more than one service handling the same task, then services have more than one option when trying to make a request for a certain service. Hence the evaluation of a service that has other services doing the same job should be different from the first case. When only one service exist for any task.

Now the main points of the system of content trust has been laid out. Developers of an application will have to decide themselves of how to use such system. The mentioned points above can all have the same evaluation level. Meaning that all parts of the system will have the same weight when deciding on trusting a service or not. For example: Highly evaluated indirect trust, would have the same effect as highly evaluated past experience.

On the other hand, a different team of developers developing another application, could think differently. The way they would use the content trust system is similar but with different weighting. For example: A highly evaluated indirect trust doesn’t have the same effect as a highly evaluated past experience. Developers could think that for this particular application that’s being developed, past experience should have more effect than the evaluation of indirect trust. And this will be applied to the rest of the points in the system. Each point could have a different weight from the other. This will cause different results if two systems used the exact system of trust but had different weighting systems. Hence the trust relationships between involved entities will be different.

## Overall structure

Application (3)

Figure ‎3‑1 Overall structure

Figure 3.1 shows a possible structure of the proposed system. The system has the following parts:

* A frontend which consists of micro frontends
* Services
* One or more databases
* Communication system between services
* Content trust system

The Blog will have a frontend that helps users interact with it. The frontend will consist of more than one part. Each part is called a micro frontend. Each micro frontend is a small independent application that can be deployed independently and even reused in other applications. Each micro frontend can be developed using different technologies and frameworks.

The user that’s interacting with the frontend will not be able to notice any difference from interacting with a monolithic application. The frontend will appear to the user as if it is a one big frontend. Hence, it’ll be very hard for the user to tell where a micro frontend starts and ends.

Each service is represented by a circle, as figure 3.1 shows some services have a direct contact with a database. While other services don’t have a contact with a database. This simply explains that the system will have more than one database. The reason for this comes from the definition of a microservice that’s mentioned in the second chapter. Each microservice is an independent unit that can be deployed independently. As a result, some services will have their own small database. The service as well as the database can be reused in other applications.

It’s also worth noting that some services will not have their own database. Where for example the task of such services would be to validate some values or make a calculation. Such values will be provided by other services and the result will be provided back.

A communication system among services is proposed to make sure that the application operate as it is supposed to. Such communication system is represented by arrows in figure 3.1. Each micro service in the Blog will have an interface. This interface helps the microservice to interact with other microservices. Therefore, services will be communicating with each other to handle user’s requests.

A content trust system is also proposed. This system will help microservices validate the behavior of each other. In other words, the content trust system will help services make sure that they can trust the behavior of each service they have to interact with. A content trust system is represented in figure 3.1 by a blue rectangle between the arrows connecting the services.

# Bibliography

[1] H. Gebhardt, “Dezentrale Autorisierung in,” 2010.

#### Bezeichner für Anhang A

##### Bezeichner für Anhang A.1

<xml>

<element id=”guid”>example</element>

</xml>

Listing ‎A.1 Mit Alt+ F9 bearbeiten (hängt von Heading 4 statt von 1 ab)

# Glossary

Glossarbegriff

Im Glossar können ausgewählte Begriffe genauer definiert werden…

HTML

Bei HTML (Hypertext Markup Language) handelt es sich um eine Auszeichnungssprache …

# Index

**No index entries found.**

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Chemnitz, den 9. January 2019

<TODO: Vorname Autor> mpc

TODO: Es wird empfohlen die offizielle Selbständigkeitserklärung des ZPAs zu verwenden: [http://www.tu-chemnitz.de/verwaltung/studentenamt/zpa/formulare/ Allgemein/allgemein/selbststaendigkeitserklaerung.pdf](http://www.tu-chemnitz.de/verwaltung/studentenamt/zpa/formulare/%20Allgemein/allgemein/selbststaendigkeitserklaerung.pdf)

Für weitere Hinweise siehe Abschnitt **Error! Reference source not found.** „**Error! Reference source not found.**“