Department of Computer Science

Distributed and Self-organizing Systems Group

<TODO: Art der Arbeit>

<TODO: Titel der Arbeit>

[Comments] mpc

Chemnitz, 22 February 2019

**Examiner:** <TODO: Prüfer>

**Supervisor:** <TODO: Betreuer>

**mpc,** [Comments]

<TODO: Titel der Arbeit>

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

Chemnitz University of Technology, 22 February 2019

Abstract

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 case

Table of Contents

[List of Figures vii](#_Toc536558538)

[List of Tables ix](#_Toc536558539)

[List of Listings xi](#_Toc536558540)

[List of Abbreviations xiii](#_Toc536558541)

[1 Introduction 1](#_Toc536558542)

[1.1 Current Situation 2](#_Toc536558543)

[1.2 Motivation 3](#_Toc536558544)

[1.3 Problem 5](#_Toc536558545)

[1.4 Objective 7](#_Toc536558546)

[1.5 Outline 8](#_Toc536558547)

[2 State of The Art 11](#_Toc536558548)

[2.1 Requirements 11](#_Toc536558549)

[2.1.1 Requirements of Microservices and Micro frontends 12](#_Toc536558550)

[2.1.2 Requirements of Content trust 16](#_Toc536558551)

[2.1.3 Requirements of Developers and Users 19](#_Toc536558552)

[2.2 Literature Review 20](#_Toc536558553)

[2.2.1 Microservices and Micro frontends literature review 20](#_Toc536558554)

[2.2.2 Microservices vs monolithic 23](#_Toc536558555)

[2.2.3 Trust literature review 28](#_Toc536558556)

[2.3 Analysis 35](#_Toc536558557)

[2.3.1 Microservices analysis 36](#_Toc536558558)

[2.3.2 Content trust analysis 38](#_Toc536558559)

[3 Concept 41](#_Toc536558560)

[3.1 Concept of micro frontends 42](#_Toc536558561)

[3.2 Concept of microservices 43](#_Toc536558562)

[3.3 Concept of content trust 47](#_Toc536558563)

[3.3.1 Properties of the Content trust 48](#_Toc536558564)

[3.3.2 Context of Content trust 52](#_Toc536558565)

[3.4 Overall structure 53](#_Toc536558566)

[4 Implementation 57](#_Toc536558567)

[4.1 Implementation of microservices 57](#_Toc536558568)

[4.1.1 Microservices details 62](#_Toc536558569)

[4.2 Content trust implementation 67](#_Toc536558570)

[Bibliography 72](#_Toc536558571)

[Bezeichner für Anhang A 73](#_Toc536558572)

[Bezeichner für Anhang A.1 73](#_Toc536558573)

[Glossary LXXV](#_Toc536558574)

[Index LXXVII](#_Toc536558575)

# List of Figures

[Figure ‎2‑1 Scale Cube [35] 25](#_Toc536467436)

[Figure ‎3‑1 Overall structure 53](#_Toc536467437)

[Figure ‎4‑1: Overall interaction 58](#_Toc536467438)

# List of Tables

[Table ‎2.1: Comparing monolithic and microservices [34] 27](#_Toc536467687)

# List of Listings

[Listing ‎4.1: JSON format 59](#_Toc536467591)

[Listing ‎4.2 ContactUs API 61](#_Toc536467592)

[Listing ‎4.3: Structure of contact document 62](#_Toc536467593)

[Listing ‎4.4: Contact data example 63](#_Toc536467594)

[Listing ‎4.5: Installing jsonwebtoken using npm 64](#_Toc536467595)

[Listing ‎4.6: Generating a signed token 65](#_Toc536467596)

[Listing ‎4.7: Generated JWT [52] 65](#_Toc536467597)

[Listing ‎0.1 Mit Alt+ F9 bearbeiten (hängt von Heading 4 statt von 1 ab) 71](#_Toc536467598)

# List of Abbreviations

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

# Introduction

Traditional web applications are a software that comprises several parts and all these parts come together to form the final product. However, at the end this product will look as if it is made of one large 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, which is sometimes called three-layered architecture contains presentation, business and data as its three layers.

The word “monolithic” is used in some of the literature to describe a three-layered architecture [2] [3] [34]. In this thesis, both words, monolithic and three-layered will be used interchangeably to refer to a traditional web application that’s composed, mainly, of User interface, Logic layer, and a data access layer.

Once the application is ready to be deployed, developers will have to approach it as if it is, essentially, composed of two parts that cannot be divided into smaller units. Such division can be very useful when there is 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 the 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 in the micro frontends architecture. In essence, the frontend will be 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 that need to be asked when it comes to the microservices architecture. As an example, when considering the functional requirements of the application, developers need to think about how their application can be divided into smaller parts. It should also be considered whether following certain standards can be helpful in making the final product better. There are also other questions regarding the nature of the microservice architecture itself, such as the size of each microservice. Moreover, the communications between microservices are another challenge that deserves thoughtful consideration.

On the other hand, there are also concerns related to the frontend of the application. In principle, the frontend is what the user sees and interacts with. It could be an issue that the frontend may also become complex and will subsequently need to be divided into micro frontends. The concept of microservices can further be projected into the other side of the web application. Microservices concept is not just concerned with the backend of the system - 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. It is essential that 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. In this case, trust does not just mean believing each microservice to be what it claims to be. It requires having a system that helps microservices to trust the behaviour of each other.

There is still no standard definition of microservice architecture and there is no clear guidelines for 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 and, each component should be 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 is a central body that manages the interactions between the services. While in other cases, there is no central 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. With micro frontends, the frontend of the application is composed of many different small self-contained applications. That is, instead of having the frontend as 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. And then each micro frontend can be written in a different framework depending on its needs.

## Motivation

The current architecture that is 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 [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, the application is still very tightly coupled [16]. There are many dependencies 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 cause 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 – either not being able to achieve the wanted updates and fixes as much as they’re supposed to, or 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 a lot of efforts. Microservices architecture is developed to make applications more flexible. With microservice-based application, the system is now more accepting of changes. Developers do not 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.

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

Such trust is very important, especially, when there is a need to use 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 a scenario microservices should not start exchanging this type of data without verifying and knowing more details about the microservices on the receiving end. This is where an implementation of content trust could mean a more secure system. It can ensure that exchanging data between microservices only happens after each microservice trusts the other one. Once it is known that no microservices have any harmful code, or bad intentions then data exchange should take place smoothly between microservices.

## Problem

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

One possibility is that the Web application already exists based on the three-layered architecture, but there is a need to migrate it into microservices architecture. This thesis does not try to find answers or better solutions for migrating from a traditional web application into a microservice-based application.

Another case could be that developers want to develop the required system from the beginning based on the microservices architecture. One reason for this could be that the application is expected to grow. With microservices architecture, it is easier to scale the system as much as needed compared to three-layered architecture [34]. Another reason could be that the application has a complex nature and requires different technology stacks for its various parts. Such complex applications are not very common but they are mostly developed to provide solutions to big firms such as Amazon or Netflix [2]. Hence building such solutions using microservices architecture will help developers to use different tools and technologies for different parts as needed. 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 as well.

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.

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

In essence, micro frontends are autonomous independents parts [1]. These parts are not divided according to how they are going to render on the screen. They are 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 displaying the products, another for the product details and a third micro frontend for adding items to the cart. There could also be a fourth for the check out and payment process. Other micro frontend can be focused on other functionalities of the web site such as handling user data. An example would be, registration and the creation of new accounts. This task can be assigned to a micro frontend. Another micro frontend could also be created to handle security concerns such as when a user forgets his password.

A web application based on microservices could be composed of many microservices. At some point, these microservices may need to exchange sensitive information such as logins or bank details. Hence an implementation for content trust between services can help each service form an evaluation of trust before exchanging data with other microservices.

This situation would be more pressing if microservices were not all developed inside the same company. Such a scenario could arise when small companies want to build their applications using microservices architecture. In this case, when having a small team of available developers, one might consider using ready-made solutions. Developers could use third-party microservices to save time and money. However, doing this could expose the developed solution to more risks. Hence the need to establish a trust mechanism to help microservices evaluate how much each service trust the service on the other end, and if this trust is considered enough for each service to exchange data. Moreover, when adding new microservices to an already running system, both the existing microservices as well as the newly added ones need to be able to have a way to assess how much they trust a service before deciding to exchange data or not. The kind of trust discussed here helps each microservice to form an opinion about microservices that are on the other side of the communication line before making the decision of whether to exchange data or not. 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 the need for a human intervention especially if the application becomes bigger and embraces hundreds of microservices. 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.

## Objective

The focus of this thesis will be on building a system out of microservices and micro frontends while providing a solution for content-trust among microservices.

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

While there are many questions and uncertainties 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 the building of a Blog based on the microservices architecture and the development will involve using the latest technologies and tools to build the Blog. Solutions for challenges faced will be documented.

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 so are the ways of developing web applications. Moreover, it will not try to present a technology comparison of the possible ways to develop a solution based on microservices.

This thesis will also address 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’s identity to the other microservices, but it is also about having a mean or way of evaluating the trust between any two involved microservices.

## Outline

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. Furthermore, a literature review and analysis of the requirements against the literature review will be provided.

The third chapter will focus on the concept of building microservices and establishing a content trust mechanism. 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 will investigate how a system with many moving parts can work and offer a stable and seamless experience for its users. While at the same time, have a very clear division and separation of functionalities into small autonomous collaborating tasks. 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 made during the development will be discussed to help make other researchers 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. The way in which this method is developed will also be outlined and discussed.

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

The last chapter is the conclusion and in this chapter a conclusion and a summary of the thesis will be presented.

# 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 will also discuss the requirements of content trust between microservices. Hence it will 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 will analyse the literature review against the requirements.

## Requirements

This part of the thesis will present requirements analysis for a microservices-based web application that uses a content trust mechanism between its microservices. 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

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.

### 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 keeping any modifications for the other small apps as minimum as possible. 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, the system will continue to function.

Since micro frontends are basically microservices architecture applied to the frontend part of the application [50]. 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

* Replicable
* Respect Interface
* Reusable

Furthermore, respecting the five mentioned requirements results in the following features as described by [1, 5, 29]:

* Resilience
* Scalable
* Easy to deploy

The following pages will go in details about each one of the requirements. For each one, 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 functional requirements of the business. If each microservice handles only one task, developers can increase the chances of developing an application that respects other requirements of microservices, such as autonomous and reusable.

Each microservice is supposed to be small, the size of each service should be scaled down until it cannot be reduced 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. Moreover, the more services are divided, the more microservices the system will have. As a result, having many small dynamic parts in the system will make it harder to manage and could add extra complexity [1].

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 be capable of being deployed independently. This gives applications more flexibility as it helps to make the application more loosely coupled. It is now easier to isolate each microservice and the whole system is more scalable [30].

Furthermore, having independent microservices allow developers to develop each microservice autonomously. Each team has the freedom to choose which toolsets to use for the development of each particular microservice. Such freedom allows developers to choose the most suitable tools based on the requirements of each task.

This also applies to the micro frontends. Each micro frontend should be developed as an independent app. As a result, small teams of developers can be assigned small tasks rather than a large team working on an entire project. This could lead to a faster development cycle for each app, and developers have greater freedom when choosing which tools to use for each task.

**Has an Interface**

Since the application consists of many small independent parts that work together, each part of the system should provide some form of communication channels to other parts in order to be able to work with them and not in isolation. Hence each microservice should provide an Application Programming Interface (API) that enables other microservices to send or receive instructions and exchange data with it [33]. Having an API means that microservices adhere to the principle of encapsulation. Each microservice has the freedom to hide its internal implementation and expose only a channel of communication.

Furthermore, the presence of an API makes the system better aligned with the principles of microservices. Each microservice can be updated or changed without affecting the rest of the system, as long as it respects the same conventions of the original API.

**Replicable**

When many modifications are required for one or more microservices, sometimes a redesign of the microservices or replacing them with new ones could be a better solution. The overall application must be designed in a way that gives developers the freedom to make changes to microservices. Each microservice must be developed in a way that makes replacing it with a new microservice a possibility anytime there is a need for such change. Furthermore, replacing one microservice with another microservice should be a seamless process where other microservices keep doing their usual routine when interacting with the new microservice.

**Respect Interface**

This requirement results from respecting the requirement of having each microservice as a replaceable entity. Since microservices can only communicate through the API, developers of the system should strive to keep each interface unchanged. After each update to the microservice or even after completely replacing an old microservice with a new one, the rest of the microservices should not be affected by changes concerning other microservices. When developers respect the interface style of older or unmodified microservices, other microservices won’t need to change the way they make calls to them. Hence, the application will keep functioning no matter what changes are made to the internals of each microservice.

In case a change of the interface is absolutely necessary, the influence of such change must be kept minimal [30].

**Reusable**

Each microservice should be designed to be as reusable as possible. Since each microservice is designed to handle one task in the application, if the same task is needed in another application then then it should be possible to reuse the microservice that handles this task in the second application [5].

Respecting this requirement leads to saving valuable resources such as time, money and efforts, while keeping the final application more loosely-coupled. Moreover, it could open the door to reusing the same microservices on different platforms. For example, 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. As a result, designing each microservice with reusability in mind helps in keeping the system more loosely-coupled where such microservices will not be dependent on other parts of the application.

Each micro frontend is an independent small application. Hence, it is possible to reuse micro frontends in different applications. Developers should implement their micro frontend as reusable entities. In some cases, a little modification to the styling is required to make the micro frontend more suitable with its new application.

### Requirements of Content trust

This section will discuss the requirements for a content trust mechanism that will be implemented to help microservices have an evaluation of trust among each other when making calls from one service to another.

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.

Content trust has more dynamic nature than other types of trust such as authentication-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 content trust of web resources [17]. Moreover, this implementation is only concerned with having content trust between services in a microservice-based application.

The measures used to evaluate content trust differ in how hard it is to obtain the required information. Each microservice should be able to assess the following information in order to make this evaluation:

* The identity of a microservice
* Service sensitivity
* Number of interacting services
* Evaluation by other microservices
* Age of the microservice
* Last successful activity
* Deception

Content trust requirements of microservices are a combination of policy-based and reputation-based trust. Microservices have to make a decision whether to trust another microservice based on evaluating several factors. There is a large margin of flexibility, so developers could implement a content trust mechanism based on their specific needs for a certain application.

**The identity of a microservice**

When an application starts using third-party microservices, there should be a requirement to verify their identity. A system must be in place to help microservices verify each other and make sure that each is what it claims to be.

A verification mechanism is implemented as a part of the content trust evaluation. Each microservice must have the opportunity to verify the identity of any microservice which makes calls to it. As this establishes the identity of particular microservices, it plays a role in evaluating trust.

**Service sensitivity**

The services provided by the different microservice will vary in nature. Hence the implementation of content trust should help microservices to know how sensitive the service of each microservice that makes a is. Some services will offer routine task which do not process sensitive information, whilst others will. For example, one service might offer routing to help the user navigate from one page to another, while another would offer a login service to the user. The later service has a higher sensitivity than the other.

Microservices should be able to know how sensitives the service they are interacting with are. This knowledge will help microservices to be more strict in terms of the level of trust they demand when dealing with other microservices that handle important data. On the other hand, this knowledge also helps them to be more tolerant with the level of trust required for services which have not been attributed as highly sensitive.

**Number of interacting services**

Each microservice should be able to know the following information about any microservice:

* The number of requests made to a it by other microservices
* The number of microservices that trusted it enough to exchange data and handle a request successfully.
* The number of microservices that did not trust it enough, and denied its requests.

All this information will help when deciding to trust another microservice or not. Basically, a large number of successful interactions between any microservice and other microservice could play a role in increasing the trust in this microservice. Conversely, when a large number of requests have been denied, the trust of this microservice could be affected negatively.

**Evaluation by other microservices**

When a request is made from one microservice to another, each of them should be able read the evaluation of trust given by other microservices. This evaluation represents how much microservices trusted each one of the two involved microservices in their last interaction.

Such information helps the two involved microservices to evaluate their mutual trust. A good evaluation given to one microservice by other microservices will help in increasing the trust in this specific microservice, at this specific time. while a negative evaluation by other microservices will be detrimental.

**Age of the microservice**

Microservice in the system might have different ages in terms of operation, this difference come from the nature of the architecture of microservices itself. Microservices can be 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 fulfill new requirements or fix a newly discovered bug. When making requests from one microservice to another, both microservices should be able to read the age of one another.

The age of microservices can play different roles depending on what the designers want. Some could consider the older a microservice is, the better, assuming the longer a microservice is operating the more trusted it must be by the developers as it did not need to be replaced. On the other hand, some could consider the older a microservice is, the worse. As newer microservice are more up-to-date and could be using better technologies.

**Last successful activity**

Each microservice should be able to show the last time it engaged successfully in a request. This information shows that the concerned microservice is active. A microservice that was not being used for long time raises more suspicious. As a result, such long period of inactivity from accomplishing a task would have a negative effect on the trust evaluation by other microservices.

**Deception**

Each microservice should be able to check if there are any records of deceptions attempts made by a certain microservice. For example, a microservice sending queries instead of the requested data in an attempt to, illegally, fetch data from a database. Such incident, if discovered, should be documented. The involved microservices should be identified.

When a microservice is trying to evaluate its trust of another microservice, it will also check to see if the concerned microservice has any deception record. In case such record exists then the trust would be affected negatively.

### 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-party

When having all the microservices as an internal product, something developed by the same company, then trust between microservices is not as important. The reason for this is that when developers develop a microservice they can be sure that no hidden features are implemented or any malicious script has been put in place intentionally. However, it should anyway be taken into consideration that they may in future decide to introduce some microservices from a third-party.

When some microservices are developed by a third-party, developers must make sure that microservices of both sides will be able to communicate with each other to evaluate their trust. Developers should be able to adapt any third-party microservices. In such a way that makes 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 a certain microservice or not. Failing to provide this information while having a content trust mechanism could create many problems.

Content trust requirements for end-users of the application are outside the scope of this thesis, since the focus is on content trust between microservices. Hence content trust requirements for the end-users of a web application will not be discussed.

## 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 there has not been much discussion in the literature so far. Nevertheless, some re-sources online have discussed it. The rest of this section will present definitions and discussions about micro frontends.

Micro frontends are similar to microservices where 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 [42], micro frontends are independent 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. [42].

In [44] it is argued that Instead of writing a single, monolithic application functionality can be divided into small parts. Moreover, [45] agrees with [42] [43] where it mentions that, micro frontends are the concept of microservices applied to the frontend. Furthermore, [45] presents an important feature of micro frontends which is developing each part using the appropriate technology, where developers can use different toolsets for different micro frontends depending on the need of each frontend.

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], p. 16) also agrees with the above mentioned definitions, it states: “Microservices are relatively small, autonomous services that work collaboratively together”. The authors of [32] go on explaining that microservices architecture is a product of Service Oriented architecture (SOA). The same is also mentioned in [31]

The size of each microservice is also given a considerable amount of attention when discussing microservices. Each microservice should be as small as possible [1]. While [5] mentions that each microservice must implement only one business requirement. And it’s been argued in [33] that no rules have been given to how small each service should be. Both [5] and [33] agree that each microservice should try to represent one business functionality.

The recommended size for microservices, as well as the technique for measuring their size varies from one system to another. Counting how many lines of code each service is as well as counting the number of days each service takes to be developed is suggested in [1]. It is advised that each microservice should not take more than two weeks to be developed [1]. According to [29] the name “micro” suggests that microservices should be small. Like other researchers, [32] describes applications built with microservices architecture as a composition of small services, while [33] mentions that each microservice should only be concerned with implementing one task.

The literature also discuss how microservices should communicate between each other. Firstly, it is mentioned in the definition of microservices architecture, that microservices collaborate with each other [1] [5]. This implies that microservices should interact with each other by exchanging data. Both [31] and [33] mention that each microservice should implement APIs where microservices can use these APIs to exchange data. It is also stated that microservices only communicate with each other using network calls [1]. Researchers in [30] agree that microservices should have APIs to communicate with each other. However, [29] Does not discuss how microservices can communicate with each other, it only mentions that communication between microservices is distributed.

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]. Similarly, there is agreement that each microservice should be independent. For example, [30] explains that each microservice should operate in its own process. [1] and [31] both use the word autonomous to describe the independence of each microservice. “Each service is fully autonomous” states ([31], p. 3) while ([1], p. 16) says that “microservices are small, autonomous services”. In ([5], p. 16) 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. Researchers in [33] Agree that microservices should not affect each other, the term “loosely coupled” ([33], p. 4) is mentioned to describe the nature of the relationship between microservices. The word “isolation” is mentioned in ([1], p. 18) to describe how microservices should not affect each other when changes happen. On the other hand, such isolation could introduce “overhead” ([1], p. 18). The write goes on and describe that microservices should be able to change independently from each other. Researchers in ([5], p. 17) agree with the concept of having microservices independent from each other, they say: “Loose coupling is critical to a microservices-based system”.

Furthermore, [29] suggest that microservices are supposed to be easily-replaced components. The same is suggested in [1] which states microservices should be able to be isolated from the rest of the system or even be completely replaced. Replacement or internal changes should not create complications for the system.

Moreover, the term “bounded context” has been mentioned in ([33], p. 4). It explains that to develop a microservice it is not necessary to know how the other microservices were 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 their communication. They go on to explain that this property gives more freedom to developers to use different tools for different microservices. This same concept is explained in [1] where the author states that such freedom in choosing different tools could help developers in choosing the appropriate tool for a particular 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 architecture follows the principle of loose coupling, in case of failure it is possible to isolated and fix the faulty service while the rest of the system is still operating [1] [30].

### Microservices vs monolithic architecture

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 consists of different services where each service is collaborating with one or more other services. The functionality of the system as a whole is the result of this dynamic interaction between multiple components. 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]. Some researchers even suggested counting lines of code to measure the size of the service [29]. Many other researchers suggest that functionality should be the measure; each service should be concern with handling just one task [33]. This task should be derived from the business requirements.

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 best suited to each service.

Conversely, monolithic applications can be said to be comprised of layers rather than distinct components. They are generally divided into three layers: backend, logic layer and frontend [34]. Each layer covers 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 teams of developers will be more complicated as each team needs to collaborate with other teams during development. Moreover, teams often do not have as much freedom when choosing toolsets for development. They are bounded to what other teams are using and whether a toolset is compatible with the others.

**Scalability**

Applications based on microservices are able to grow when there is a need for new features in the application. Adding new features means adding new services to the system, which is generally an easy task.

However, with monolithic architecture, as more features are added, the application grows to the point where it is hard to scale. The codebase becomes very large and complicated and each additional feature requires a fair amount of work to allow other aspects of the application to adjust appropriately.

The book ‘The Art of Scalability’ [35] introduces the concept of the scale cube. Figure 2.1. illustrates that the Scale Cube has 3 axes an X-axis, a Y-axis, and a Z-axis [35] which represent the following:

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

Generally, applications based on monolithic architecture can scale only on one axis, the (X-Axis). Microservice-based applications, however, 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 only connections between services are via a well-defined interface. Each service can be modified as long as it still respects the conventions of its interface. This is what is meant by loose coupling of the application components. However, monolithic applications developers have to take extra measures to ensure that parts of their applications don’t overlap, which costs them more time and work. The importance of loose coupling is stated clearly in [5].

**Maintainability**

Since microservices are independent entities, it is possible to isolate a particular service in case of failure while the rest of the system continues to operate [1]. However, the same is not be true of monolithic-based applications and in worst cases a single failure in the system could cascade and stop the whole system from operating [1].

In [34] researchers use different factors to compare monolithic-based applications to microservices-based applications. Table 2.1 presents the comparison. They conclude that both architecture styles have advantages and disadvantages. In general, micro-service architecture better suited to projects with a large codebase. But with a small project, building it with microservices architecture could bring 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 monolithic and microservices [34]

### Trust literature review

This part of the thesis presents a review of literature on trust and the different approaches used when adopting trust in software development. First, a definition of trust is presented, to give the reader a basic understanding of trust and its role in this context. 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, and many researchers have formed different definitions of trust and explored what it means. The reason for this is because trust plays an important role in people’s lives and is involved in a broad range of 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 generating a definition of 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], p. 3) the author concludes that there is no single definition of trust which is universally agreed upon; “little consensus has formed on what trust means”. In his research, he agrees with [9] that there has been much discussion on trust and a variety of definitions given. On the other hand, researchers in [18] attempt to give a definition or an explanation of how trust can be evaluated. Their idea is that trust between two parties is a variable with many dependencies.

A distinction between six types of trust is presented in [6]:

* Trusting Intention
* Trusting behaviour
* Trusting Beliefs
* System trust
* Dispositional trust
* Situational decision to trust

While [18] makes a distinction between only two types of trust, execution trust and code trust. Execution trust is trust that the provider of the service will correctly allocate the required resources for the execution, whereas code trust is trust from the side that will be consuming the service that the code does not contain any harmful scripts.

Moreover [18] gives other distinctions for the trust, this extended distinction 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 different from one situation to another. Contrary to this definition, [10] thinks that this type of trust is not a situation specific.

The second type of trust is the *Trusting behaviour* [6]. The definition for trusting behaviour is also given in [11] where it is explained as a voluntarily dependence from one person to another. This dependence is situation-specific where in some cases negative consequences could happen.

[6] Goes further and tries to decompose trusting behaviour into different subcategories:

* Cooperation
* Information sharing
* Informal agreements
* Decreasing controls
* Accepting influence
* Granting autonomy
* Transacting business

Researchers in [12] studied the *trusting behaviour* in their work, named: “Belief in others’ trustworthiness and trusing behavior”. They show that many factors play a role in trusting behaviour, and it is not just about individual gain.

The third type of trust in ([6], p. 33) 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]. In their explanation they give an example of a vendor-consumer relationship.

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] the importance of perceived information and its consequences are discussed. When low quality information is provided but it does not have great importance then the consequences of such false information are relatively low. However, when the provided information has high importance, the negative consequences of trusting this information could be high if it turns out to be of poor quality.

Moving on to the fourth type of trust that was distinguished by ([6], p. 36), this type is called *system trust*. It is described as “the extent to which one believes that proper impersonal structures are in place to enable one to anticipate a successful future endeavour”. Researchers in ([14], p. 197) give an example of system trust, involving an ecommerce system. They concluded that system trust has an impact on the intentions of customers to decide whether or not to make a purchase, “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], p. 38), 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].

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 “([6], p. 38). Although it is recognized as a different type of trust, it does not exhibit much difference from the first type of trust which is *Trusting Intention*.

In his paper about the concept of trust, ([8], p. 55) defines trust as “a ‘leap of faith’ or willingness to be vulnerable”. He argues that trust is a tool learnt at an early age in infancy. People use it as a tool to approach uncertain situations “trust is learned in infancy and enables the individual to deal with the unknowable in the social con-text”. In his explanation for the term ‘leap of faith’ he presents it 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. “([8], p. 56).

However, another definition of trust is also presented in ([8], p. 57) trust is seen as a “social capital”. The author describes the role trust plays for individuals in society and the role each individual plays in society.

Lastly, ([8], p. 59) also presents trust as a component of the “power-knowledge” theory where knowledge leads to power and trust plays an important role in acquiring knowledge.

As can be seen that there is no single definition of trust in the literature and many researchers have put forward different meanings 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 hold trust the system. In case the system rejects the credit card of the buyer, neither party will lose trust in the system. The seller will suspect that the buyer is the cause of this rather than the system itself.

**Policy based-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], p. 85) 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 certain access rights. Furthermore, [19] describes authentication as “allows identity verification of any entity.” Moreover, the authentication of users as “the basic feature of protecting data from computer system intruders” ([19], p. 33). Wallace ([20], p. 2) 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 in [21] as a very important aspect of computer systems security. 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.

In the book “Information Security: Principles and Practices”, [22] encloses the goals of security in three point: “Protect the confidentiality of data, preserve the integrity of data, promote the availability of data for authorized use” ([22], p. 20). Furthermore, [19] states similar points as the task of authentication and authorization.

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.

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.

Additionally, ([21], p. 1) 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. It is also explained that despite all the advances that took place in both hardware and software, the combination of username and password is still in use for verification in computer systems.

([18], p. 86) 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 “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] describes the resource as “an object that is accessed by Subjects. It can be a CPU, a storage device, software, data” ([18], p. 86).

Another interesting definition 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” ([18], p. 86). 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 in [18] as “a set of rules/requirements” ([18], p. 86). 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], p. 1) 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” ([28], p. 1).

However, in [24] trust has been divided into two distinctions one as “strong and crisp” where it uses “logical rules” for making decisions ([24], p. 1). While the other as “soft and social”, according to [24], this distinction is concerned with reputation based trust. “reputation-based trust relies on a ‘soft computational’ approach “([24], p. 1). 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:” first-hand experiences” comes from own experiences and “external experiences” which is recommendations from others based on their own experiences [25].

On the other hand, [26] uses the term “behavioral trust” instead of “reputation trust”. It is defined as realizing the expectation of others. And it is classified into two categories: Direct trust and indirect trust. Where direct trust means the experiences gained 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 a binary one since it is either positive or negative [24]. And such trust depends on well-defined measures such as certificates and is referred to as “strong security” [24].

Also [27] agrees with the mentioned studies [24] [25] [26], it states that “reputation serves as the basis for trust”. Thus, 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 “([17], p. 228). Both types of trust are related to each other’s.

Whether someone trusts some resource online is a personal matter that differs from one person to another. Where each person makes their judgment based on many influences that are affected by personal experiences. [17] mentions that some resources might be preferred to some people over other resources based on the context in which the resources are being judged. The context in which a resource is evaluated is argued more, an example of travel information is given where students may use different source for information than families. The date of which resources are consumed also has a value as stated in [17].

Moreover, [17] identifies 19 factors that influence content trust:

|  |  |
| --- | --- |
| * Topic | * Incentive |
| * Context and Criticality | * Limited resources |
| * Popularity | * Agreement |
| * Authority | * Specificity |
| * Direct experience | * Likelihood |
| * Recommendation | * Age |
| * Related resources | * Appearance |
| * Provenance | * Deception |
| * User expertise | * Recency |
| * Bias |  |

Aditionally, [17] explains that some of the factors are related. And some others can be grouped together such as Direct experience and Recommendation under reputation. Furthermore, [17] acknowledges that determining which of these factors can be put in use is not an easy task.

## 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 analysed 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 microservices 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 them-selves. 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. Figure 2.2 presents the concept of service orchestration



Figure ‎2.2: Service orchestration [39]

In figure 2.2, service consumer acts as a coordinator that coordinates all the services calls to respond to the coming request. Whereas, figure 2.3 shows the concept of service choreography.



Figure ‎2.3: Service choreography [39]

As figure 2.3 shows there is no central service that coordinates communication between services. Each service may call another service independently depending on the context and its needs.

This previous comparison leads to the discussion of microservices architecture and Service Oriented Architecture (SOA). Both architectures are not strange to each other, and microservice architecture is another revision of SOA [32]. Yet there are some main differences among the two. SOA focuses on the concept share-as-much-as-possible while microservices architecture follows the concept of share-as-little-as-possible [39]. This means that SOA-based applications will try to share the resources as much as possible. Such applications will try to share databases, and use other services to handle its tasks. On the other hand, microservice architecture-based applications try to minimize the sharing as much as possible. For example, some microservices will have their own databases. This minimizing of sharing makes the application more loosely-coupled and helps in introducing changes and modifications. According to [39] microservices-based applications use an API layer while SOA-based applications use a messaging middleware. This messaging middleware can be a single point of failure where congestions could slow down the application. On the other side, a single point of failure does not exist in microservice-based applications, as a result these applications are more resilient and handle failures smoothly. Table 2.2 shows a comparison of both architectures where they are compared against some of the requirements mentioned in the first section of this chapter that both architectures share in common.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Scalability | Failures handling | Toolsets diversity | Reusability |
| SOA | **+** | **-** | **++** | **+** |
| MSA | **++** | **++** | **++** | **++** |

Table ‎2.2: MSA vs SOA

In table 2.2 microservices architecture is referred to as MSA. The scale for comparison goes from - -, -, to +, and ++. The lowest score is -- while the highest score is ++. In terms of scalability, both architecture have the ability to give applications a good degree of scalability. Since SOA-based applications are generally, more tightly-coupled then the scalability for a MSA-based application will be easier.

SOA-based applications use what is known as Enterprise Service Bus (ESB) which realize a communication system between the services. This communication system can be a single point of failure when there is a congestion or the ESB fails itself, thus making MSA applications more resilient to failures.

Both architecture patterns enjoy the ability to have their applications developed in different technology stacks where each service or microservice can be developed in different toolsets.

The reusability of microservices is easier to achieve as each microservice is small in size and focuses on one business functionality, while services in SOA can be big and focuses on implementing many business functionalities at the same time, as a result reusing microservices is easier.

To summarize, microservices must adheres to the requirements mentioned in the first section of this chapter. Where each microservice is supposed to be small. How small each microservice can and how to measure such a requirement is up to the developers and the application in hands. Moreover, each microservice is supposed to be independent and can be deployed independently. Having the ability to change one service without affecting other services should also be respected.

To help microservices communicate and exchange data, each one of them must have an interface that helps other microservices when trying to make a request. furthermore, when making changes to a service or when replacing the service completely, the interface of the service should be respected as much as possible. In addition, designing services to be a replaceable entity helps in any future modifications of the application.

Respecting those requirements helps in making the application:

* Resilient
* Scalable
* Easy to deploy
* Possibility of continuous delivery

### 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 micro-services 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 and exchange data safely. In such scenario, having high evolution of con-tent trust among microservices will lead to more exchanged data and cooperation.

Trusting the behaviour 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 more than once in content trust requirements. 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] as well 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 implementation. 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 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 in-creasing 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 in a specific image or not. It can be seen that the name *Content trust* is the same but it is not the same concept as the one being discussed in this thesis. Reputation-based trust is not involved, for example. And it is not a dynamic but more of a static evaluation. Unlike content trust where its value changes with every evaluation and each evaluation depends on many factors.

Following the requirements of content trust mentioned in the first section of this chapter, the mechanism of content trust depends on concepts form policy-based trust as well as reputation-based trust.

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 trust layer between the small blocks. An implementation of content trust could be helpful once the blocks of the application is developed by different sources. Hence it will be a mechanism that helps each service to trust 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 will give an overview of how, micro frontends, microservices, and content trust implementation can all work together to produce a more secure, flexible and robust system.

# Concept

The following pages will present the concept of building microservices-based application where a mechanism of content trust helps services make better selections when making calls to other services. This way, each microservice will have the chance to make a call for the service that gets the highest trust evaluation among other services.

This chapter will be composed of four sections. The first one will discuss the concept of micro frontends and how the frontend of a web application can be divided into small parts. The second section will focus on the concept of microservices and how services interact with each other. While the third section will discuss the content trust that will help microservices evaluate how much they trust each other before carrying on with their exchange of data. Finally, the last section will show how the three parts can work together in the final system.

## Concept of micro frontends

Micro frontends are a sum of small frontends that together form the final page that’s presented to the end-user. The concept behind micro frontends is derived from microservices [50]. Essentially, when applying the principles of microservices to the frontend of a web application the result would be micro frontends. As a result, micro frontends share many of the principles with microservices.

Nevertheless, micro frontends impose few more challenges that don’t exist in microservices. Such challenges include:

* Routing: frameworks such as ReactJS or Vue.js provide tools that help developers write applications that makes navigation from one page to another smooth and easy. When several applications are put together to form the final rendered page. The routing tools of each framework will not have access to the path of parent or overall page. Hence, navigating from one page to another from within a micro frontend is a challenge.
* Data exchanging: When micro frontends are delivered to the browser, they don’t have a default communication channel that help the apps to exchange data. The solution for this challenge depends on how micro frontends are stitched together, and what technologies are used to render the micro frontends in the browser.
* Styling: Micro frontends might face naming conflicts when each micro frontend has its own CSS files. If different micro frontends have the same set of HTML classes, then unwanted styling from one micro frontend might be applied to elements in another micro frontend.
* Events answering: micro frontends should be able to answer events that happens in other micro frontends. For instance, when the final application consists of a navigation micro frontend and some other micro frontends. If the user clicks on a link in the navigation app, a response should be delivered from another micro frontend where the user could be navigating from one page to another. In this case, other apps should be listening to events coming from the navigation app and act accordingly when they are meant to.

Two micro frontends will be rendered to the user at any given time. While the rest of the micro frontends will be rendered depending on the events happening in those two micro frontends. The idea is that, the functionalities of the frontend will be divided into tasks and each task will be handled by one micro frontend. For example, the navigation bar will be handled by one micro frontend, while the main area in the screen or the body of the frontend will be handled by another app. Figure 3.1 shows the positions of the different micro frontends. There are two areas that will be occupied by two apps at any given time. The first area is the navigation bar where it is represented by the dark red colour in figure 3.1. The navigation bar app will be rendered in this position. The second area is the body area and It is represented by a big white rectangle in figure 3.1. This area will host the rest of the micro frontends where only one micro frontend will be rendered to the body area at any given time. Micro frontends will be rendered in the body area according to the events that are coming from the end-user of the Blog. Those events will be applied to the navigation bar and the body area in the frontend.

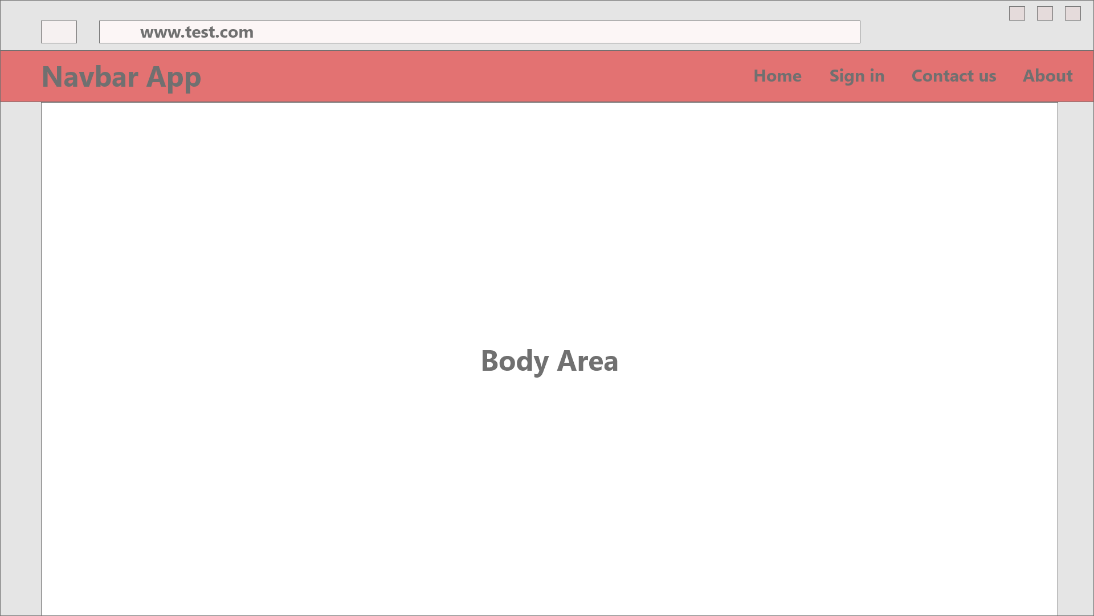


Figure ‎3.1: Micro frontends positions

Next chapter in this thesis will try to find answers for the mentioned challenges and suggests alternative solutions when it is applicable. Moreover, a solution will be provided to putting all the micro frontends together and make them operate in harmony.

## Concept of microservices

The Blog will have two parts, a frontend and a backend. The frontend will be composed of micro frontends, while the backend will include microservices. Some of the microservices will have access to one or more databases while others don’t need such access. Between the microservices a mechanism of content trust will resides.

The Blog will have microservices handling the following tasks.

* Creating users accounts
* Login operations
* Submitting emails to the admins
* Handling posts related requests
* Handling comments related requests.

There will also be services to handle other tasks such as:

* Protecting the Blog from invalid user’s input
* Protecting the Blog from Duplicated posts, comments or messages

Moreover, there will be microservices that will help other microservices by providing them with inputs. Such inputs could be reading certain values from the database, or processing data before submitting it to the responsible microservice.

The Blog will be simple in terms of functionality; it will offer the following services to its clients:

* Reading posts available in the Blog
* Commenting on posts
* Sending messages to the admins of the Blog
* Creating new user’s accounts
* Logging in with the created accounts
* Submitting new posts to the Blog
* Modifying, and deleting own submitted posts

Keeping the Blog simple, will help to focus on the implementation of the microservices, the content trust mechanism between the microservices, and the frontend. The frontend will be composed of many micro frontends. One could describe the Blog as a full-stack microservices application. Both, the frontend and the Backend uses the principles of microservices architecture to deliver the final product.

The services in the Blog will have certain features that make them suitable to be used in a microservice-based application. Such services adhere to the requirements mentioned in the second chapter. The services will be:

* Small
* Independent
* Has an interface
* Reusable

Each service in the Blog should have a small size where it handles one task. For example, one service would handle requests related to storing a new post, retrieve a post from the database or delete a post. Another service would handle clients logins, while a third service handles creating new accounts for new clients.

The reason for making services small is to be able to get the most possible benefits from using microservices architecture. When services are small, it becomes easy to replace them with new services, or isolate a service when it is not running correctly. 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.

Although microservices will be designed to be small but they will not be designed to be too small. For example, the Blog will have one or more microservices handling tasks related to posts such as making a new post, reading posts, deleting own posts...etc. Such microservice could be further divided where one microservice will handle creating a new post, another service handles reading a post, and so on. While such division makes each service handles only one specific task, it will also add overhead and unnecessary complexity to the Blog. As mentioned in [1], when the application has many small parts interacting together, there will be more overhead and complexity added to the application. A trade-off should be considered that helps in following the requirements of microservices while it keeps the complexity of the application as small as possible.

Furthermore, each service in the Blog is as independent as possible where it, generally, 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.

For example, a microservice that creates new user’s accounts will be created. This microservice will have its own database where it stores the newly created accounts. Hence, this microservice and its database are completely independent entities. They can simply be used in any other application that requires user’s registration.

Additionally, services in the Blog should be independent 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 services in the Blog will offer an interface where other services can use it to 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.

Services will offer each other APIs that help them to make requests. Requests will be made over Hypertext Transfer Protocol (HTTP) using REST architecture. REST stands for Representational State Transfer. It is an architectural style that is composed of six constrains. REST helps in developing applications that are loosely-coupled [47]. More details about the constrains of REST will be provided in the next chapter.

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 a user’s 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.

As an example, at least one microservice will offer login services to the clients of the Blog. This service needs access to the database of the registered users so it compares the data it receives from the frontend with the data of the users in the database. This service can be reused in any other application that requires a microservice to handle login tasks. A small modification is required to help the microservice connects with the databases of different applications.

The Blog itself should also have few features that comes from using microservices architecture. After all, if those features are not met then the benefits of using such architecture are not reached to an acceptable level. On the contrary, microservices architecture brings its own challenges. Hence, using such architecture without getting the most of it adds overhead in the development. Where developers have to deal with many small apps and each app has its own development cycle, requirements and deadline. Development of many small apps to finally work together can get quiet complicated. The Blog will be:

* Scalable
* Resilient
* Loosely-coupled

Where new features can be added easily. The scalability of the Blog 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.

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 behaviour when dealing with failures compared to a monolithic application.

Furthermore**,** 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 flexible when facing problems. Or when some services should be replaced by others. It also helps when performing updates on the Blog.

In the future when new functionalities are needed in the Blog, for example, when categories are introduced to classify posts of the Blog into categories then the only changes that are required must happen in the Post microservice. Few changes will be needed to the micro frontends but most of the other microservices will stay the same, where there is no need to touch them.

## 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 [17]. From the requirements provided in chapter 2, it can be seen that the reputation of the involved entities will play a role in 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 about trusting other services or not. 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 while 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 no 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 about the other. If one of the services decides that it can’t trust at least one of the other services, then, theoretically, 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 behaviour.

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 take 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.

Content trust could be implmented as part of each microservice, in this case it is an internal implmentation. Another case could be where content trust evaluation is a microservice itself, thus it is an external implementation. In the first case where content trust is part of each microservice, then the values that is related to content trust that need to be persisted can be spread among all the databases of each mcroservice. Figure 3.2 shows a diagram representing the scenario where each microservice contains an internal implementation of content trust. In order for this case to work, each microservice must have its own database. Although some microservices may not need a database, but content trust implementation requires persistance of data and since data of content trust is stored locally for each microservice, then each microservice needs a database to store its own content trust data.

This implementation violates the size requirement of each microservice. According to [1], each microservice must be small and handles one task. With this method of implementing content trust in a microservice-based application, each microservice must handle its own content trust evaluation on top of its original task. This leads to an extension of each microservice size, moreover, each microservice will be handling at least two functionalities that are its own original task and its content trust evaluation.

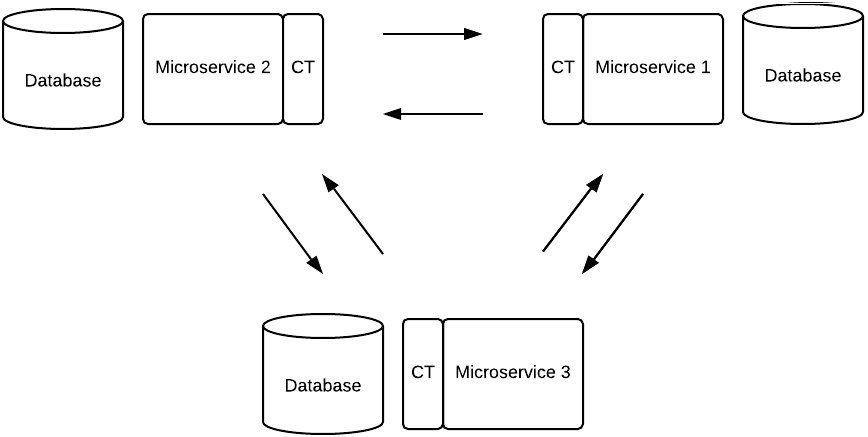


Figure ‎3.2 Content Trust internal implementation

The second possibility of implementing content trust is by having at least one microservice responsible for the evaluation of content trust on behalf of other microservices. Figure 3.3 shows a diagram representing this scenario where content trust implementation is external to the microservices.

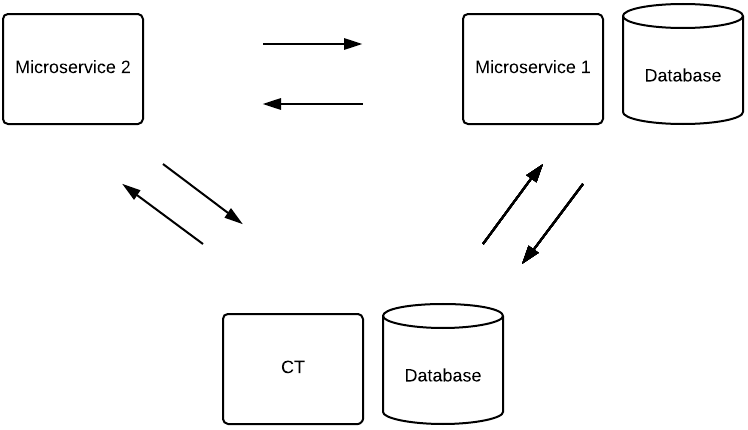


Figure ‎3.3: Content Trust external implementation

Each microservice needs to evaluate the trust about another microservice, then it makes a call to the *Content Trust* microservice with all the required information, *Content Trust* microservice will then calculate the evaluation of content trust on behalf of the calling service and sends the result back. In this case, there will be only one database responsible for storing any data related to the content trust. And microservices are not required to have a database just to store the values of content trust. Moreover, each microservice does not need to care about implementing content trust as part of its functionalities and thus will keep respecting the principles of microservice architecture where each microservice should be handling one task.

External content trust implementation helps in respecting the requirements and principles of microservice architecture, but it poses its own challenges that is mainly having a single point of failure. When the *Content Trust* microservice fails or cannot access its database then the evaluation of content trust will not happen. One solution to overcome this problem is by having more than one microservice calculating the evaluation of content trust. The implementation of the Blog will follow the external model of content trust, because it helps in respecting the requirements of microservice architecture. Additionally, adding more than one *Content Trust* microservice will solve the single point of failure problem, especially since the number of the available microservices is not big and very few *Content Trust* microservices can handle the evaluation of content trust even when some of them fails.

### Properties of the Content trust

Content trust implementation will depend on one database to store the information about different microservices. This database will be accessible to all microservices and its data will change after each call made from one microservice to another.

**Unique Identification**

Each service will have unique identifications, this identification helps services to differentiate and identify each other. To challenge the mechanism of content trust, some services will be developed as an in-house development, while others will be developed as third-party microservices. Both types of microservices should be able to have unique identifications.

**Sensitivity classification**

Each microservice will have a sensitivity classification based on the services it provides. This classification will have different levels and each level belongs to a degree of sensitivity. When services are interacting with each other and evaluating their mutual trust, they will be able to see the sensitivity classification of each other. This classification will help services to decide what is the minimum value of trust required to trust one another.

When a service has a high level of sensitivity then the other service will only trust it until a high value of trust is acquired. On the other hand, if a service has a low sensitivity classification then a low trust would be sufficient to trust the service.

**Direct trust**

Microservices will look into their previous experiences and use it when deciding about trusting new interactions or not. For simplicity, each microservice will only be able to look into the last interaction that it had with any other microservice. When a request is sent from one microservice to another, each microservice will look into their last direct trust evaluation. If the evaluation recorded a good level of trust, then the current evaluation will be affected positively. On the other hand, if the last direct interaction recorded a low level of trust then the current interaction will be affected negatively.

**Reputation-based trust**

Four Types of information will be used from the concept of reputation-based trust:

* The number of interacting microservices for a certain microservice
* The number of successful interactions with other microservices
* The number of failed interactions with other microservices.
* The evaluation of trust given by other microservices for a certain microservice

Each microservice will be able to see how many interactions a certain microservice has. This number will increase with every request this service receives as well as with every requests it initiates.

Each successful interaction with other microservices means that the service has reached a level of trust that was enough to the other service. At the same time, it has trusted the other service enough. When both cases happen then the successful interaction with other services will be increased.

In order for an operation to be successful, both microservices, the one who makes a request, and the one who receives a request, will have to trust each other enough. If one of them could not trust the other microservice enough then the interaction will not continue and the microservice who made the request must look for another microservice to fulfil its request. In this case, the number of failed interactions for both microservices will be increased.

Whenever a microservice makes a request to another microservice or receive a request from it, both microservices will create an evaluation of trust about each other. Hence for any microservice, an evaluation of trust from other microservices might exist. When a microservice makes a call to another microservice, both microservices will be able to see the trust evaluation that’s made by other microservices. In case a microservice has never had any interactions with any other microservice then it will not have any evaluation of trust.

**Time factor**

Time will be used to give two pieces of information that will help microservices in evaluating the trust of each others:

* The operation age
* Last successful activity

Each microservice will have its operation time recorded. When two microservices are interacting with each other, they will be able to see the age of each other. The age of any microservice could have an influence on the evaluation of trust by other microservices.

How the age affects the evaluation of trust depends on the context and how the designers want the system to behave. For the proposed Blog, the older the service the more trusted it is. The reason for this is, older services are still in the system because they have not exhibited any malfunctions that required replacing them, moreover it is a sign that the microservice is handling its task well. So an older age means a better evaluation of trust.

It is worth noting that the same principle may not apply to other applications. Some designers may prefer newer microservices over older ones since they might be more up-to-date. In such case, the evaluation of trust could be higher if the task is new. On the other hand, the evaluation of trust could be low if the microservice is old.

When two microservices are about to interact with each other and they are still evaluating the trust about one another, they will be able to see when was the last time each microservice has a successful interaction with other microservices. A successful interaction means when a certain microservice had a good evaluation of trust by another one while it also evaluated well how much it trusts the other service. Not having a recent successful interaction means the service has failed in trusting other services or was not trusted by other services for long time. Such information will have negative effects on the evaluation of trust.

**Development origin**

Who developed the microservice plays a role in evaluating the trust of it. Some microservices would be in-house developed while others developed by third-parties. Microservices that are in-house developed will have a high rating. Additionally, those that are developed by well-known developers will also have a high rating. Whereas, microservices that are developed by other developers could have a lower rating. It all depends on how the designers of the application would like to give rating and what they consider trusted developers.

**Number of services**

When an interaction between two microservices fails. Then a lack of trust from one or the two involved microservices happened. In this case, the microservice that made the request must make another request for another service that handles the same task. When there are no more microservices available that can handle the request, then the standers of trust of this microservice must be lowered and it must trust the service that has an evaluation of trust as close as possible to what it originally demands. Failing to do so could mean that the request will never be fulfil, hence the original request made by the client of the Blog would be rejected. This scenario must be avoided especially for important applications such as online banking. When a user is trying to make a transaction online then it is not acceptable that his request was rejected.

### Context of Content trust

Now the main points of content trust have been laid out, developers of an application will have to decide themselves on how to use such system. For example, 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 instance: Highly evaluated indirect trust, would have the same effect as highly evaluated direct trust.

On the other hand, a different team of developers developing another application, could think differently. The way they would use the content trust mechanism is similar but with different weighting. For example: A highly evaluated indirect trust doesn’t have the same effect as a highly evaluated direct trust. 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

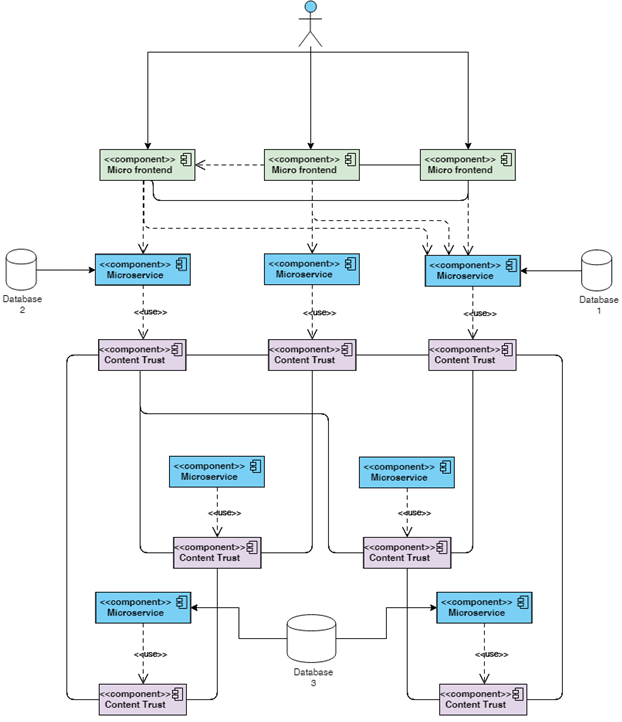


Figure ‎3.4: Overall structure

Figure 3.4 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
* Content trust implementation
* Communication system between services

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 will be very hard for the user to tell where a micro frontend starts and where it ends.

Figure 3.4 shows some services that 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 microservice itself 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 sent back.

A content trust implementation is also proposed. This system will help microservices evaluate the trust of each other. In other words, the content trust mechanism will help services make sure that they interact with other microservices that score the highest trust evaluation. Content trust is also represented in figure 3.4 and has a label to indicate to it.

A communication system among services is proposed to make sure that the application operate as it is supposed to. Such communication is represented by links in figure 3.4. Each micro service in the Blog will have an interface. This interface helps microservice to interact with other microservices. Therefore, services will be communicating with each other to handle user’s requests. Yet, communication will only happen after having both microservices evaluating the trust about each other. If the desired trust level is not reached, then communication may not take place and the involved microservices may look for other microservices that satisfy their desired level of trust.

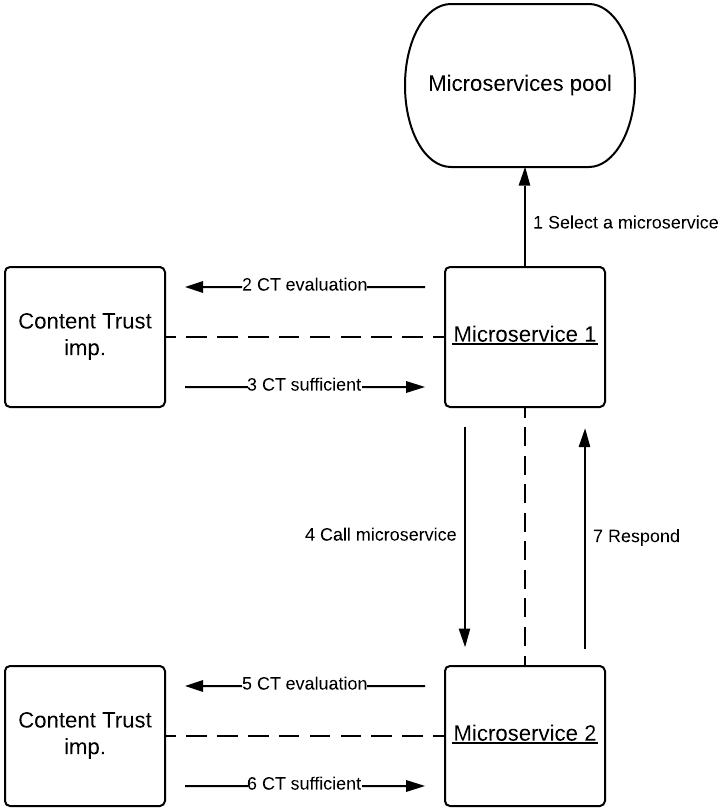


Figure ‎3.5: Microservices collaboration

Figure 3.5 shows an abstract interaction between two microservices. At first a microservice must choose what other microservices it needs to make a call to. Once a microservice is selected then content trust evaluation should be calculated. This evaluation estimates how much this microservice trusts the other microservice. Each microservice has the ability to calculate its evaluation of trust about any other microservice. Once the evaluation of trust has been calculated, it should be estimated if it is sufficient or not. If it is sufficient then this microservice will make its call to the target microservice. The target microservice will start its own evaluation of trust about the microservice that sent her a request. If its evaluation of trust is sufficient then it will respond to the request that it has received from the other microservice.

This diagram shows that at any given time, any involved microservices will have to make an evaluation of trust about each other. Communication between those two microservices will continue if both of them have a sufficient evaluation of trust about each other.

# Implementation

The following pages will present details of the implementation of the Blog. The Blog is developed to show a demonstration of a website based on microservices architecture. Where its frontend is developed based on the micro frontends. And an implementation of content trust to help microservices have an estimation of trust of each other before any exchange of data.

The first section will talk about the implementation of the microservices. It will give an overview of how microservices are implemented. The second section will discuss the implementation of Content trust. It will show what microservices uses to evaluate the trust of each other and how the evaluation process happens. Last section, will discuss the implementation of micro frontends and will provide a brief overview of alternative methods to implement micro frontends in a web application.

## Implementation of microservices

Just like many other websites, the Blog has a backend and a frontend. Both sides are implemented using the microservices architecture concept. The backend of the Blog is composed of many small services, and each service implements one task.

Furthermore, the communication between microservices passes through content trust. This approach helps microservices evaluate the trust about each other and enable them to exchange information with other microservices that achieve the highest trust evaluation each time.

The following services implements the functionality of the backend:

* Registration
* Userid
* Usercheck
* Login
* Islogged
* Search
* Post
* Contact us
* Comment
* Duplication
* Validation
* Content Trust

Each service is responsible for serving one task once requested. There are services that serve the clients of the Blog. On the other hand, some services only serve other services and don’t have any interaction with the users or clients of the Blog.

Some of the services are replicated, where there will be more than one service handling the same task and has the same name. The reason for this is to distribute the load balance across more than one service. And also to make the content trust implementation more efficient.

These services are RESTful web services, meaning that they follow the standards of the Representational State Transfer (REST) architecture.

REST could simply be described in the following scenario where it involves a client and a server. On one side, the server is running a resource (files, database records…etc.) stored in the server. On the other side, a client that requests these resources. The client asks for a resource, basically the data. In the case of the Blog, the data mostly represents posts made by the users, comments, or user’s data. The client doesn’t care about how the data is stored on the server side (what technologies are used…etc.). What it receives from the server is a representational state of the data. JavaScript Object Notation (JSON) is a common format for resource representation in REST architecture.

JSON format has a key-value representation. For example, When the microservice sends the post back to the frontend, it could have the following format:

{

“title” : ”Lorem ipsum dolor sit amet, consetetur sadipscing elitr”,

“body” : ”Nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat“

}

Listing ‎4.1: JSON format

In his famous PhD dissertation [47], Roy Thomas Fielding defines six constrains for REST architecture style. Those constrains are:

1. Client-Server
2. Stateless
3. Cache
4. Uniform Interface
5. Layered system
6. Code-On-Demand
7. Client-Server  
   As described in [47], seperation of the client and the server allows both sides to develop separately making the system loosely-coupled when it comes to the connection between the client and the server. Moreover, the client now has more portability where it can be a web application, a mobile application, or an application operating on any platform. As long as it respects the API offered by the server.
8. Stateless

The connection between the server and the client must be stateless as mentioned in [47]. The request made by the client must contain all the required information that helps the server to satisfies the request. Once the server replies to the request, no information will be stored on the server regarding this request. Any new request must contain all the relevant information, where the client doesn’t assume any knowledge the server could have from answering previous requests.

1. Cache

This constrain is added to improve network efficiency as described by [47]. When the server replies to a request and this reply contains data. This data should be noted as cacheable either explicitly or implicitly. The client cache could then use this data if it is cacheable for later requests.

1. Uniform Interface

Resources that are in the system and could be requested by the client must have one and only one Uniform Resource Identifier (URI). [47] defines four interface constrains, that are:

* Identification of resources
* Manipulation of resources through representations
* Self-descriptive messages
* Hypermedia as the engine of application state.

1. Layered system

This constrains means that the architecture is composed of hierarchal layers. Each component in a layer doesn’t have access to any layer beyond its adjacent layers [47]. This approach has a disadvantage as noted in [47] where it adds overhead and latency to the processing of data.

1. Code-On-Demand

This constrains allow the server to send a script or an applet to the client as part of the response [47]. This constrain improves system’s extensibility but reduces the visibility, hence it is considered optional by [47].

Each service in the Blog is built using Node.js, Express.js, and other modules that are different from one service to another. Node.js is a JavaScript runtime environment that can execute JavaScript outside the browser [48]. When Node.js is installed, Node Package Manager (NPM) will be installed too. NPM helps in adding modules to the application. One can think of modules as packages that can be installed or added to the application. Each module can do one or more tasks that helps making the development faster. It is basically reusable units that the developer can use to achieve certain task without having to rewrite new code to implement the same functionality.

Based on the literature review given in the second chapter and the concept of microservices provided in the third chapter, microservices are small independent unit, that can be deployed and reused when needed. Each service in the Blog is developed based on the concepts presented in the third chapter. Hence many services have their own database. As a result, the Blog uses more than one database to provide its services to clients. There can be more than one services have access to the same database. While other services don’t need to access any database.

The following pages will provide a closer look at some of the services implemented in the Blog. Many services share similar characteristics and have a similar overall implementation. hence, in case of similarity between two or more services, only one example of the implementation will be discussed. And at the end of the thesis, a complete list of the services, with their inputs, outputs and a description will be provided.

### Microservices details

**ContactUs**

This service provides the users with the possibility of contacting the admins of the Blog. Once the user submits a message to the admin, the message will then be stored in the database. This service has its own database. It provides only one API. This API helps the client to send a message to the service. The message must contain a name, an email, and the content of the message.

app.post('/contact', function(req,res)

Listing ‎4.2 ContactUs API

Listing 4.2 shows how the API is provided by the service. The API ends with ‘/contact‘ and starts with the address of the server and the number of the port that the service uses. This service contacts other services to make sure that users are not submitting invalid information. To contact other services ‘ContactUs‘ uses Axios to make HTTP calls. Axios is Promise based HTTP client for the browser and node.js [49]. Promise simply means the final result of the asynchronous operation will be produced in the future. Asynchronous method means the caller will not be blocked while waiting for an answer for its call. While HTTP stands for Hypertext Transfer Protocol. The result of the request could have one of three values: the request is fulfilled, the requested is denied or the request is still pending. A call-back function could be associated with Axios requests to handle the outcome of the request. In such case developers could check the result of the request in the call-back function and act accordingly.

Depending on the results, the service will either store the message in the database or inform the requestor of an error that happened via the result of the API call.

MongoDB is the selected database management system to help store data for services in the Blog. According to [50] MongoDB is a cross-platform document database. It is also known as Not Only SQL (NoSQL) database [51]. MongoDB uses the concept of key-value, where each document has its own auto generated ID. Documents are stored in collections. And a database can have one or more collections. Each collection has one document or more. The internal structure of documents inside collections can be different from one collection to another. In MongoDB, a JSON-like structure can be used where inside any given document, data can be stored in a key-value pair.

const Contact = mongoose.model('Contact',{

name :{

type: String,

required: true

},

email :{

type: String,

required: true

},

content :{

type: String,

required: true

}

});

Listing ‎4.3: Structure of contact document

Listing 4.3 shows the structure of a document that will be stored in the contact database. It has a JSON-like structure. It has attributes as well, for instance, if a certain field is required or not or if it has a default value.

{

"name": "Lorem ipsum",

"email": [ipsum@gmail.com](mailto:ipsum@gmail.com),

“content”: “Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod...“

}

Listing ‎4.4: Contact data example

Listing 4.4 shows an example of how data can be stored in the Contact database. To retrieve the name, one should uses the key “name”. Same applies to the email and the content of the message.

**User Registration**

The microservice “Registration” is responsible for adding new users to the Blog. Any new user can simply submit their name, email and password and the “Registration” service will register the information and create a new user’s account if the provided information has no duplication of user’s email or any other errors.

Once the new user’s data has been submitted to the Registration service from the frontend, Registration service will validate the data to make sure that all inputs comply with the rules regarding the name, email and password. This is done by contacting another service to validate the inputs via a POST request. The next step is checking the input data against the data that’s already stored in the database. Service Registration will make a POST request to another service to check whether the user’s data is unique or not. In case a negative response came as a result for the previous request then user cannot be registered. Otherwise, user’s data will be inserted into the database and Registration service will send a response to the frontend to help it recognize the result of the initial new user registration request.

Registration service has its own database. It uses MongoDB as its database management system. Other services that checks for the uniqueness of the entered data also have access to the same database. Registration interacts with other services such as Validation to check for the validity of the input values, and Usercheck to check if the entered data already exists in the database or not. Services interact with each other via HTTP requests.

**Login**

Login microservice helps users to login to the Blog after they have been registered successfully. Essentially, the Login microservice takes email and password as its inputs, and based on this data the user is either logged in or not. Once the email and the password are submitted to the Login microservice from the frontend, the Login microservice will take those inputs and validate them by passing them to another service for validation via a POST request. If the entered values by user are valid email and password, then the Login service will check this data against the database. If a match is found then user is logged in, otherwise and error message is sent as a response to the request. Which then will be forwarded to the frontend to show the error message.

Logging the user in is achieved via JSON Web Token (JWT). Since REST architecture is stateless where being stateless is one of its six constrains [47] then session based authentication is not suitable for microservices application that uses REST architecture. The principle behind session based authentication is that once the user is logged in, the server will create a new session for the user/client, then it will send the session ID back to the user while keeping the session stored in the server. The client will then store the session ID in a cookie in the browser. With every request the user makes to the server, the cookie will be sent with the request. Once the server receives the request and the cookie, it’ll compare the session ID stored in the cookie with the session that the server has already stored internally. If both matches the user’s request is answered, otherwise, the request is declined.

In the Blog the chosen approach is JWT. Once the user sends a request to login, if the user’s data are valid and a match is found in the database, the *Login* microservice will create a JWT, signs it and sends it back to the client. JSW is created using a secret, that’s chosen by the server. Once the server sends the JWT to the client, no information is stored in the server about that token. Each token has a secret and expiry date.

Microservice Login uses “jsonwebtoken” package. This package can be installed with NPM using the command:

npm install jsonwebtoken

Listing ‎4.5: Installing jsonwebtoken using npm

Listing 4.5 shows how *jsonwebtoken* package can be installed from the command line using Node.JS Package Manager. This package helps in generating and signing the JWT before sending it to the client using the microservice Login.

To sign the token, one piece of data regarding the requesting user is required, in this case, the ID of the user is fetched from the database when a match is found with the input data that’s sent from the frontend. A secret is required; the server is free to choose any secret that is deemed valid by it. And finally an expiry date. This date will be associated with this specific token. Once this date is passed then the server will no longer accepts the token and user will be asked to login again.

jwt.sign({userID: doc.\_id, exp: expirationDate}, secret);

Listing ‎4.6: Generating a signed token

Listing 4.5 shows how a JWT is generated and signed using *jsonwebtoken* package. Once this token is ready, then it is sent back to the client as a result of the request made to *login* microservice. A JWT could have the following shape:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.

TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

Listing ‎4.7: Generated JWT [52]

Listing 4.7 shows an example of what JWT would look like after it has been generated.

**Comment**

This service allows the user to make a comment about a specific post. The user doesn’t need to be logged in, in order to submit a comment. The *Comment* service takes name and email as an input from the user. In addition to the ID of the post which is submitted by the responsible micro frontend to the microservice. *Comment* microservice is also protected by many other microservices to help to protect the database from duplicated comments or bad input from the user. Moreover, *Comment* service has its own database, and all submitted comments will be stored in this database.

**Search**

This service helps the user to search the posts for a specific term. It also calls other microservices to validate the input that it receives from the frontend.

## Content trust implementation

In order for the content trust implementation to work, several properties and features must exist to help microservices evaluate the trust about each other. One database will exist to serve the implementation of content trust. This database will be using MongoDB as its Database management system. It will have two collections:

* Services
* Relations

The collection *Services* will store data about the different microservices that operate in the Blog, this data includes:

* ID of the microservice
* Name of the microservice
* The port of the microservice
* Sensitivity of the microservice
* Development source
* Start date of operation
* Number of interactions
* Number of successful interactions
* Number of failed interactions
* Date of last successful activity

Each microservice will be registered in the database and for each registered microservice there will be an ID associated with it. Each ID is generated by the application automatically for each registered microservice.

The name of the microservice will also be stored in the database. Microservices that offer the same type of services will have the same name. This name will help other microservices in choosing what microservices they want to call.

Sensitivity of the microservice will also be stored in the database. This sensitivity comes from the nature of the task each microservice performs. For example, microservices that handle clients’ logins have a higher sensitivity than microservices that handle bringing comments from the database to the frontend. Essentially, sensitivity of the microservices has three classes:

* High
* Medium
* Low

This classification helps microservices in deciding the minimum requirements for trust evaluation. When two tasks are evaluating their mutual trust, if one of them has a sensitivity classification of low while the other has a classification of medium then their tolerance level will be different from each other. The one that has a classification of low could be trusted even if its evaluation was not considered high. On the other hand, the microservice with the medium sensitivity classification could not be trusted if its evaluation was considered low.

Development source refers to the developers of the microservice. In this case, for simplicity, it will have only two cases, either in-house, or a third-party. The source of the development will affect the trust evaluation based on the value it has. If the microservice has an in-house value, then the effect will be positive. Otherwise, this field will affect the evaluation of trust negatively. Source of the development could be extended to have more than two values. A classification could be given to the developers of each microservice. If the source can be trusted, then this will be reflected as a good value that will affect the trust evaluation positively. If the source is not well known, then it will be reflected as a value that will affect the trust evaluation negatively. This is flexible and depends on how specific the designers of the application want to be. In the Blog, only two values will exist for simplicity and because microservices are all developed by one source, yet some microservices registered as a third-party for testing and evaluation purposes.

The starting date of operation will also be stored in the database for each service. This will help to calculate the age of each microservice. The content trust implementation for the Blog will consider if a microservice has an old age then its trust evaluation will be more positive than if it has a young operation age. Basically, this is a design decision and depends on the context of where the content trust implementation is being used. In other cases, a young operation age could be considered better than an old operation age.

The number of interactions with other microservices for any microservice can help in evaluating the trust. Moreover, the number of successful interactions that each microservice has can also help in evaluating the trust of a certain microservice. The bigger the number, the better the trust evaluation will be. On the other hand, the number of failed interactions can also play a role when evaluating the trust of a microservice. When the number of failed interactions is high then the trust evaluation will be affected negatively.

On the other hand, the collection *Relations* stores the following information:

* The ID of the microservice
* The evaluation given by other microservices to this microservice

*Relations* will have an array of objects, and each object has a key-value pair. Each object contains the port of one microservice as a key and an evaluation of trust as a value. For any microservice, all other microservices will be mentioned in this array. So the ports in the array each represents one microservice. While the values in the array each represents the trust evaluation given by the different microservices. So for each microservice there will be one document in the *Relationships* collection. Those trust evaluation represents the overall trust evaluation that this microservice gained from interacting with other microservices. If the value is null, then no previous interaction between those two microservices happened in the past.

When one microservice wants to make a request to another microservice, first, it must search the *Services* collection to find all the microservices that offer that services it is looking for. Once it gets a list containing microservices identifiers, it chooses one of them randomly. Each service wants to initiate a call with another microservice will send the name of this service to the *content trust* microservice and waits for an answer that contains the port of the microservice that has the highest evaluation of trust for this interaction.

Service.find({name: req.body.serviceName }).then( services =>{

Listing ‎4.8: Service discovery

Listing 4.8 shows how *content trust* microservice will look for all the services that matches the name it received from the calling service. All the services found will be stored in an array. This array will contain objects and each object contains the port of the service and its name as shown in listing 4.9.

svs.push({port:services[i].port,name:services[i].name});

Listing ‎4.9: List of matches

A random microservice is chosen from this array to start evaluating its trust. Once a service is chosen, its index in the array of services will be stored so it doesn’t get to be selected again if its trust evaluation is insufficient and another service must be chosen. The reason for randomly choosing a service is because if services are selected in the order they are found in the database then the first microservice that meets the minimum required evaluation will always be selected. Additionally, its trust evaluation will be enhanced each time while other microservices will never be selected.

Since for each service, its name and port will be stored in a list, the port will be used to get the data of the selected service from the *Service* collection. The port for each service is unique and no two services share the same port. Once this data is obtained, it will be pushed into an array to help with the evaluation later.

sdata.push({\_id:data.\_id,name:data.name, port:data.port, source:data.source,

sensitivity:data.sensitivity, startdate:data.startdate,

lastsuccess:data.lastsuccess, interactions:data.interactions,

successful:data.successful, failed:data.failed});

Listing ‎4.10: Storing service's data

Listing 4.10 shows how the obtained data from the *Service* collection will be stored in the array *sdata* that stands for *service data* for the evaluation.

The next step is to bring all the trust evaluation that other microservices have about the service in concern. This is done by querying the *Relation* collection in the database.

for(v=0;v< retn.services.length;v++){

var key="trust"+c;

if(retn.services[v].port==selfPort)//if the port matchs the port of the service

{

key="strust"; //then this is the direct trust value

sTrustIndex=v;//send the index of the direct trust/self-trust

}

else

c++;

sdata.push({[key]:retn.services[v].trust})

}

Listing ‎4.11: Storing trust evaluations

Listing 4.11 shows how the *content trust* service stores the trust evaluation of the microservices. It loops through all the data available in the *Relation* connection for the selected microservice. It then pushes the trust into the *sdata* array. While pushing the trust values it must detect the previous direct trust. This trust is stored in the last interaction the two microservices did. It also must store the index of the direct trust in the array of trusts that is stored in the *Relation* collection.

*sdata* now has two types of data, the first one is the data obtained from the Service collection, this data is now one object in the *sdata* array. The second one is the trust values obtained from the *Relation* collection. Each trust is now an object by itself in the array. In order to process all the values stored in the *sdata*, they must all be combined in on object. Since having many objects in the array will make it hard to reach the required data automatically.

acc=sdata.reduce( function(acc, x) {

for (var key in x) acc[key] = x[key];

return acc;

}, {});

Listing ‎4.12: Organising the obtained data

Listing 4.12 shows how the data can be combined in one object and stored in the *acc* array. *acc* stands for accumulated, as in the data that has accumulated so far after all the queries and processing.

The next step is to start evaluating the data. First it checks the difference between the successful and the failed interactions. Depending on how much the difference is and whether it is positive or negative, a value will be added to the final trust. If it is negative five points are deducted from the trust, if it is positive and the difference does not exceed 10, then two points are added, otherwise five points are added to the final trust.

Later, the Development Source of the microservice will be interpreted as values between one and ten. The Development source in this case can either be in-house, or third-party. In-house will be evaluated to ten while third-party will be evaluated to five.

The next step is calculating all the trust values evaluated by other microservices, basically, the reputation-based trust. Then the previous direct trust is also added to the final trust. Once all the required values are added to the final trust, this value must be rounded to be in the range 0 to 10. This can be done in several different ways and each way will have an impact on the final evaluation of the trust. These methods will be discussed in the chapter of evaluation which is the next chapter to see how the final evaluation of trust can be affected according to the algorithm used to process all the data in hands.

The very final step is to check if the final evaluation of trust is sufficient or not. This is done be comparing the final trust with the sensitivity of the concerned microservice. The final evaluation of trust will be a value between zero and ten. If the value that’s obtained bigger than ten, it will be considered ten. And if the value is smaller than zero, it will be considered zero.

The required evaluation of trust for each microservice depends on the sensitivity level for each microservice and is as follows:

|  |  |
| --- | --- |
| Sensitivity | Sufficient Trust |
| Low | 1-3 |
| Medium | 4-7 |
| High | 8-10 |

Table ‎4.1: Sufficient trust evaluation for each sensitivity

Depending on the value each microservice has it will be decided if the other microservice will trust it or not.

When a microservice evaluate the trust of another microservice less than the sufficient level then the *content trust* microservice will do all the steps again but with another microservice that will be chosen from the list of the available microservices that it has obtained earlier. In case all the microservices fail to have sufficient trust then the microservice with the highest evaluation among all will be selected.

Once a sufficient evaluation of trust is available, the port of the microservice that scored this evaluation is sent to the requesting microservice. Then *content trust* microservice will start writing data back to the database. It will increase the number of interactions for all the microservices that an evaluation of trust has been made, then it will increase the number of failed interactions for all the microservices that failed to score sufficient trust. And finally, it will increase the number of successful interactions for the microservice that scored sufficient trust. Moreover, it will store the current date in the last successful activity in the *Service* relation.

## Micro frontends implementation

Micro frontends are small applications that form togeher the final frontend. Each micro frontend is an independent application that handles parts of the functionality of the frontend. There are different methods and technologies that can be used to implement micro frontends this includes:

* Single SPA
* iFrame
* Web Components

Discussing and comparing all the possible methods of creating micro frontends is outside the scope of this thesis. A brief overview of some of these methods will be presented as well as a detailed discussion of the used method to implement micro frontends for the Blog.

**Single SPA**

Single SPA framework helps in putting together applications developed in different JavaScript frameworks in one application. Basically, it makes it possible to divide the frontend into units and a assign each unit to a different application. Single SPA is the method that is used to implement micro frontends of the Blog, more detailed discussion about it will be provided.

**iFrame**

Stands for inline frame works in a way that enables developers to include an HTML document inside another one. The biggest drawback of iFrame is that it provides complete isolation for the included document. For example, if a parent HTML document includes four HTML document children, there will be no communication between any two given HTML children. Furthermore, no communication will also exist between the parent file and any child file. As a result, each included HTML file will be an isolated one and any data exchange or events that should be sent from one frontend to another will not take place.

The concept of iFrame is simple and implementing it is a matter of one line of code but it violates at least one of the requirements of the microservices architecture. Where each micro frontend should be able to interact with other micro frontends. Moreover, iFrame does not work well when trying to implement a responsive frontend design, thus micro frontends that use iFrame could end up rendered badly on a mobile device.

**Web Components**

Helps in creating reusable elements that can be used in an HTML document. Essentially, it helps in creating web applications in a modular way. Each application can be divided into smaller units and each unit can be written as a web component. Each web component is a reusable entity, and thus it can be reused many times in the same web application. Additionally, web components can also be reused among different web applications. In simple words, the concept of web components helps in creating custom HTML tags that encapsulate the functionality and the styling thanks to features such as shadow DOM and ES module.

Passing data from one component to another can be done via the properties of each component. Each component can have a set of properties that help in making the component more customizable. And when reusing a component, its properties can be adjusted to suit its new context.

Web components are not yet fully supported by all the widely used web browsers. According to [51] Safari web browser from Apple does not support all web components features, while Microsoft Edge browser support is being implemented now.

### Blog micro frontends

The functionalities of the frontend of the Blog are divided among eight applications. Each application is an independent one and can be deployed to operate autonomously or in another micro frontends-based application. The eight apps are:

* Navbar
* Home
* About
* ContactUs
* Register
* SignIn
* SignOut
* New post

These apps collaborate together to form the final product to the end-user. At any given time, two apps will be rendered simultaneously in the browser. The Navbar application will always be present at the top of the web page, and another app will be present in the body of the page depending on the context and what the end-user is doing. Single SPA library is used to help implement the concept of the micro frontends.

in order for Single SPA to be able to combine different applications into one frontend, it needs the source code of each application, an index file that has place holders for each application and a configuration file. This file will help to register each application in the Single SPA library and points to its entry [52].

Accordingly, the frontend folder consists of the following:

* A source folder that contains eight folders, and each folder belongs to an application.
* An index.html file that contains placeholders for each application
* A single-spa.config.js where applications can be registered with the Single SPA library, moreover, this file helps Single SPA to know the starting point of each application.
* Package.json for the dependencies, settings and Webpack server.

To register an application in the Single SPA the function *registerApplication* must be called. This function takes three arguments:

* Name: the name of the function
* Loading function: Asynchronous call to load the application
* Activity function: basically, this depends on whether the application is active or not and could return True or False accordingly.

registerApplication('navBar', () => import ('./src/navBar/navBar.app.js').then( module => module.navBar), () => true);

Listing ‎4.13 Registering Navbar application

Listing 4.13 shows how Navbar application is registered in the Single SPA library.

Applications in Single SPA have the following lifecycle: Bootstrap, mount, unmount and unload. Each application must implement all the functions of the lifecycle except the unload which is optional [53].

Implementing the lifecycle of each application depends on the framework that was used to develop the application. For applications developed with ReactJS a simple implementation can be like:

const reactLifecycles = singleSpaReact({

React,

ReactDOM,

rootComponent: Home,

domElementGetter,

})

export const bootstrap = [

reactLifecycles.bootstrap,

];

export const mount = [

reactLifecycles.mount,

];

export const unmount = [

reactLifecycles.unmount,

];

Listing ‎4.14: Implementing Single SPA lifecycle

Listing 4.14 shows the implementation of Single SPA lifecycle for an application developed with ReactJS. The base component of the application is specified as *Home.* Once the implementation is done, Single SPA must know where to mount the application. For this, a place holder must exist in the index.html file, and it will be specified using the *domElementGetter* function as listing 4.15 shows:

function domElementGetter() {

return document.getElementById("home")

}

Listing ‎4.15: Specifying the placeholder of the application

Every application in the Single SPA library must follow the previous steps in registering the application, implementing the lifecycle and finally specifying its placeholder.

One drawback of Single SPA is that it does not offer a way of communication between micro frontends. Each micro frontend is not isolated from the other as there is already a way to send events from one micro frontend to another but exchange of data is not possible till this moment. Micro frontends in the Blog need to exchange data. One example of such need is when the user logs into the Blog then all the micro frontends must be notified. The name of the logged in user also must be exchanged between few micro frontends. Moreover, the behavior of the micro frontends could change depending on whether the user is logged in or not. When the user logs in, a JWT is sent back to the *sign in* micro frontend. Later when the user wants to create a new post, this JWT must be used by the *new post* micro frontend when sending the request to the responsible microservice. JWT must sent from the *sign in* micro frontend to the *new post.*

One way to overcome this challenge is by using cookies as a mean of data exchange between the micro frontends. When the user logs in, and after receiving the JWT, a new cookie will be created. This cookie will contain the name of the logged in user and the received JWT. When other applications that are concerned of whether the user is logged in or not are loaded, they check for the existence of this cookie. If it exists and contains a name and a JWT, then the user is logged in and they act accordingly.

For example, when the user logs in and the *sign in* application receives the JWT, a cookie will be created using the following script shown in listing 4.16:

date.setTime(date.getTime() + (min \* 60 \* 1000));

document.cookie = "jwt" + "=" + response.data + "; expires=" + date.toGMTString();

document.cookie = 'email' + "=" + this.state.email + "; expires=" + date.toGMTString();

Listing ‎4.16: Setting a cookie

Other micro frontends can now read the values of those two cookies and act accordingly. When *new post* micro frontend wants to send a new post to the backend, it first reads the JWT from the responsible cookie and sends it along the request to the backend:

var headers = {

"Content-Type": "application/json",

"Authorization": "Bearer " + this.getCookie('jwt')

}

Listing ‎4.17 Reading JWT from the cookie

After obtaining the JWT, the *headers* variable can be sent now with the request to be processed by the backend. It is better to make the validity of the cookie equal to the validity of the JWT received from the backend, so that the frontend will ask the user to log in back again when the JWT is not valid anymore.

With this, the implementation of the Blog is finished. The Blog uses microservice architecture for its backend, micro frontends for its frontend while offering a content trust implementation between the various microservices of the Blog. Next chapter will present an evaluation of the implemented Blog.

# Evaluation

This chapter will provide an evaluation of the implementation that was presented in chapter 4. At first, the evaluation will assist how much the implementation respects the requirements of microservices architecture, it will check the requirements against the implementation for both, the backend and the frontend. Secondly, the content trust implementation will be evaluated, tests will be run to analyze the output and checking of whether the content trust is selecting the microservices with the that score sufficient trust or not.

## Microservices and micro frontends evaluation

## Content trust evaluation

For content trust evaluation, the simulation of the situation where not all microservices are in-house developed. Several microservices will be introduced to the Blog as if they were developed by third-paties. For this assumption to work, the registeration of microservices in the *Trust* database will give each one of these microservices the value of *out* for their source of development. This value indicates that the concerned microservices are not developed by the original development team of the application.

Several tests will be run to extract information about the how the content trust mechanism will react and evaluate services that are developed by third-parties. In real world scenario, microservices could be developed by more than one thrid-party, and thus each development source could carry a different weight when evaluating the the trust for each service. For the situation in hand, microservices are assumed to be developed either by the original development team and in this case they will have the value *in* as a develoment source in the *Trust* database. The second possibility is that they will have the value *out* to indicate that they were developed by a thrid-party. Of course, different weights are given depending on the development source.

Additionally, the test will try to run using different evaluations formulas:

* Direct trust will carry a double weight of the indirect trust while using only non-zero relations.
* Direct trust and indirect trust will have the same weight while using only non-zero relations.
* Number of services will be composed of all the available services while direct and indirect trust have the same weight

The test will run with having the original service that is in-house developed and two additional services that are assumed to be developed by third-parties. Each API call will be repeated 100 times and the call will be considered finished when a value is returned.

**Number of services holds all the services in the Blog while direct trust and indirect trust have the same weighting:**

|  |  |  |  |
| --- | --- | --- | --- |
| In-house | No. of retries | Third-parties | No. of retries |
| 56 | 16/56 | 44 | 21/44 |

Out of 100 calls, content trust mechanism selected the in-house developed service 56 times while the third-parties two services were selected 44 times. Content trust mechanism had to perform 16 recalls for each of the 56 while it had to perform 21 recalls for each of the successful 44 calls. The recalls happen when the selected service does not score sufficient trust and content trust will be evaluated again.

**Number of services contains only non-zero trust while direct trust and indirect trust have the same weighting:**

|  |  |  |  |
| --- | --- | --- | --- |
| In-house | No. of retries | Third-parties | No. of retries |
| 100 | 100/300 | 0 | 0 |

When the average evaluation of the trust is divided by the total number of available services in the Blog, no service will score sufficient trust, and content trust mechanism will evaluate all the available services for each call. Selected services are the ones that scored the highest possible trust. In every case it was the service that is in-house developed.

**Direct trust has double weight of the indirect trust while using only non-zero relations:**

|  |  |  |  |
| --- | --- | --- | --- |
| In-house | No. of retries | Third-parties | No. of retries |
| 72 | 25/72 | 28 | 8/28 |

One in-house service dominated the selection while two third-parties services were only able to be selected just over 25%.

# 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 ‎0.1 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 22. February 2019

[Comments] 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 ‎2.14 „Die Selbstständigkeitserklärung“