TECHNISCHE UNIVERSITÄT
CHEMNITZ

Department of Computer Science

Distributed and Self-organizing Systems Group

Master's Thesis

Microservices & Micro Frontends Web Application using Content Trust

Chemnitz, 8 July 2019

Examiner: Prof. Dr.-Ing. Martin Gaedke
Supervisor: Valentin Siegert M.Sc.

Microservices & Micro Frontends Web Application using Content Trust
Master's Thesis, Department of Computer Science
Chemnitz University of Technology, 8 July 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

List of Tables

List of Listings

List of Abbreviations

SOA	Service-Oriented Architecture
REST	Representational State Transfer
iFrame	Inline Frame
API	Application Programming Interface
JSON	JavaScript Object Notation
XML	eXtensible Markup Language
CPU	Central Processing Unit
HTTP	HyperText Transfer Protocol
JWT	JSON Web Token
ESB	Enterprise Service Bus
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
MSA	Microservice Architecture
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets

1 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 [1]. This architecture, is called layered architecture and it is also known as n-tier architecture [2].

Once the application is ready to be deployed, developers have to approach it as if it is, essentially, composed of two parts: A frontend and a backend. Further division of these two parts 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. Furthermore, updating a layered application and adding new features to it becomes harder the bigger the application is [3].

Microservices architecture tries to overcome the challenges that are imposed when the application is created based on the n-tier architecture. The idea of microservices is to have the system composed of many independent small services 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.

1.1 Current Situation

Microservices is still a new concept, although some companies have already migrated to the microservices architecture such as Amazon and Netflix [4]. There are still many questions that need to be asked when it comes to the microservices architecture. As an example, regarding the calls that happen between services, it should be considered whether services have to select the first one that is available or to follow a different algorithm for services selection.

A mechanism could be in place to help microservices trust each other. This method will select a microservice from a pool of several available options where the selected microservice achieves enough trust level. The concept of trust between microservices is inspired from the content trust of web resources [5].

There is still no standard definition of microservice architecture and there are no clear guidelines for how an application based on microservices should be built [6]. Yet, over the last few years some characteristics for a microservices-based application have been developed and some basic outlines are now commonly used [3] [4] [6] [7] [8] [9]:

- Many small units: A microservices-based application should consist of more than one component. Unlike 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. In some cases, 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.

1.2 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 [3]. The application usually consists of three layers on top of each other [4] [1]. Moreover, some applications could end up having their logic layer divided further into more layers.

Although a three-layered web application is divided into layers, the application is still very tightly coupled [10]. There are many dependencies between the layers. As a consequence, the system will be hard to maintain and update [3].

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.

In microservice based architecture, clients discover services using service-discovery methods. There are two main approaches for this as discussed in [11]:

- Client-side discovery
- Server-side discovery

In client-side discovery, the client makes a request to the service registry, and once it gets an answer containing all the available services that can handle its request, it selects one of them based on load-balancing. Whereas, in server-side discovery, the client sends a call to the load-balancer which does all the work from discovering all the available services to selecting one of them and returning its address to the client.

Those two methods do not take into account the previous performance of each service and no matter if the service handled previous requests well or failed in doing so, there is always a possibility of it getting selected once it is free based on the load-balancing algorithm.

Other alternatives can be helpful in selecting the services based on many factors and not just based on the availability and the overall load of requests. As an example, the previous performance can be taken into account, the better the performance each task has the more it should get selected. Or how satisfied each client with the service it has received.

A mechanism of trust between microservices could help make the interaction more secure. The idea of having microservices trust each other before exchanging data gives microservices the possibility of choosing which services to interact with based on how much trust they have about each other. The evaluation of trust between microservices can be established based on many factors

Such trust is 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 the microservice on the other end of the connection can be trusted then data exchange should take place smoothly between microservices.

1.3 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 [6]. 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 n-tier 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 n-tier 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 a layered architecture [3]. Another reason could be that the application has a complex nature and requires different technology stacks for its various parts.

Microservices architecture is basically one variant of Service Oriented Architecture (SOA) [6], but operations conditions are not quite the same as with traditional SOA [12].

Developers have to decide what kind of communication methods and protocols should be used among microservices. 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 one frontend will respond to changes happening in another frontend although both frontends are independent apps.

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.

1.4 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.

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

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.

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.

1.5 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 web resources.

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. 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 will be presented. The way in which this method is developed will also be outlined and discussed. It will discuss the implementation of the microservices and micro frontends, as well as provide the implementation of the developed content trust algorithm and discuss its different parts.

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 try to provide conclusions and outcomes for the research and how much the developed methods respect the requirements and the concept.

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

2 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 mainly composed of two subsections. One for microservices and one for content trust. Additionally, a third subsection for the requirements of developers and end-users.

The second part will layout the literature review for microservices and for content trust. This section will be divided into two subsections. 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.

2.1 Requirements

This section 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

2.1.1 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, and able to communicate with the already existing microservices, then the system will continue to function.

Since micro frontends are basically microservices architecture applied to the frontend part of the application [13], then the same requirements, in principle, 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* [7], are basically small independent services, that work together. From this definition and from the definitions given in [14] [15], the basic requirements of microservices can be derived.

- Small
- Autonomous
- Has an Interface

Furthermore, respecting the five mentioned requirements results in the following features as described by [7] [14] [15]:

- 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 one 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 and serves some of the features of the architecture, such as creating a resilient application with reusable components.

Each microservice is supposed to be small, where the size of each service should be scaled down until it cannot be reduced anymore [7]. 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. Such approach will help to magnify the gains but also adds more overhead [7]. But the more the 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 can add extra complexity [7]. In general, a compromise must be reached in which there are enough microservices to run the application effectively, without hindering the ability to maintain, manage, scale and modify the system by having too many very small microservices.

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

In his book *SOA Principles of Service Design*, Thomas Erl [16] distinguishes two types of Autonomy:

- Runtime Autonomy (execution)
- Design-Time Autonomy (governance)

Runtime autonomy refers to how much control a service has over its runtime environment, where for example, does it depend on other services or a shared database to run. On the other hand, design-time autonomy refers to how much a service can scale and develop without affecting other services that are using its services.

Having independent microservices enables 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 make requests to this service and exchange data with it [9]. 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.

As for the micro frontend, each app should be able to send and receive data to other apps in the frontend. So far, not all the available solutions to develop micro frontends support the exchange of data among the frontends of the application and establishing such data exchange can be tricky.

2.1.2 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, for example, 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 [5]. 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, basically, 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
- Degree of trust in the developers
- Deception

Content trust requirements in a microservice-based application 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

A system must be in place to help microservices verify each other and make sure that each is what it claims to be. Identity check becomes more urgent when applications start using third-party microservices.

A service discovery mechanism is implemented as a part of the content trust evaluation. Each microservice must have the opportunity to read information regarding the identity

of any microservice which makes calls to it. The identity information of a certain microservice 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 call 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 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 one of them should be able read the evaluation of trust given by other microservices. This evaluation represents how much the rest of the microservices trusted each one of the two involved microservices.

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 comes 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 fulfil 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.

Degree of trust in the developers

The reputation of the developers of each microservice plays a role in the evaluation of trust. If the developers have good reputation, then the evaluation of trust of their developed microservices will be affected positively, otherwise, the reputation of the developers will have a negative influence on the evaluation of trust of their 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.

2.1.3 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. The application should be able to adapt any third-party microservices. In such

a way that makes them able to provide the requested information. Such information as identity, 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 forces the content trust to make its evaluation with less information which could result in bad judgments.

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.

2.2 Literature Review

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

2.2.1 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 resources 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 [17], 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. [17].

In [18] it is argued that Instead of writing a single, monolithic application functionality can be divided into small parts. Moreover, [19] agrees with [17] [20] where it mentions that, micro frontends are the concept of microservices applied to the frontend. Furthermore, [19] 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 [21] is a way to develop an application that is composed of a group of small independent services. Similar definition is given in [9] where it explains: "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 [7]. The definition given in ([14], p. 16) also agrees with the above mentioned definitions, it states: "Microservices are relatively small, autonomous services that work collaboratively together". The authors of [6] go on explaining that microservices architecture is a product of Service Oriented architecture (SOA). The same is also mentioned in [8]

The size of each microservice is also given a considerable amount of attention when discussing microservices. Each microservice should be as small as possible [7]. While [14] mentions that each microservice must implement only one business requirement. And it has been argued in [9] that no rules have been given to how small each service should be. Both [14] and [9] 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 [7]. It is advised that each microservice should not take more than two weeks to be developed [7]. According to [15] the name "micro" suggests that microservices should be small. Like other researchers, [6] describes applications built with microservices architecture as a composition of small services, while [9] 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 [7] [14]. This implies that microservices should interact with each other by exchanging data. Both [8] and [9] mention that each microservice should implement Application programming interfaces (APIs) where microservices can use these APIs to exchange data. It is also stated that microservices only communicate with each other using network calls [7]. Researchers in [21] agree that microservices should

have APIs to communicate with each other. However, [15] 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 [7] [14] [15] [8] [6] [9]. Similarly, there is agreement that each microservice should be independent. For example, [21] explains that each microservice should operate in its own process. [7] and [8] both use the word autonomous to describe the independence of each microservice. "Each service is fully autonomous" states ([8], p. 3) while ([7], p. 16) says that "microservices are small, autonomous services". In ([14], p. 16) it is stated: "Microservices are relatively small, autonomous services". Also [9] agrees that microservices should be independent from each other.

The relationships and the effects each microservice has on other microservices is also discussed. [8] Mentions that when changing the implementation of a microservice other microservices should not be affected. Researchers in [9] Agree that microservices should not affect each other, the term "loosely coupled" ([9], p. 4) is mentioned to describe the nature of the relationship between microservices. The word "isolation" is mentioned in ([7], p. 18) to describe how microservices should not affect each other when changes happen. On the other hand, such isolation could introduce "overhead" ([7], p. 18). The write goes on and describe that microservices should be able to change independently from each other. Researchers in ([14], 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, [15] suggests that microservices are supposed to be easily-replaced components. The same is suggested in [7] 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 ([9], 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. [21] 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 [7] 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 [7] [15] [6]. 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 [7] [21].

2.2.2 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 [22] 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 ([23], 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 [22] that there has been much discussion on trust and a variety of definitions given. On the other hand, researchers in [24] 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 [23]:

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

While [24] 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.

The first type of trust defined in [23] is the *Trusting Intention*. This type of trust means that one is able to depend on others. [23] Argues that this type of trust is different from one situation to another. Contrary to this definition, [25] thinks that this type of trust is not a situation specific.

The second type of trust is the *Trusting behaviour* [23]. The definition for trusting behaviour is also given in [26] 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.

[23] 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 [27] 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 ([23], 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 [28]. 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 [29]. Besides trusting beliefs, two more definitions are given: Disposition to Trust and Institution-based Trust.

In [30] 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 ([23], 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 ([31], 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 ([23], 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 [32].

Lastly, the sixth type of trust according to [23] 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 "([23], 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, ([33], 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. "([33], p. 56).

However, another definition of trust is also presented in ([33], 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, ([33], 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 [31] where he talks about trusting a system. The same concept of trust is agreed upon by [23] 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 resources such as data. In ([24], 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.

Authorization systems are described in [24] as systems that provide certain access rights. Furthermore, [34] describes authentication as a process which "allows identity verification of any entity." and the authentication of users as "the basic feature of protecting data from computer system intruders" ([34], p. 33).

Importance of authentication is described in [35] as a principle aspect of computer systems security.

In the book *Information Security: Principles and Practices* ([36], p. 20) summarises 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".

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 who is trying to access one or more resources.

Additionally, ([35], p. 1) attempts 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 this simple well-known identification method. Based on a username and a password. It is also explained that despite all the advances that have taken place in both hardware and software, authentication by username and password is still in widespread use for identity verification in computer systems.

([24], 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". They move on to define a service "a piece of software that provides some functionality and can be accessed by Subjects or other Services". They also give a definition for a resource, "an object that is accessed by Subjects. It can be a CPU, a storage device, software, data" ([24], p. 86).

Another interesting definition relates to the requirements given by each service in order to be accessed. This is called Service Policy, which refers to "the set of rules/requirements

associated with the Service. A Subject must conform to Service Policy in order to Access that Service" ([24], p. 86). The concept of access that is granted to reach the requested service is also defined: "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 [24] that access to the service is only granted if the service policy of the service is respected. Policy itself is also defined in [24] as "a set of rules/requirements" ([24], p. 86). This set of rules can be linked to the Subject, the Service or even the Domain, according to [24].

Reputation based trust

Reputation based trust could be used in a system where users published reviews of other users. One of the earliest examples of it was adopted by eBay. As ([37], p. 1) refers "Reputation systems are already being used in successful commercial online applications". A similar idea is put forward in [38]: "Reputation-based trust systems were mainly used in electronic markets, as a way of assessing the participants" ([38], p. 1).

In [39] trust has been divided into two distinctions one is "strong and crisp" where it uses "logical rules" for making decisions ([39], p. 1), while the other as "soft and social", according to [39], this distinction is concerned with reputation based trust: "reputation-based trust relies on a 'soft computational' approach "([39], p. 1). In this case, trust is computed from two sources: First, based on one's own experience, and secondly based on experiences of others, as referred to by [39]. Moreover, trust depends on other factors such as time and particular settings [40].

The same concept for computing trust is used in [41], where it agrees that reputation-based trust is computed from two sources:" first-hand experiences" of our own and "external experiences" which is recommendations from others based on their own experiences [41].

On the other hand, [40] 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. Direct trust means the experiences gained from one's own direct interaction, while indirect trust means other's experience of interactions. It is obvious that [40] is presenting a similar concept to [41] which refers to first-hand experiences and external-experiences rather than direct trust and indirect trust. In both definition, the resulted trust is variable and never constant, as its value changes after each

interaction, whereas with policy-based trust the resulting decision is a binary one since it is either positive or negative [39]. Such trust depends on well-defined measures such as certificates and is referred to as "strong security" [39].

[42] also agrees with the mentioned studies [39] [41] [40], it states that "reputation serves as the basis for trust". Thus, a lot of value is assigned to the experiences of other entities in the system.

A distinction between entity trust and content trust is given in [5]. Entity trust is given as an evaluation of an entity based on its ID and behaviour, whereas content trust is defined as "A trust judgment on a particular piece of information in a given context "([5], p. 228). Both types of trust are related to each other.

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. [5] 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 also important, an example of travel information is given where students may use different source for information than families. The date on which resources are consumed also has an effect, as stated in [5].

Moreover, [5] identifies 19 factors that influence content trust:

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

Additionally, [5] explains that some of the factors are related, and others can be grouped together such as 'Direct experience' and 'Recommendation under reputation.' Furthermore, [5] acknowledges that determining which of these factors can be put into use is not an easy task.

2.3 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.

2.3.1 Microservices analysis

Microservice-based applications can be built using one of two architecture styles, these are orchestration and choreography [43]. With orchestration, the work flow between services is managed centrally. One or more services are directing the calls to their intended destination. Hence, the application has a central part to manage traffic and help services communicate with each other. On the other hand, services in an application based on the choreography style architecture should handle any calls by themselves. They should identify the destination service either by its address or by the type of service it offers. Unlike orchestration, choreography offers more loosely-coupled architecture since it is decentralized [43]. As a result, such application could benefit more from having microservice architecture. Figure 2.1 presents the concept of service orchestration

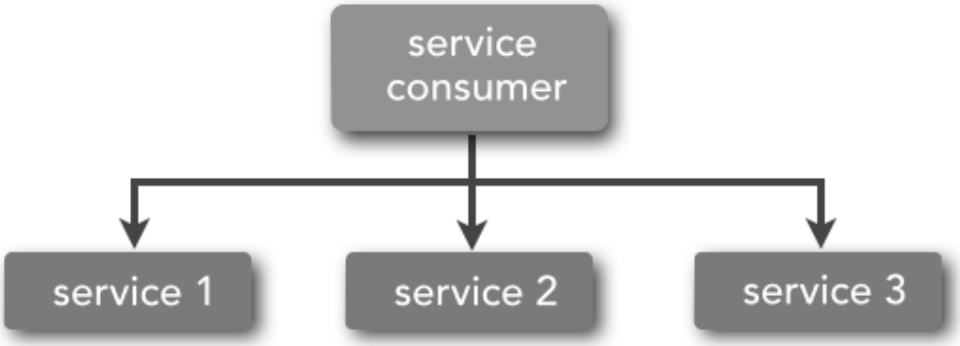

Figure 2.1: Service orchestration [12]

In figure 2.1, service consumer acts as a coordinator that coordinates all the services calls to respond to the coming request. Whereas, figure 2.2 shows the concept of service choreography.

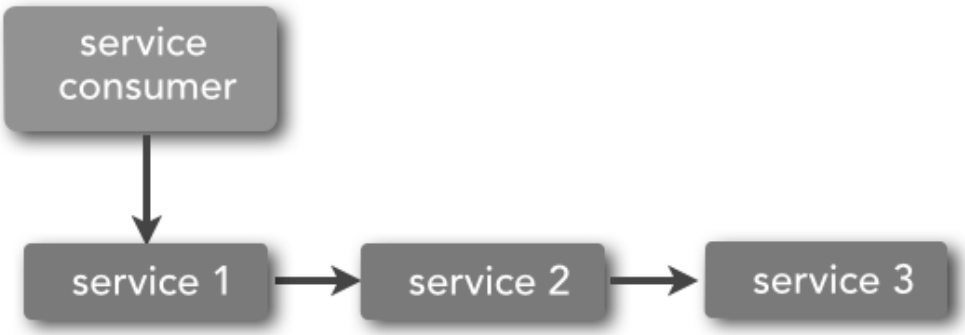

Figure 2.2: Service choreography [12]

As figure 2.2 illustrates. 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 comparison leads to the discussion of microservices architecture and Service Oriented Architecture (SOA). These architectures are not strange to each other, microservice architecture is another revision of SOA [6]. Yet there are some principle 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' [12]. 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 this sharing of resources 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 [12], 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 congestion could slow down the application. A single point of failure does not exist in microservice-based applications, and as a result these applications are more resilient and handle failures gracefully. Table 2.1 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 2.1: MSA vs SOA

The scale for comparison goes from -, - -, to +, and ++ which is the best possible score. It terms of scalability, both architectures have the ability to give applications a good degree of scalability. Since SOA-based applications are generally more tightly-coupled, scalability for a MSA-based application will be better.

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 with different toolsets.

With microservices it is easier to achieve reusability, as each microservice is small in size and focuses on a single business functionality, whereas services in SOA can be large and focus on implementing multiple business functionalities at the same time. So as a result reusing microservices is easier.

To summarize, microservices must adheres to the requirements mentioned in the first section of this chapter. Each microservice should be small. How small each microservice should be and the method of measuring its size is up to the developers and the application at hand. Moreover, each microservice should be independent and able to 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 which other microservices can use when making a request. furthermore, when making changes to a service or when replacing the service completely, the interface of the service should continue to respect the same conventions as before 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
- capable of continuous delivery

2.3.2 Content trust analysis

Different definitions of trust were presented by different researchers. Some of the definitions intersect with one or more of the provided requirements of content trust. One definition for trust distinguishes 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 ([23], p. 4) mentioned "trust leads to cooperation" this understanding is also exhibited in the requirements of content trust. The point of adapting content trust in microservices is to give the microservices another way to select which services they want to interact with. This selection is based on the concept of trust. In such scenario, having high evolution of content trust among microservices will lead to more exchanged data and cooperation.

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 [23] as well as in [32].

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 [5]. 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 [30] 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 [30]. 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 [30] 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 [35] 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 [34]. 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 an already used implementation of content trust. The most common one is what 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 [44], 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 does not go any further, for example: users of Docker images cannot provide an evaluation of their experience after using a specific image. Hence other users cannot 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.

Table 2.3 shows a comparison of three methods of selecting services to make a call to, in a microservices-based application. The first method is service discovery, the second one is reputation-based selection and the last one is Content Trust.

Method	ID Check	Sensitivity	No. interactions	Time Context	Reputation	Deception
Service Discovery	+	-	-	-	-	-
Reputation-based	+	-	+	-	+	-
Trust evaluation	+	+	+	+	+	-

Table 2.1: Service Selection Methods

The comparison checks if a selection method supports the requirements of content trust mentioned in the first section of this chapter. The comparison is not about to which extent does each method support certain requirements, and hence each requirement is either supported or not. A support is indicated by a plus sign + while the absence of support is indicated by minus sign -.

The first method which is service discovery provides support for identity check of the selected service. It selects a service out of a pool of services and checks to see if the selected service provides the desired task, this includes ID check. While this method provides no support for any other requirement. The second method is selection based on the reputation of the service. This method supports ID check, and the reputation of the service affects its selection as well as the number of its past interactions. But it does not offer any support for the rest of the requirements. The last method is the selection based on the trust evaluation, this method satisfies all the requirements except the last one. Trust evaluation does not provide any support for checking of deceptions attempts because it does not process the body of the request made from one microservice to another. Trust evaluation calculates an evaluation of trust for the service itself in a given time and context, while no support is given for checking any scripts sent from one service to another.

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.

3 Concept

The following pages will present the concept of building microservices-based applications 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 smaller parts. The second section will focus on the concept of microservices and how services interact with each other. 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.

3.1 Concept of micro frontends

Micro frontends are a sum of small frontends that together form the final page which is presented to the end-user. The concept behind micro frontends is derived from microservices [13]. Essentially, when applying the principles of microservices to the frontend of a web application the outcome will be micro frontends. As a result, micro frontends share many of the principles with microservices.

Nevertheless, micro frontends impose a few more challenges that do not 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 do not 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 happen 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, the 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. 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 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.

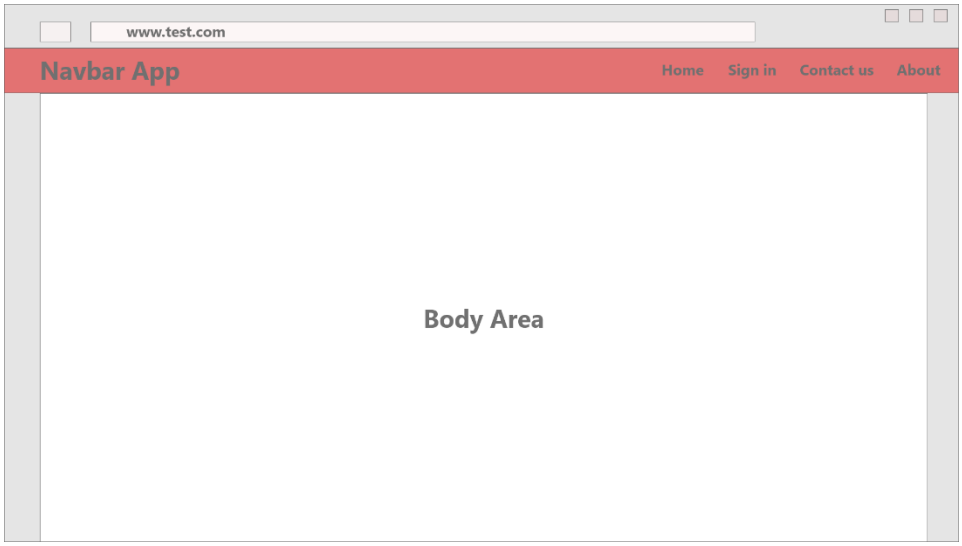

Figure 3.1: Micro frontends positions

The next chapter in this thesis will provide a solution to put all the micro frontends together and combine them into one app while preserving the autonomy of each app. This solution includes addressing the challenges that are mentioned in the previous page and providing alternative solutions when applicable.

3.2 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 do not need such access. Between the microservices a mechanism of content trust will reside.

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 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 users' accounts
- Logging in with the created accounts
- Submitting new posts to the Blog
- Modifying, and deleting own submitted posts

Keeping the Blog simple, helps to focus on developing a microservice-based web application that introduces the concept of content trust among the different microservices. 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.

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

Although microservices will be designed to be small, 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 microservices 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 it so that each service handles only one specific task, it will also add overhead and unnecessary complexity to the Blog. As mentioned in [7], when the application has many small parts interacting together, there will be more overhead and complexity added to the application. A trade-off is considered in the Blog to help in following the requirements of microservices while keeping the complexity of the application as low as possible.

Microservices are independent. Each service in the Blog is as independent as possible where it generally, does not rely on other services to perform its task. Not all services have the same degree of autonomy. The more dependencies each service has the less independent it is. A good design can help microservices be as independent as possible while respecting the specific requirements each service has. The more independent each service is, the easier it is to form a loosely-coupled application. Microservices will be designed to follow the principle of 'share-as-less-as-possible', the word 'share' here is concerned with dependencies such as a shared database and it is not related to data exchange between microservices.

For example, a microservice that creates new users' accounts will be created. This microservice will have its own database where it stores the newly created accounts. This way, more autonomy is given to this microservice since it is using its own database and not a shared resource. Hence, this microservice and its database are completely independent entities. They can simply be used in any other application that requires user's registration.

Microservices in the Blog are not isolated from each other. Services in the Blog should be independent but this does not mean that services will act as isolated islands where no communication is happening amongst them. Services 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 [45].

Services are reusable entities. Since each service is performing one small specific task, it means that there is a high chance that the same functionality will be needed in other applications. For example, a service that is 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. Since most services are designed from the beginning as reusable entities, it is easier to do it this way than to take each service and adapt 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 that it can compare 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 connect with the databases of different applications.

The Blog must be able to handle failures where they do not cascade in a way that affects other services and stops the Blog from operating. The Blog must be flexible in a way that allows for failures isolation where the malfunction services are isolated from the rest of the Blog. They can then be temporarily 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 an n-tire application.

The loosely-coupled structure of the application helps the Blog to be more flexible when facing problems or when some services need to 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.

3.3 Concept of content trust

Content trust as defined in [5] 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 plays a role in the final judgment.

Content trust and reputation trust are related but they are not the same [5]. 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 on whether or not to trust other services. On the other hand, such mechanism of content trust must be designed with care, otherwise the system could behave in an unpredictable way. When such system is not given a thoughtful design and enough preparation and testing, then sometimes services could end up making negative judgments on each other. Such negative judgment could happen although 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 for the other. If one of the services decides that it cannot trust at least one of the other services, then, theoretically, the transaction may not take place. The system will eventually refuse to complete the transaction leaving the client with an unhandled request. Hence, clients could end up leaving such system and never using its services because of its unpredictable behaviour.

This kind of scenario, raises many design questions. One such question is whether or not a system should have more than one service providing the same service. In the previous example, if one service cannot 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 and it consumes more resources. The Blog tries to avoid the previously mentioned scenario. Microservices will have more than one option when deciding to make a call to other microservice. If all trust evaluation fails in achieving enough trust, then the microservice that scores the highest trust evaluation will be selected.

Content trust could be implemented as a part of each microservice and in this case it is an internal implementation. 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, the values that are related to content trust which need to be persisted can be spread among all the databases of each microservice. Figure 3.2 shows a diagram representing the scenario where each microservice contains an internal implementation of content trust. This implementation is referred to as CT in figure 3.2 which stands for Content Trust. It can be seen in figure 3.2 that content trust implementation is depicted as an internal part or an extra functionality for all the available microservices. In order for this case to work, each microservice must have its own database. Although some microservices may not need a database, content trust implementation requires persistence of data and since data of content trust is stored locally for each microservice, then each microservice is forced to have a database to store its own content trust data. An alternative solution can be where only one database is responsible for storing the data of content trust for each microservice. This situation means that every microservice will need access to one database at least while some microservices may need access to more than one database where the first one is the database for the content trust and the other databases are for their original data. This design results in adding an extra dependency for each microservice, this dependency is content trust database, and a single point of failure in case content trust database fails.

This implementation violates the size requirement for each microservice. According to [7], each microservice must be small and handle 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. Additionally, the implementation of content trust is the same for all the services of the Blog and adding this implementation as a part of each microservice means that the implementation will be repeated as many times as there are services in the Blog. Hence more redundancy is added to the microservices of the Blog which is a sign of a bad design.

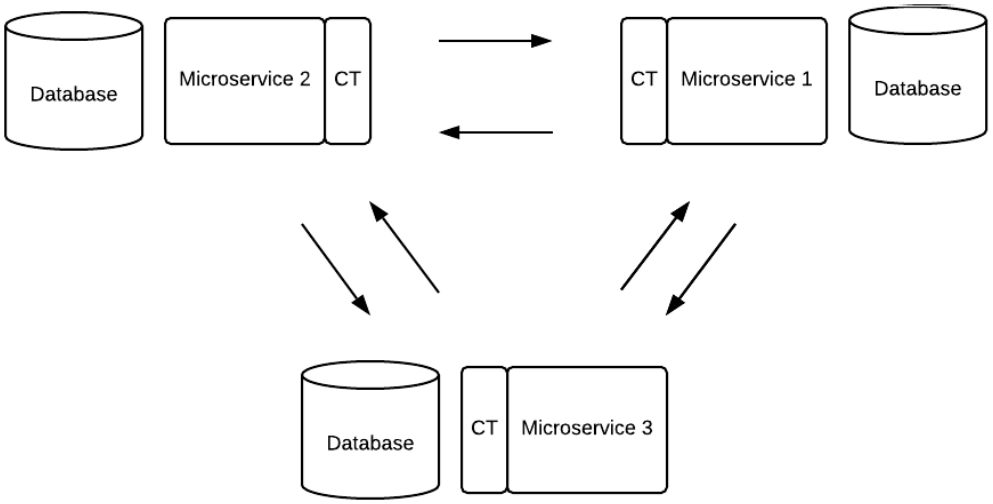

Figure 3.2 Content Trust internal implementation

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. Content trust is illustrated as an independent process and labelled as CT.

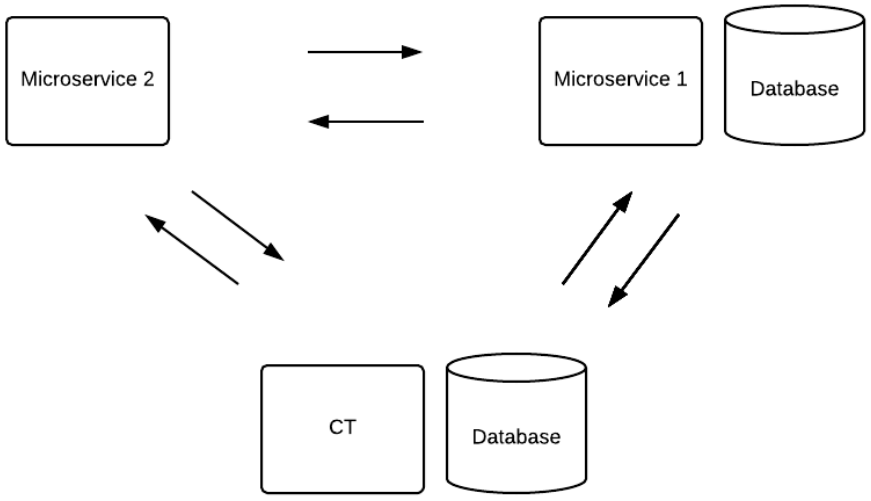

Figure 3.3: Content Trust external implementation

When a microservice needs to evaluate the trust about another microservice, 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 then sends the results back. In this case, there will only be one database responsible for storing any data related to the content trust. Additionally, microservices are not required to have an extra database just to store the values of content trust. 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 only one task.

External content trust implementation helps in respecting the requirements and principles of microservice architecture, but it poses its own challenges, mainly related to the fact that it will have 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. Considering both designs and their advantages and disadvantages, the Blog will follow the external content trust implementation concept. This design helps the Blog to better respect the requirements of microservices 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 few *Content Trust* microservices can handle the evaluation of content trust even when some of them fail. This solution helps in solving the redundancy of content trust data where this data will not be repeated in the different databases of each microservice. On the other hand, it could also mean that when the *Content Trust* microservice is duplicated, its database might need to be duplicated too. Thus, there will be some redundancy but to a lesser degree.

3.3.1 Properties of the Content trust

Content trust will be implemented as an independent microservice. It has its own database and an interface. Any microservice wants to evaluate the trust of another microservice, it can make a call to the *Content Trust* microservice to calculate the trust on its behalf. Once the trust evaluation is calculated, *Content Trust* microservice will return a response to the calling microservice. Some information about each microservice must

exist in order to help *Content Trust* microservice to calculate the trust when requested. This information is:

Unique Identification

Each service has a unique ID, this ID helps in identifying each service and processing its data in the database. 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 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 if 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

Content Trust microservice will look into the previous experiences of the two involved microservices. These previous experiences help when evaluating the trust about a certain microservice. For simplicity, *Content Trust* microservice will only be able to look into the last interaction that took place between any two microservices. When a request is sent from one microservice to evaluate the trust about another microservice, *Content Trust* will look into their last mutual 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 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:

- 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

Content Trust 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 request it initiates.

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

Time factor

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

- The operation age
- Last successful activity

Each microservice will have its operation time recorded, basically the start date of its deployment. *Content Trust* microservice is able to see the start date of any microservice. The age of any microservice could have an influence on the evaluation of trust by other microservices. This is another option of how the data about each service can affect the trust evaluation. Developers of any system could decide when using a content trust implementation of whether or not they want to take the operation age into consideration.

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 that, older services are still in the system because they have not exhibited any malfunctions that required them to be replaced. Moreover, it is a sign

that the microservice is handling its task well. Therefore, 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 of one another, *Content Trust* microservice will be able to see when was the last time each microservice had a successful interaction with other microservices. Not having a recent successful interaction means that the service has failed to trust other services or was not trusted by other services so far. This includes the case of never been called or a request was never made to it. Such information will have a negative effect on the evaluation of trust.

Development origin

The developers of the microservice play a role in the evaluation of its trust. Some microservices will be in-house developed while others could be developed by third-parties. Microservices that are in-house developed will have high ratings. Additionally, those that are developed by well-known developers will also have high ratings. 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 ratings and what they consider to be trusted developers.

Number of services

When an interaction between two microservices fails, this means that the lack of trust from one or the two involved microservices occurred. 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 standards of trust of this microservice must be lowered. It must then 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 fulfilled. Subsequently, 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 if the request is rejected.

Request Body check

Content Trust microservice will not offer any interface to check the body of a request that is sent from one microservice to another. The point of checking the body of the request is to make sure that no microservice sends malicious scripts to another microservice in an attempt to access certain data that it is not supposed to. One method of checking the body of the request is by using Artificial Intelligence techniques and deep learning methods where *Content Trust* microservice keeps improving its performance and its bad script detection ratio over time. Implementing this is outside the scope of this thesis.

3.3.2 Context of Content Trust

Now that the main points of content trust have been laid out, developers of an application will have to decide for themselves on how to use such a mechanism. 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 whether or not to trust a service. For instance, highly evaluated trust by other microservices would have the same influence as highly evaluated mutual trust by the two involved microservices.

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 trust by other services does not have the same influence as a highly evaluated mutual trust. Developers might consider that for a particular application, past mutual experience should have more effect than the evaluation of trust given by other microservices. This will then 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 implementation of trust but with different weighting systems. Hence the trust relationships between involved entities will be different.

3.4 Overall structure

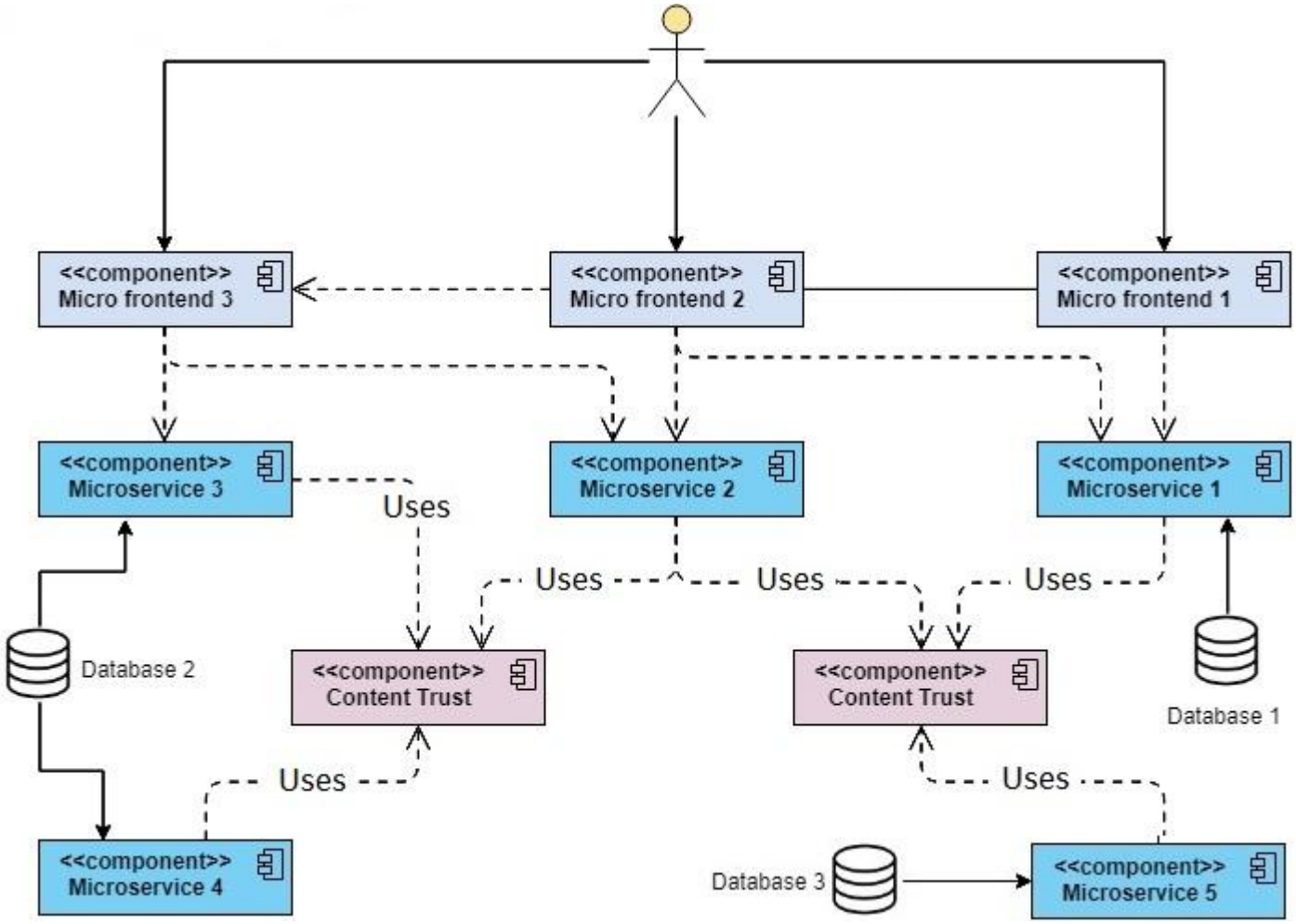

Figure 3.4: Overall structure

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 has a frontend that helps end users interact with it. The frontend consists 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 is interacting with the frontend will not be able to notice any difference. 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 services that have a direct contact with a database such as microservice 1, while other services do not have any contact with a database like microservice 2. Additionally, more than one service can have access to the same database such as microservice 3 and microservice 4. This simply explains that the system will have more than one database. The reason for this comes from the definition of a microservice that is 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.

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 operates as it is supposed to. Each micro service in the Blog has an interface. This interface helps microservices to interact with other microservices. Therefore, services will be communicating with each other to handle users' requests. Yet, communication will only happen after both microservices evaluate the trust of 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.

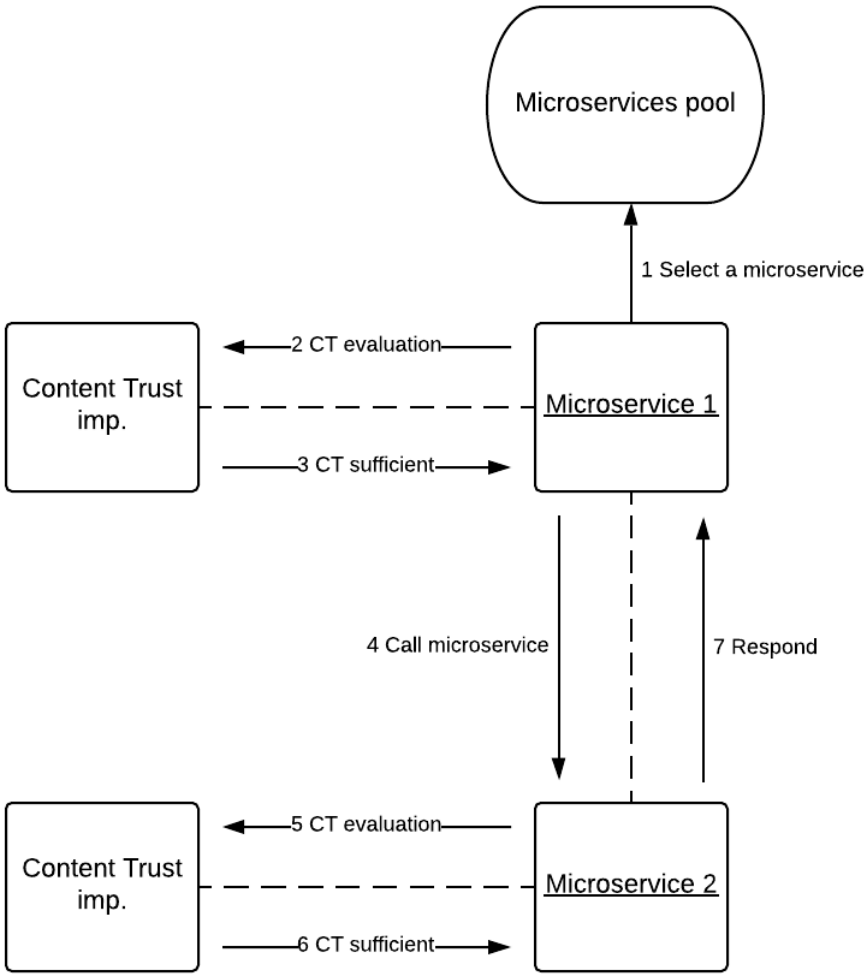

Figure 3.5: Microservices collaboration

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 for any other microservice. Once the evaluation of trust has been calculated, it should be estimated whether 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 for the microservice that sent 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 for each other. Communication between those two microservices will continue if both of them have a sufficient evaluation of trust for each other.

4 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 which uses an implementation of content trust. While content trust is used to help microservices have an estimation of trust of each other and select the service that has the highest estimation to interact with.

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 use 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.

4.1 Implementation of microservices

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

The communication between microservices passes through content trust mechanism. 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 implement 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. Whereas, some services only serve other services and do not have any interaction with the frontend 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 in case the evaluation of trust of one service fails then there are other options available.

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, the 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 users' data. The client does not care about how the data is stored in the server (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 a post back to the frontend, it can have the format shown in listing 4.1:

```
{
     “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 [45], Roy Thomas Fielding defines six constrains for REST architecture style. Those constrains are:

- Client-Server
- Stateless
- Cache
- Uniform Interface
- Layered system
- Code-On-Demand

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 [46]. When Node.js is installed, Node Package Manager (NPM) is 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. Basically, modules are reusable units that the developer can use to achieve certain tasks without having to write own new code to implement the same functionality. Node.js is selected as a framework for the implementation because it uses JavaScript which is widely-used programing language. Moreover, possibility of adding packages to the project to make the development time faster and more efficient with well tested and widely accepted packages makes Node.js an attractive choice. Additionally, JavaScript is easy to understand and many developers already know the syntax or some of it which makes it a good choice for compatibility and for any future reviews and development by anyone interested.

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 do not need to access any database.

4.1.1 Microservices details

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. At the end of the thesis, a complete list of the services will be provided.

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 [47]. 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 [48] MongoDB is a cross-platform document database. It is also known as Not Only SQL (NoSQL) database. 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 or more documents. 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,
     “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 use 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 is 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 the new 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 an email and a 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 the user are valid email and password, then the Login service will check this data against the database. If a match is found then the 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 [45] 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 will 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 an 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 which is read from the database when a match is found with the input data that is sent from the frontend. A secret is also required to sign the token, 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 the 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 [49]

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 does not need to be logged in, in order to submit a comment. The *Comment* service takes a name and an email as 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.

4.2 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 uses 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
- Trust of the developers
- 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 based on the trust of the developers of the microservice. If the developers can be trusted, then this will be reflected as a good value that will affect the trust evaluation positively. If the developers are 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 and what they consider to be a trusted and what is considered not trusted.

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. These trust evaluations represent 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.

Figure 4.1 illustrates the algorithm that is used by the *Content Trust* microservice. Content trust evaluation is run, at most, as many times as there are instances of the called service at any given time. In case the final trust evaluation fails in scoring sufficient value, then *Content Trust* microservice had already evaluated the trust of all the available options. The service with the biggest trust value will be selected. The steps shown in figure 4.1 are discussed in the following pages.

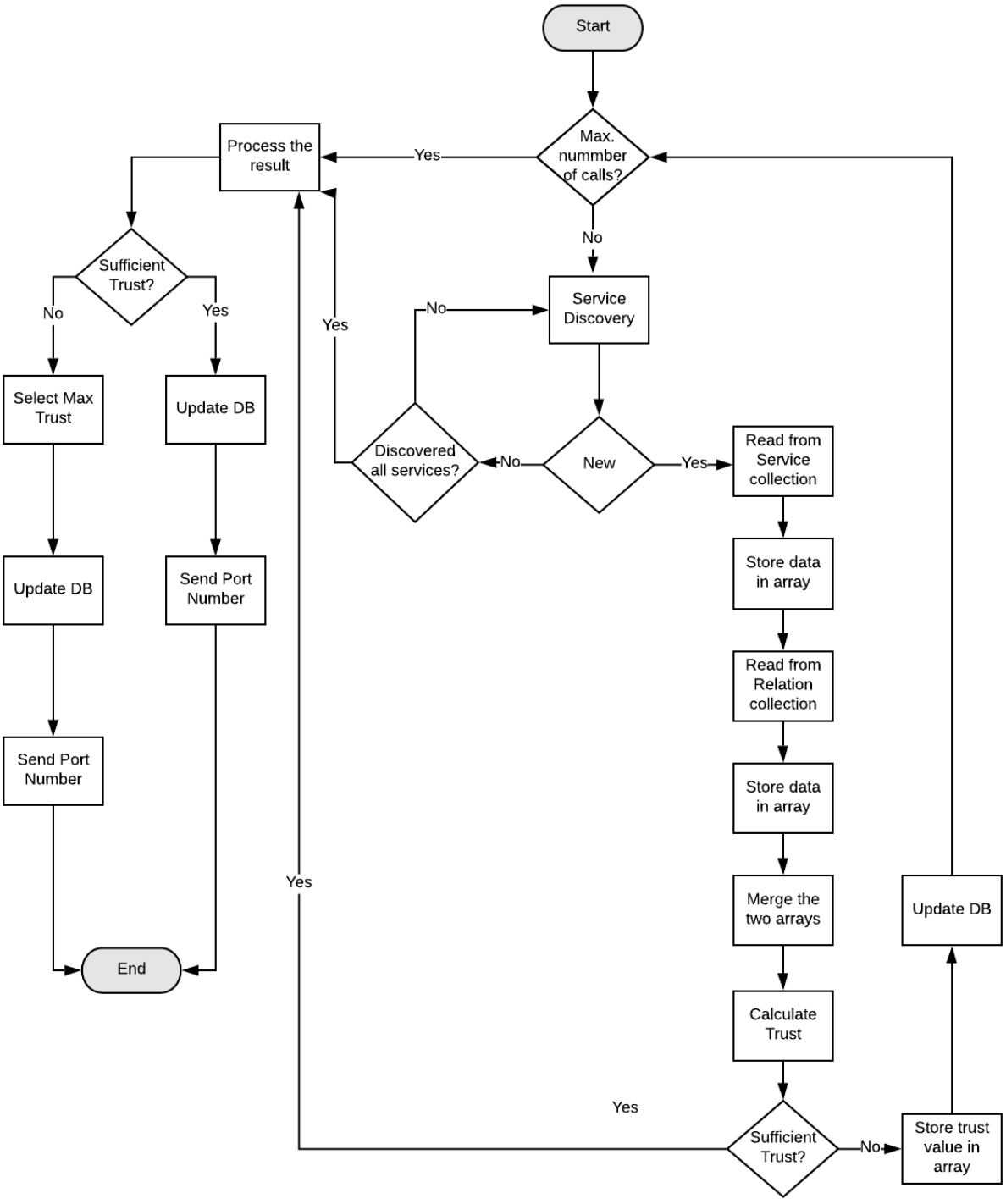

Figure 4.1: Content Trust workflow

Each service wants to initiate a call with another microservice will send the name of this service to the *Content Trust* microservice as well as its port number. Next, it waits for an answer that contains the port of the microservice that has accepted trust evaluation or has the highest evaluation of trust among all the possible options for this interaction.

When *Content Trust* microservice receives a request to calculate the trust evaluation of a service in the Blog. It will search the *Services* collection to find all the microservices that offer the required service. Once it gets a list containing microservices identifiers, it chooses one of them randomly.

```
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 does not get to be selected again in case 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 *Services* collection. The port of 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 *Services* 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 evaluations that other microservices have about the service in concern. This is done by querying the *Relations* 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 matches 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 *Relations* collection of 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 represents the result of last successful interaction the two involved microservices did. It also must store the index of the direct trust in the array of trusts that contains the data obtained from the *Relations* collection.

sdata now has two types of data, the first one is the data obtained from the *Services* collection, this data is now one object in the *sdata* array. The second one is the trust values obtained from the *Relations* 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 one object, since having many objects in the array will make it hard to reach each specific value when needed.

```
acc=sdata.reduce( function(acc, x) {
    for (var key in x) acc[key] = x[key];
    return acc;
}, {});
```

Listing 4.12: Organizing 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. This is flexible and developers could adjust the added value to the trust as it fits their specific application.

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. Additionally, the previous direct trust is also added to the final trust. This value will then be divided by the number of all the services that had influenced the content trust evaluation. This division will help to put the value of the final trust in the range 0 to 10. A final check is run to make sure that the value is indeed somewhere between 0 and 10. In case the value is not and it is bigger than 10, it will be set back to 10, and in case it is negative, it will be set to 0.

The very final step is to check if the final evaluation of trust is sufficient or not. This is done by comparing the final trust with the sensitivity of the requested microservice. The final evaluation of trust will be a value between zero and ten. If the value is 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 depends on the sensitivity level for each microservice. The sensitivity classification is as shown in table 4.1:

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 *Content Trust* microservice evaluates the trust of another microservice less than the sufficient level then *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 had their trust evaluated, then it will increase the number of failed interactions for all the microservices that failed to score sufficient trust. Finally, it will increase the number of successful interactions for the microservice that scored sufficient trust. Finally, it will store the current date in the last successful activity in the *Services* collection.

4.3 Micro frontends implementation

Micro frontends are small applications that, together, form 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 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, and it 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, then 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 isolated 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. 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 [50] Safari web browser from Apple does not support all web components features, while Microsoft Edge browser support is being implemented.

4.3.1 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. Each frontend is developed using ReactJS since it is a widely used JavaScript framework, but any of the apps can also be developed using any other JavaScript framework such as Angular or Vue.js. 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 [51].

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 [52].

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 helpful feature of Single-SPA library is the possibility of navigating from one micro frontend to another from within any micro frontend. This feature is used in the Navbar micro frontend where the user can navigate and move among different micro frontends. To move from one micro frontend to another the function *navigateToUrl(obj)* should be called. This function requires at least one parameter which is the URL of the destination micro front while it has no return value.

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 exchanging 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 behaviour 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 be sent from the *signin* micro frontend to the *newpost.*

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 *signin* application receives the JWT from the backend, 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 concept and the implemented Blog.

5 Evaluation

This chapter will provide an evaluation of the concept of content trust mechanism and how a full- stack microservices application operates with a content trust implementation. At first, the evaluation will assess how much the proposed concept respects the requirements of microservices architecture, and it will check the requirements against the backend and the frontend. Secondly, the content trust implementation will be evaluated, and tests will be run to analyse the output and understand how the content trust implementation operate in different situations.

5.1 Microservices and Micro Frontends Evaluation

For the evaluation of microservices and micro frontends, a static test is run to evaluate the requirements of microservices and micro frontends against the concept and implementation.

5.1.1 Size Evaluation

The first test is run to measure the size of the microservices and micro frontends of the Blog. This test will help to verify which microservices respects the size requirement and to give an overall picture of the sizes of the microservices. Each microservice and micro frontend is supposed to be small in size. To measure the size of each microservice the metric of Source Lines of Code (SLOC) is used. According to [53], SLOC is one of the most popular metrics used by researchers and developers to estimate the size of a piece of software. There are two types of SLOC, these are Physical SLOC (LOC) and the Logical SLOC (LLOC) [53]. The first one, Physical LOC is concerned with counting the lines of the source code that are neither blanks nor comments. The second type, Logical SLOC, is related to counting the statements in the source code. Listing 5.1 shows a simple for-loop.

```
    for(let i=0;i<10;i++) {
      process(i);
}
```

Listing 5.1: Simple for-loop

This loop contains four Logical SLOC since each statement is considered one SLOC, hence i=0 is the first statement, i<10 is the second, i++ is the third and process(i) is the fourth statement. Additionally, listing 5.1 contains three Physical LOC.

To measure the size of each microservice, both metrics will be used; the physical LOC as well as the Logical LOC. A tool called LocMetrics is used to help with measuring both Physical LOC and Logical LOC metrics. This tool takes a source code file as an input and produces both measures as output among other measures such as comment lines, blank lines and so on.

No.	Service	Physical LOC	Logical LOC
1	Comment	94	58
2	ContactUS	82	48
3	Duplication	57	45
4	Login	49	41
5	Post	169	130
6	Content Trust	356	272
7	Search	38	28
8	Usercheck	22	19
9	UserID	27	26
10	UserReg	87	51
11	validation	67	41
12	About (F)	48	25

13	contactUS (F)	133	44
14	Home (F)	649	272
15	Navbar (F)	57	26
16	New post (F)	121	44
17	Register (F)	118	37
18	SignIn (F)	108	46
19	SignOut (F)	47	23

Table 5.1: SLOC for microservices

Table 5.1 shows the Physical SLOC for each microservice as well as the Logical SLOC. The letter F that appears next to some services is to denote that this is a micro frontend.

Line of Code	**50**	**100**	**150**	**200**
PLOC	31.5 %	63.1 %	84.2 %	89.4 %
LLOC	73.6 %	84.2 %	89.4 %	89.4 %

Table 5.2: Percentage of LOC measures

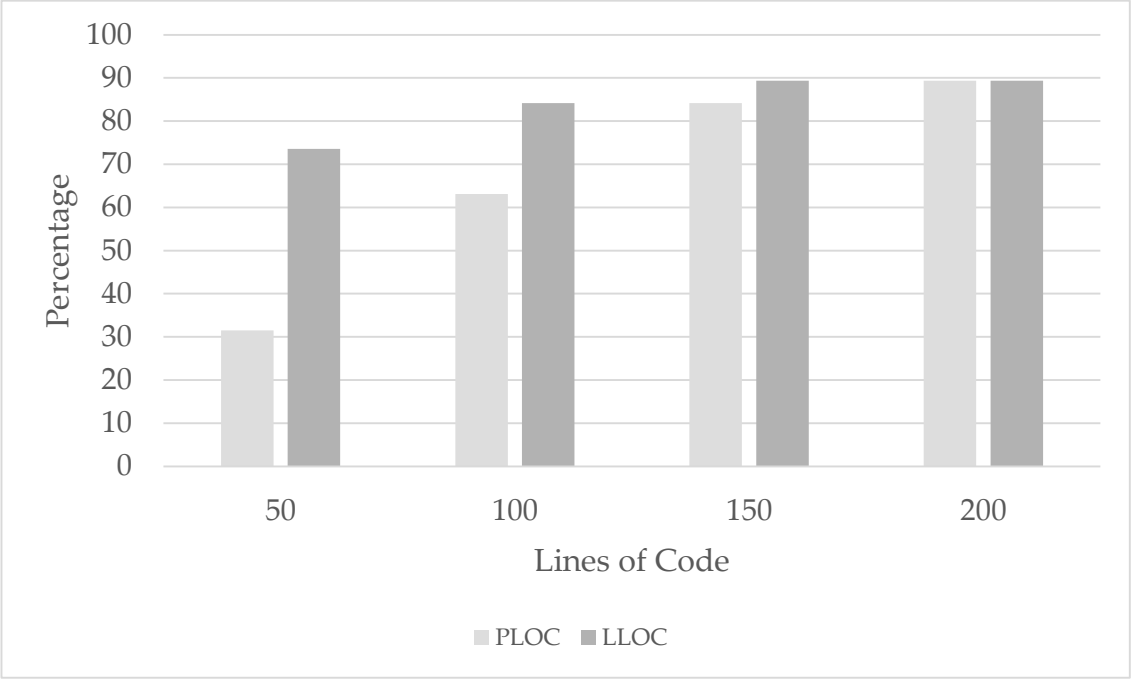

Figure 5.1: Microservices size

Table 5.2 shows the percentage of microservices different sizes. While figure 5.1 shows a bar chart representing the percentage of the sizes of the microservices according to their LOC metric. The diagram shows the results of the two metrics the Physical LOC and the Logical LOC. As table 5.2 shows that the majority of the services fall under 200 LOC for both metrics with a percentage of 89.4% and only 11.6% of all the services exceed 200 LOC for both metrics Physical LOC and Logical LOC.

5.1.2 Autonomy and Interface Evaluation

To test the autonomy of the services of the Blog, a dependency analysis is run for all the microservices and micro frontends. This analysis searches for any dependencies a service has, such as a database or files or anything that is used by this service.

To run the dependencies analysis, Madge tool is used to help analyse the dependencies between all the services in the Blog and to generate a dependency graph after the analysis is finished. Figure 5.2 shows the dependency graph for the micro frontends of the Blog. Madge shows any dependency from one module to another but it does not show HTTP calls as part of the dependencies for each micro frontend. Any arrow going from

one node to another is considered a dependency. Nodes that do not have arrows going from them to another nodes means that these nodes are dependency-free.

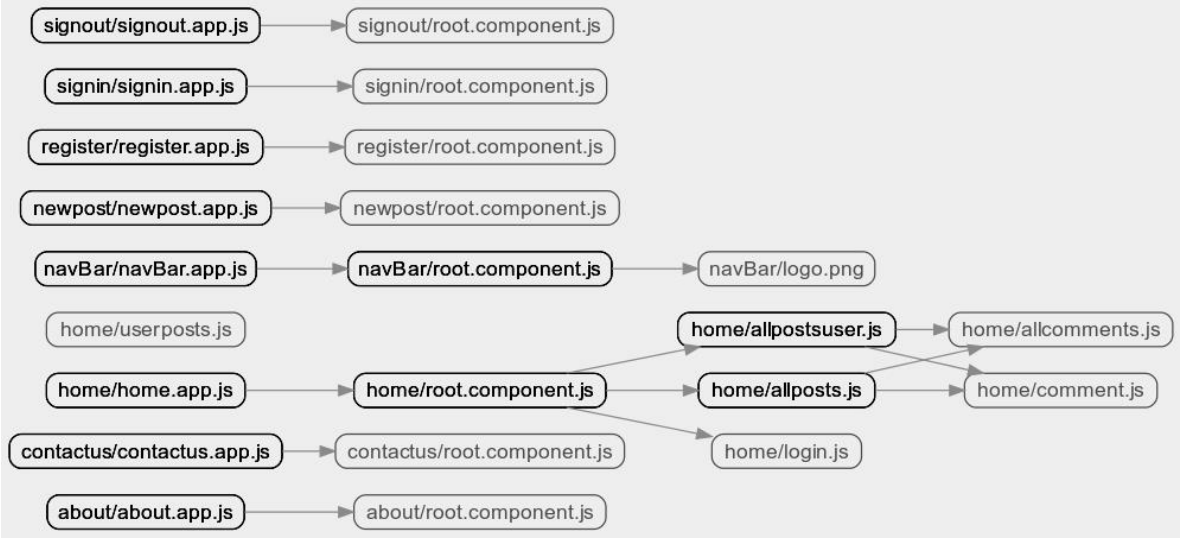

Figure 5.2: Micro frontends dependencies

Additionally, Madge was not able to detect any dependencies related to the Single-SPA library for any of the micro frontends, although all the micro frontends depend on it to operate in a micro frontends environment.

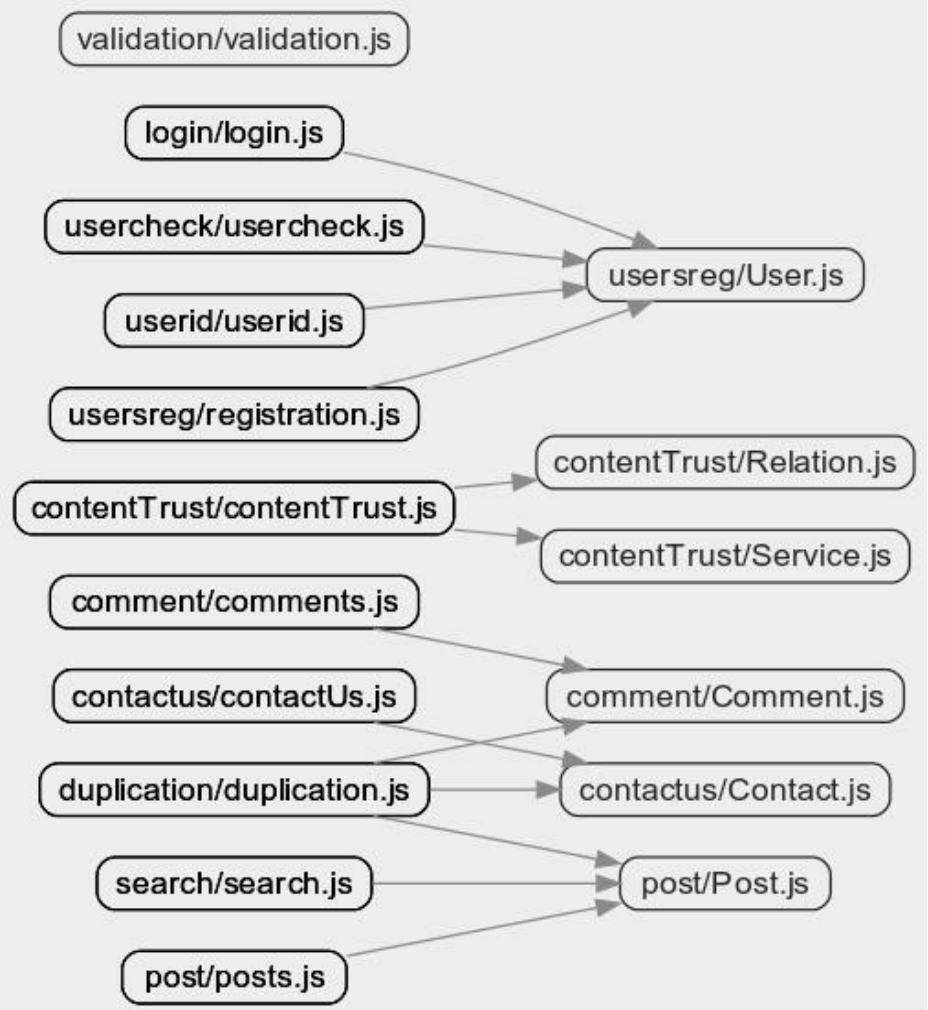

Figure 5.3: Microservices dependencies

Figure 5.3 shows the dependency graph for microservices, any node with an arrow going from it to another node means there is a dependency.

Table 5.2 shows the number of dependencies each service and frontend have as well as the number of APIs each service offers. The final dependencies count will take into account any HTTP calls between the services. When counting the number of dependencies each service has, its dependency to its own database is not considered, but any shared database will be considered. Additionally, any calls to the *Content Trust* microservice will not be considered either because it is not part of the original functionalities of each service.

No.	Service	No. of Dependencies	No. APIs
1	Comment	2	2
2	ContactUS	2	1
3	Duplication	3	1
4	Login	2	1
5	Post	3	9
6	Content Trust	0	2
7	Search	2	1
8	Usercheck	1	1
9	UserID	1	2
10	UserReg	2	1
11	validation	0	1
12	About (F)	1	0
13	contactUS (F)	2	0
14	Home (F)	3	0
15	Navbar (F)	2	0
16	New post (F)	2	0
17	Register (F)	2	0
18	SignIn (F)	2	0
19	SignOut (F)	1	0

Table 5.3: Dependencies count

Figure 5.3 illustrates the distribution of microservices over the existing dependencies. Over half of the services have two dependencies. 21% of the services have only one dependency. 10.5% of the services have zero dependencies while 15.78% of the services have 3 dependencies which is the highest recorded number of dependencies for any microservice or micro frontend. It is important to mention that each micro frontend must have at least one dependency which is Single-SPA because it has to implement several

functions of the Single-SPA library to be able to run in a micro fronts environment, as mentioned in the previous chapter.

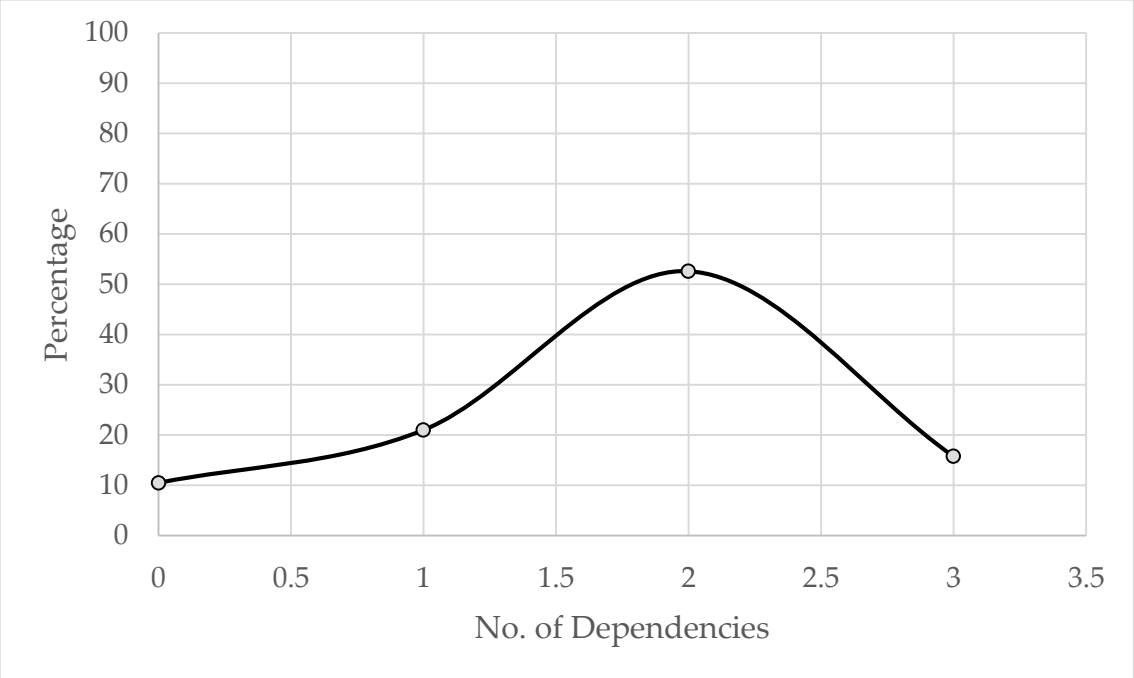

Figure 5.3: Dependencies of microservices and micro frontends

Additionally, table 5.3 shows how many APIs each microservice and micro frontend provide to their consumers. All the microservices offer at least one interface where the biggest number of APIs is offered by the *Post* service which is 9. On the other hand, micro frontends offer no interfaces to each other where Single-SPA library, until this moment, provides no channel or support for data exchange between micro frontends. The only communication between micro frontends which Single-SPA offers is sending events from one app to another. To help exchange data between the micro frontends, cookies were used as was explained in the fourth chapter.

5.2 Content trust evaluation

This section will be divided into two subsections. The first one will provide an evaluation for the content trust mechanism and will try to show how it operates in different context, while the second one will provide a performance evaluation of the content trust mechanism.

5.2.1 Content Trust Mechanism Evaluation

Carrying a content trust evaluation is not an easy task. One idea could be to create a study where participants have to select one software out of a few available ones, after which the results of the study can be compared with the results taken after running a content trust evaluation for the same set of software. Each one of the available software will have different properties. The properties come from the formula of the content trust evaluation, which has eight factors influencing the evaluation. The factors are:

- Trust of the development team
- Number of interactions
- Number of successful calls
- Number of failed interactions
- Sensitivity level of the service
- The last trust the service has from the calling service.
- The last trust the service has from other services
- Age of the microservice

For every factor, different values can be given. For simplicity, assuming only one of two values are possible, although each factor has more than two possibilities of values available. For example, the development team of each software can either be trustworthy or untrustworthy. Since eight factors are available and for each factor one of two values are possible, then the total number of available options is $2^8 = 256$

So even after simplifying the answers, each participant in the study has to select one option out of 256, which makes such a study difficult to conduct.

Another possibility is to run the content trust mechanism and analyse its outputs. Since many different factors influence the trust of each service, such as the number of interactions it has, the evaluation it has by other services and so on, it creates a need to run the content trust many number of times, while changing the different factors each time. But having at least 128 possible options makes it hard to run the test for all the available options at the same time. The testing environment cannot handle running all those services at the same time and this will cause it to crash.

A solution can be to run the content trust mechanism for a large number of times using a relatively small number of contesting microservices while changing the values given for each influencing factor in the content trust formula. At the end, analyse the outcomes to reach a conclusion regarding the content trust mechanism itself.

The following scenario is proposed: The development team of the Blog will implement some of its microservices and will also depend on third-parties microservices to fulfil other functionalities of the Blog. Thus, several microservices will be introduced to the Blog as if they were developed by third-parties.

In a real-world scenario, microservices could be developed by more than one third-party, and thus each development team is assigned a different weight when evaluating the trust for each service. For the situation in hand, there are ten different microservices and each one is developed by a different team.

A black box test is run to evaluate the content trust mechanism. Black box testing is concerned with the output of the test and does not focus on the internal structure of the software being tested [54]. A black box testing for the content trust mechanism will help to analyse the output of the content trust. Additionally, it will help to discover any problems in cases where content trust selects certain services over others where these services are less likely to be selected. Content trust will evaluate the trust of the different microservices in a situation where there are five services making the requests on one side and ten services are available to handle the incoming requests on the other side. Before any processing of the requests and any exchange of data, a trust evaluation of the microservices will be calculated. Each time a trust evaluation is carried out, one of the ten microservices will be selected by the content trust mechanism to handle the coming request. The services initiating the call are developed by the original development team and thus they are already trusted.

The evaluation environment is a PC with 4 GB of RAM memory, i3-CPU with 1.70 GHz, x64-based processor with 64-bit Windows 10 Operating System. All the ten services are assumed to be developed by third-parties. Their development teams' trusts are given randomly. The test is run where services are added gradually to the system to simulate a real-world scenario. In such case, the system is growing and services are added accordingly. At the first 20 calls, four services are available in the system, and then after each 5 calls, one service is added. By the call number 50, all ten services are operating in the system. The ten services are being called by five trusted services. Each service will do 20 of the 100 calls and each service will make one call at a time. The total number of the operating services in the test is 15 different services. The functionalities of each service are not of any importance to the test and will not be discussed.

Name	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**	**S10**
20 calls	8	0	8	4						
25 calls	3	0	1	0	1					
30 calls	0	0	1	1	2	1				
35 calls	1	0	0	0	1	2	1			
40 calls	0	0	1	0	2	1	1	0		
45 calls	1	0	2	1	0	0	0	0	1	
50 calls	2	0	1	0	0	0	0	1	1	0
100 calls	9	0	13	2	4	5	7	5	5	0

Table 5.4: Number of services responses

Table 5.4 presents the number of responses each service had after 100 calls. The first raw in the table shows the names of the services where S1 represents Service1, S2 represents Service2 and so on. After the first raw, each raw represents how many responses each service made for the coming calls. The number of the coming calls is given in the first column of each raw.

Two services never achieved enough trust evaluation. One of them was operating from the beginning of the test. They both were assigned very low development team trust.

Two other services dominated the responses to the coming calls and their responses were 27% and 24% respectively. Both of them were assigned the two highest development team trust and were operating from the beginning. The rest of the services responded to 49% of the calls. The previous test shows that the trust of the development teams is very important at the early stage of the operating of the Content Trust mechanism.

Now all the services are operating in the system. Another test is run for 100 calls with the current values and properties each service is holding.

Name	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**	**S10**
100 calls	31	0	33	5	6	7	9	7	2	0

Table 5.5: Number of services responses

Table 5.5 shows the number of responses each service had after having its content trust evaluated by *Content Trust* microservice after 100 calls. The two services that dominated the first 100 calls, dominated the second 100 calls with 33% and 31% respectively. The rest of the services responded to 36% of the calls except two services that never achieved enough trust levels. These two services were not able to achieve enough trust in the first 100 calls either.

It can be seen from the two tables that the longer a service operates, the higher its chance of achieving enough trust since the number of its interactions and its successful as well as failed interactions are taken into account. On the other hand, when a service fails several times, its chances of achieving enough trust becomes less with every failed interaction.

Since the two most high achieving services were assigned high development teams trust value, in the next run they will be assigned low development team trust values, while the rest of the properties for all the services are left as they are. All the ten services will be contesting to handle the requests that are coming from five trusted services. 100 calls will be initiated.

Name	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**	**S10**
100 calls	18	0	15	12	13	12	14	10	6	0

Table 5.6: Number of services responses

Table 5.6 shows how many responses each service had. The two services that failed in the first two runs to achieve enough trust also failed in the third run. The rest of the services had relatively close responding rates. Downgrading the trust value for the development teams of the two most achieving services harmed their trust evaluation and they only scored 18% and 14% respectively. Nevertheless, they scored a high percentage of responses overall.

The last results confirm that the history of each service has an influence on its future chances. It is also apparent that having a big number of successful interactions helped in the last run although their development trust team were not the best. The trust for the development team of a service plays a role in evaluating its trust, and it has a more important role at the beginning of operation since it is one of the few available pieces of information at such an early stage. The longer a service operates the more information there is available about it. As a result, the influence of the development team trust decreases, while other factors play bigger roles such as the evaluation given by other services, how many requests a service has answered, how long a service has been operating in the system and so on.

In the long run the content trust mechanism gives equal importance to each factor influencing the content trust evaluation. But on the other hand, when services run for a long time their previous evaluations are enforced. This means that if a service starts with low team development trust, then its chances of having enough trust may not increase after running for a long time. However, the longer it runs the higher its chance of having a trust evaluation that is below the requirement.

5.2.2 Content Trust Performance Analysis

To analyse the performance of content trust mechanism a test is run while recording the response time after each response received from the *Content Trust* microservice. At first, only four microservices are available to handle the incoming requests. After each 20 calls, two new microservices are added to the system until the total number of available services is 40. This helps in understanding how much effect the number of microservices in the system has on the performance of the content trust mechanism.

Table 5.7 shows the response times for the *Content Trust* microservice. The number of available services appears to the left of the respective response time.

No. of Services	Response Time	No. of Services	Response Time
4	0.19	**24**	0.192
6	0.15	**26**	0.196
8	0.199	**28**	0.189
10	0.201	**30**	0.22
12	0.191	**32**	0.21
14	0.192	**34**	0.21
16	0.1713	**36**	0.187
18	0.177	**38**	0.2
20	0.168	**40**	0.14
22	0.194		

Table 5.7: Response time

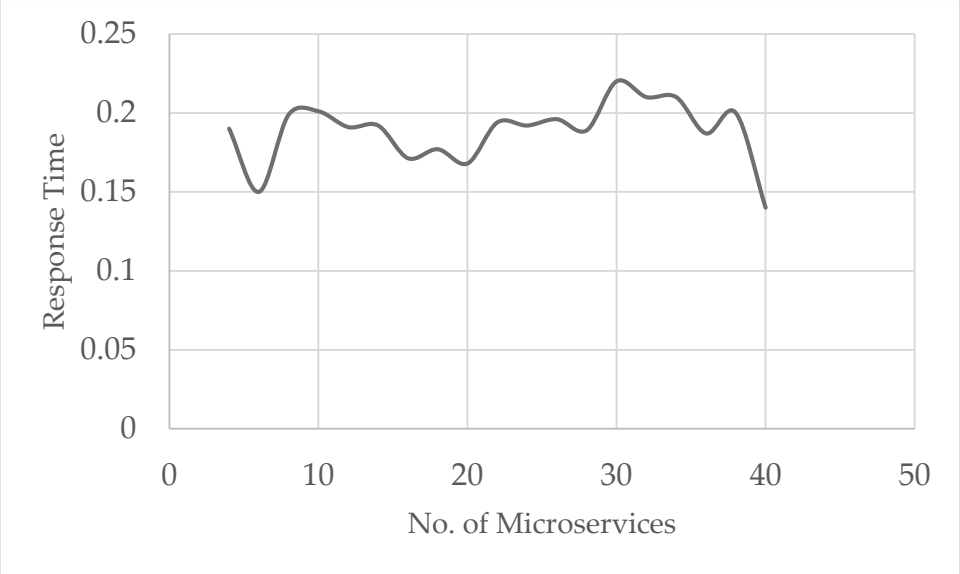

Figure 5.4: Response Time

Figure 5.4 illustrates the response time of the *Content Trust* microservice at any given time in relation to the number of available microservices. The graph shows that with each increase in the number of available microservices the response time of the *Content Trust* microservice did not increase accordingly. This means that the number of available services and the response time of *Content Trust* microservice are not directly propor-

tional. Thus, the performance of the Content Trust microservice was not affected negatively with the increase of the number of available services that should have its trust evaluated after each call.

Table 5.7 also shows that when adding a content trust mechanism to the system a delay should be expected. The delay that is available on the test machine did not exceed 0.22ms for as many available microservices as 40. This delay can, of course, be bigger if the system is run on a machine less powerful than the test machine.

6 Conclusion

This thesis has introduced the concept of content trust of web resources to the microservice architecture environment. The motivation behind it is to make use of the concept of content trust of web resources and transform it from being a relationship between an end user (Human) and a web resource into a relationship between applications without human intervention.

The first chapter introduced both microservices architecture and content trust while describing how the connection between the two can exist and be helpful. The second chapter derived the main requirements that should exist for microservice-based applications and a content trust implementation. It went on to present a literature review for microservice architecture and content trust, and finally an analysis for both.

Next, the concept behind employing content trust of web resources in a microservice-based web application was presented. It has described how a web application can be developed to become a full-stack microservices application, in which the backend uses microservices architecture and the frontend uses micro frontends. From there, two possible concepts were presented on how content trust can be used in such an application. The advantages and disadvantages of both concepts were laid out and it was explained why one concept is more favourable for this context.

After presenting the concept, the developed algorithm of content trust and its workflow was shown. The algorithm was developed to be flexible and can be easily adjusted to different situations. Moreover, this chapter shows how the frontend and the backend are both developed based on the microservice architecture. It also demonstrated how content trust implementation is used in this environment.

The fifth chapter illustrated the tests and evaluations run to test the application. A black-box test was run to understand the behaviour of the content trust implementation. With every test the inputs were modified and adjusted to test the content trust from different angles. Additionally, a static test was run to check if the application adhered to the requirements of microservices architecture.

The developed Blog in this thesis is not considered a big web application, and as a result, the research was not able to show what are the effects of using content trust implementation in an environment that is composed of hundreds or more microservices. Such shortcomings are mostly because of the timeframe that is required to develop an application that has hundreds of microservices while at the same time develop the content trust mechanism itself.

Understanding what the effects of using content trust in applications that have hundreds of microservices can be solved in future work. The presented implementation of content trust can be employed to simulate its usage in a complex web application. Additionally, verifying the body of the requests made from one service to another can be introduced. Content trust implementation can use Artificial Intelligence techniques to learn about harmful requests and be able to classify microservices accordingly.

Bibliography

[1] Microsoft Patterns & Practices Team, *Application Architecture Guide, 2nd Edition (Patterns & Practices),* Microsoft Press, 2009.

[2] Software Architecture Patterns by Mark Richards, Publisher: O'Reilly Media, Inc.2015

[3] K. Miika, N. Mäkitalo, and T. Mikkonen, "Challenges when moving from monolith to microservice architecture," in *International Conference on Web Engineering*, 2017, pp. 32-47.

[4] P. Di Francesco, P. Lago and I. Malavolta, "Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption, " *in International Conference on Software Architecture (ICSA),* 2017.

[5] Y. Gil and D. Artz, "Towards content trust of web resources," *Journal of SSRN Electronic Journal,* 2007. doi: 10.2139/ssrn.3199370.

[6] P. D. Francesco, I. Malavolta and P. Lago, "Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption," *2017 IEEE International Conference on Software Architecture (ICSA),* Gothenburg, 2017, pp. 21-30.

[7] S. Newman, *Building microservices,* Sebastopol, O'Reilly Media,2015.

[8] C. Pahl and P. Jamshidi, "Microservices: A Systematic Mapping Study," in *Proceedings of the 6th International Conference on Cloud Computing and Services Science,* 2016, pp. 137-146.

[9] S. Daya, N. Van D., Kameswara Eati, C. M Ferreira, D. Glozic, V. Gucer, M. Gupta, S. Joshi, V. Lampkin, and M. Martins, *Microservices from Theory to Practice*, IBM Redbooks, 2015.

[10] L. Zvirblis, 'Securing Information Flow in Loosely-Coupled Systems', Norwegian University of Science and Technology, 2011.

[11] C. Richardson and F. Smith, *Microservices: From Design to Deployment*, NGINX, 2016 microservices design and deploying

[12] M. Richards, Microservices vs. Service-Oriented Architecture, O'Reilly Media, Inc., 2016.

[13] T. Söderlund, "Micro frontends—a microservice approach to front-end web development", Medium, 2018. [Online]. Available: https://medium.com/@tomsoderlund/micro-frontends-a-microservice-approach-to-front-end-web-development-f325ebdadc16. [Accessed: 02- Jan- 2019].

[14] S. De Santis, L. Florez, D. V Nguyen, and E. Rosa, *Evolve the Monolith to Microservices with Java and Node,* IBM redbooks, 2016

[15] E. Wolff, *Microservices Flexible Software Architecture.* Addison-Wesley Professional, 2016.

[16] T. Erl, *SOA Principles of Service Design,* Prentice, 2008

[17] Ł. Kyć "Independent micro frontends with Single SPA library," Pragmatists, 30-Jul-2018. [Online]. Available: https://blog.pragmatists.com/independent-micro-frontends-with-single-spa-library-a829012dc5be. [Accessed: 30-Dec-2018].

[18] A. Kothari, "What is a micro frontend? | Packt Hub", Packt Hub, 2019. [Online]. Available: https://hub.packtpub.com/what-micro-frontend/. [Accessed: 17- Nov- 2018].

[19] G. Benedict, "What is Micro Frontend and How it Provide Benefits to the Startups?", Techuz Blog, 2019. [Online]. Available: https://www.techuz.com/blog/what-is-micro-frontend-and-how-it-provide-benefits-to-the-startups/. [Accessed: 14- Nov- 2018].

[20] B. Johnson, "Exploring micro-frontends", Medium, 2019. [Online]. Available: https://medium.com/@benjamin.d.johnson/exploring-micro-frontends-87a120b3f71c. [Accessed: 01- Dec- 2018].

[21] D. Namiot and M. Sneps-Sneppe, "On Micro-services Architecture," *International Journal of Open Information Technologies,* vol. 2, no. 9, 2014. ISSN: 2307-8162

[22] S. P. Marsh. "Formalising Trust as a Computational Concept", "Doctor of Philosophy", University of Stirling, 1994.

[23] D. Harrison McKnightNorman L. Chervany, "The Meanings of Trust,"1996

[24] S. Singh, S. Bawa, "A Privacy, Trust and Policy based Authorization Framework for Services in Distributed Environments," *international journal of computer science,* vol., 2 no. 2, pp. 85-88, 2007. ISSN: 1306-4428

[25] G., J. J., "The Development of Trust, Influence, and Expectations," In *Athos, A. G., Interpersonal Behavior: Communication and Understanding in Relationships,* Prentice-Hall, 1978, pp. 290-303.

[26] J. David Lewis and Andrew J. Weigert, "Trust as a Social Reality," *Journal of Social Forces,* vol. 976, no.63, 1985. doi: 10.2307/2578601.

[27] A. Macko, M. Malawski, and T. Tyszka, "Belief in others' trustworthiness and trusting behavior," *Polish Psychological Bulletin,* vol.45, no.1, 2014. doi: 10.2478/ppb-2014-0007

[28] D. H. McKnighta, V. Choudhuryb, and C. Kacmarc "The impact of initial consumer trust on intentions to transact with a web site: a trust building model," *Inf. Sys,* vol.11, 2002. doi: 10.1016/S0963-8687(02)00020-3

[29] D. H. McKnight, D. Harrison, and N. L. Chervany, "Trust and distrust definitions: One bite at a time," In *Trust in Cyber-societies,* 2001, pp. 27-54.

[30] L. Teun, and J. M. Schraagen. "Factual accuracy and trust in information: The role of expertise." *Journal of the American Society for Information Science and Technology,* vol. 62, no. 7, 2011. Doi: 10.1002/asi.21545.

[31] R. Pennington, H. D. Wilcox, and V. Grover, "The Role of System Trust in Business-to-Consumer Transactions," Management Information Systems, vol. 20, 2003. doi: 10.1080/07421222.2003.11045777

[32] S. G. Goto "To trust or not to trust: Situational and dispositional determinants," *Social Behavior and Personality,* vol.24, no.2, 1996. doi: 10.2224/sbp.1996.24.2.119

[33] L. Dean. "Representations of the concept of trust in the literature of Library and Information Studies," 2013.

[34] S. Mohammed, R. Lakshminarayanan, R. Ramalingam, "Password-based Authentication in Computer Security: Why is it still there?," *Standard International Journals (The SIJ),* vol. 5, no. 2, pp 33-34, 2017. ISSN: 2321-2381

[35] A. Conklin, G. Dietrich, D. B. Walz, "Password-Based Authentication: A System Perspective," in *Proceedings of the Hawaii International Conference on System Sciences*, 2004, pp. 1-10.

[36] M. S. Merkow and J. Breithaupt, *Information Security: Principles and Practices*. Indianapolis, Prentice Hall, 2005.

[37] A. Jøsang, R. Ismail, and C. Boyd, "A survey of trust and reputation systems for online service provision," *Decision Support Systems*, vol. 43, no. 2, pp. 618–644, 2007

[38] "Reputation-Based Trust Management Systems and their Applicability to Grids" [Online]. Available: https://www.researchgate.net/publication/30410275_Reputation-Based_Trust_Management_Systems_and_their_Applicability_to_Grids. [Accessed: 09-Nov-2018].

[39] P. Bonatti, C. Duma, D. Olmedilla, and N. Shahmehri, An Integration of Reputation-based and Policy-based Trust Management, 2005.

[40] Vivekananth.P, "A Behavior Based Trust Model for Grid Security," vol. 5, no. 6, 2010. doi: 10.5120/922-1300.

[41] L. Kutvonen and S. Ruohomaa, "Behavioural evaluation of reputation-based trust systems," in *International IFIP Working Conference on Enterprise Interoperability*, 2013, pp. 158-171.

[42] V. Shmatikov and C. Talcott, "Reputation-Based Trust Management," *Computer Security - Special issue on WITS'03*, vol. 31. no. 1, 2005. doi: 10.3233/JCS-2005-13107

[43] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin and L. Safina, *Microservices: yesterday, today, and tomorrow*, Springer, Cham, 2017.

[44] "Content trust in Docker," Docker Documentation. [Online]. Available: https://docs.docker.com/engine/security/trust/content_trust/. [Accessed: 09-Nov-2018].

[45] R.T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures", Doctor of Philosophy, University of California, 2000.

[46] About | Node.js", Node.js, 2018. [Online]. Available: https://nodejs.org/en/about/. [Accessed: 29- Dec- 2018].

[47] Axios", npm. [Online]. Available: https://www.npmjs.com/package/axios. [Accessed: 29- Dec- 2018].

[48] Introduction to MongoDB - MongoDB Manual. [Online]. Available: https://docs.mongodb.com/manual/introduction/. [Accessed: 11-Nov-2018].

[49] S. E. Peyrott, *The JWT Handbook,* Auth0 Inc, 2017.

[50] "Web Components", MDN Web Docs. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Web_Components#Browser_compatibility. [Accessed: 10- Dec- 2018].

[51] "Getting Started with single-spa · single-spa", Single-spa.js.org. [Online]. Available: https://single-spa.js.org/docs/getting-started-overview.html. [Accessed: 08- Mar- 2019].

[52] "Building single-spa applications · single-spa", Single-spa.js.org. [Online]. Available: https://single-spa.js.org/docs/building-applications.html. [Accessed: 01- Mar- 2019].

[53] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm, *A SLOC Counting Standard*, California, University of Southern California, 2019.

[54] S. Nidhra, “Black Box and White Box Testing Techniques - A Literature Review,” *International Journal of Embedded Systems and Applications*, vol.2, no. 29, 2012. doi: 10.5121/ijesa.2012.2204.

Appendix A

List of the microservices and micro frontends of the Blog:

Name	Description
Comments	Help users submits comments for a certain post in the Blog
ContactUs	Submit messages to the admins of the Blog
Content Trust	Evaluate the content trust between the microservices
Login	Logs in registered users of the Blog
Post	Handles posts related functionalities
Search	Searches the posts of the Blog for certain terms
UserCheck	Checks if a user is already registered in the database
UserID	Brings User's information based on his/her ID
Registration	Registers new users in the DB
Validation	Validates users' input in the Blog forms
Duplication	Checks for duplicated data
About (f)	Shows information about the Blog
ContactUs (f)	App for sending messages to the admins of the Blog
Navbar (f)	Navigation app of the Blog
Home (f)	Shows posts of the Blog
Signin (f)	Helps users signin
Signout (f)	Helps users sign out
NewPost (f)	For making new posts
Register (f)	Helps users create new accounts