**Current Situation**

Microservices is still a new concept, although some companies have already migrated to the Microservices architecture. There are many companies facing many issues that could be solved by this new architecture but still are hesitant in moving forward and migrating their application to the Microservices architecture.

This hesitation doesn’t come from one or two reasons but the idea of refactoring an existing application into a new one built using another architecture is not easy and brings with it many challenges that are still waiting to be addressed. The process of migration itself is still not clear.

How can an application be divided into smaller parts? What norms and standards should be followed when making a decision about such division? How big can be any microservice? How can Microservices interact with each other and exchange data? How can those microservices be merged together? What if the Frontend is becoming also complex and needs to be divided into Micro Frontends? How can Micro Frontends and microservices work together? How can microfrontends work together and send and receive data? How should a microfrontends interact with a microservice? How can microservices trust each other? What if the developers had to use ready-made microservices solutions? How can each microservice trust the behavior of a ready-made microservice? How can sensitive data such as credentials exchanged securely among those small parts of the application? And many more.

Questions are many and numerous when it comes to migrating an existing application into a Microservices architecture, so many questions are waiting to be answered. The architecture itself is new and a great deal of research is needed to help those who want to move their application into a Microservice architecture or even if developers want to build their applications from scratch based on the Microservices architecture.

There are still no standard definition of Microservice architecture and there’s no clear guideline of how an application based on Microservices should be built. Over the last few years some characteristics for a microservices-based application have been developed and some general basic outlines are now commonly used.

* A Microservices-based application should consist of more than one component: Unlike monolithic-based applications, a system built using Microservices architecture should be composed of multiple components, each component is self-contained. This way the application can be changed, updated and modified whenever is needed since 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. But one should not forget that there are complications and challenges regarding interaction and securities between microservices.
* Decentralization: An application based on Microservice architecture is built out of many different components and each has its role, but the application in total is not a one unit and there’s no on big unit moderating the communications between the microservices.
* Different technology stack: The development cycle of a Microservices-based application involves having different teams working on different microservices. Since each microservice is a self-contained component, each team can then choose the most suitable development technologies and tools that are most suitable for their own microservice.

**Motivation**

The idea behind creating a web application from smaller application is still very new. The current architecture that’s used heavily in building web application is composed of layers built on top of each other. Each layer is responsible for a field or a big part of the application. How many layers an application can have is still dynamic and up to the developers to decide. This architecture is called monolithic architecture. It usually consists of three layers on top of each other’s, the User Interface, then underneath comes the Logic Layer and finally the Backend [x]. Moreover some applications could end up having their logic layers divided further into more layers.

Although monolithic web application are divided into layers but the application is still very tightly coupled. There’s a great amount of connections between the layers, and during the development, developers would divide the Application into two parts: A frontend and a backend. With each part different tools and technologies are used. This results in having the development divided into only two parts where the logic layer is merged or shifted to be sometimes considered as part of the backend. Such scenario would result in a highly linked system. The two parts or layers of the application will have many interconnections. Although there’s a big possibility in having each layer developed by a different toolset and even a different development team, yet having a failure in one layer will drive the whole system to shut down and break completely.

As a consequence, the system, of course, will be hard to operate, maintain and update. Each problem in the backend will make the system useless, where no information could be processed or stored in the database. On the other hand, each problem in the frontend will make the application unusable, and data can’t flow from the frontend to the backend.

Finding the problem when the system fails is a big challenge. The first step is the easiest one, developers first need to decide if the problem comes from the frontend or the backend. Once this has been identified, the real challenge starts. Each part is one big unit and in order to find why the system fails many tests should be run and sometimes thousands of lines of codes should be reviewed to find and fix the problem. Sometimes, this situation could lead to even a worse one. In some cases fixing the issue is not possible until some modification is performed to suit the new changes. This lead to the problems of updating the system.

Updating a monolithic-based application is another big problem faces developers and owners. The application behaves as a one big unit, when needing new features or updates to fix newly discovered issues, developers could find themselves trapped between not being able to achieve the wanted updates and fixes as much as they’re supposed to, and between having to perform huge system modifications and changes to accommodate the new desired features or to be able to add new pieces to the application which could help to solve the existing problems.

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

Under Microservices, the application is not just one big unit, nor is it a three-layer architecture. The system is a groups of smaller systems. Each small system is a self-contained unit. Those units can be developed and deployed independently. Moreover, each unit can be developed by a different development team, hence creating a team of small number of developers a possible choice.

The concept of Microservices is further projected into the other side of the web application. Microservices is not just concerned with the backend side of the system, but it can be also applied to the frontend as well. When microservices concept is applied to the frontend it is called Micro frontends[x].

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

The point of creating the frontend based on the Micro frontends architecture is to make the application more flexible and more adaptable. A frontend could start having problems once the application requires more features. In such case, developers might need to apply many alternations to help the system be able to accept the newly added features. When a problem is discovered, sometimes big revisions and improvements are needed to get the system to the desired working state.

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

Microservices and Micro frontends architecture helps to solve many pressing problems for the development of web application. At the same time such architecture brings its own challenges and problems.

Putting all the small parts together is not an easy task. One question would be: How can Microservices interact with each other? Or how can Microservices exchange sensitive data among each other? Once every essential Microservice is created they need to be grouped together to form the final system. What technologies should be used to group those services? When thinking about the whole system, one need to think also about the frontend. How can the frontend interact with the Microservices? There are different ways to try to answer such question. Would the best solution be by creating Micro frontends for each Microservice when needed? But then this solution brings its own set of questions that would need to be answered.

**Problem**

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

One case would be that, the Web application already exists using monolithic architecture where there is a need to migrate it into Microservices architecture for many reasons such as:

* The application is not scalable anymore, such that new features are needed but scaling the application to fit those new features in is not possible without making big alternations to the application itself.
* The application has some serious problems and developers are not able to find suitable solutions for those problems without creating many additional parts or performing huge changes to the base code.
* Some parts of the application fails constantly causing the whole system to fail, in this case developers are not able to isolate the failing parts such that whenever a failure happens the system continues to run while the problem is being fixed.
* There can be, of course, many other reasons why a web application could perform better and be more scalable and easier to maintain and run when built using Microservices architecture.

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

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

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

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

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

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

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

Obviously, the division here is concerned with the functionality of each part of the website not how those parts are outlined and organized in the website.

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

* NGINX as reverse proxy
* Varnish with ESI module enabled

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

* Single SPA library
* Web Components
* iFrame

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

To give an example, iFrame which stands for inline frame works in a way that enables developers to include an HTML document inside another one. iFrame is not a new technology and it is known since the early days of HTML. When developers decide to use iFrame to bring together their micro frontends then they will have big challenges when the need communications among their micro frontends. iFrame isolates each part and in this case there’ll be a need for finding a way to overcome such isolation.

The problem in the case of micro frontends is which technologies to use? And whether or not such technologies will need additional development to satisfy the needs of the application in hands.

Moreover, security concerns are a big challenge under Microservices architecture. How would Microservices exchange particular sensitive information such as passwords or bank details? What if Microservices were not all developed inside the same company? Such scenario could rise when small companies want to build their applications using Microservices architecture. In this case, when having a small team of available developers, one could think of using ready-made solutions. Developers could use already written Microservices to save time and money. But doing this imposes more security concerns. In this situation how could Microservices trust each other? How could the existing Microservices trust the newly added ready-made solutions? The kind of trust discussed here is related to the behavior of each Microservice. After all, malicious or harmful Microservices could hide their true intentions by expressing different behavior while a harmful one is practiced behind the scenes.

This is not the only concern here, since microservices have to authenticate themselves to each other. Each microservice must make sure that it is handing over the data to the right microservice not another one. Once authentication is performed then the actual behavior of each microservice is still in question. There should be a way to help microservices trust each other without having a human intervention. When the application is getting bigger and embracing hundreds of microservices which in turn might also be depending on other microservices to run then check the behavior for each one of them but the developers could end up being an endless task. Developers could start checking the microservices they adapted into their applications but then shortly find themselves checking microservices that are used by the microservices they used and so keep moving backwards in the string of microservices.

Such situation is not what the solution should be, on the contrary, each microservice should be doing its own task of verifying its behavior with other microservices. When one microservice fails in doing so, then it is the decision of the developer to either cancel using it or to intervene and check what is causing the problem.

**Objective**

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

Microservice architecture is still growing in the sense that it has not yet a well-known agreed upon definition. There is still no common understanding of how small each microservice should be. Some resources consider looking at the source code and making sure that it doesn’t exceed hundreds of lines. Some researchers think that the development time for one microservices should not exceed two weeks of work. Another view point is that developers should keep dividing the tasks they have until no further division can be applied. This view point argue that each microservice should be concerned with one task, and one task only. Once the task can’t be divided anymore, developers should stop and assign the task to a microservice [x].

Not everyone agrees with the above suggestions. For example, having the microservices as small as possible means that the application will end up having many small apps working together. The more moving parts an application will have the more overhead there can be for developers to put everything together. On the other hand, the more small apps an application have the easier it is to solve a problem.

When the application have many small apps forming the final product, developers will have better circumstances when problems and failures happen. Once a problem is detected, developers could then easily isolate the broken part of the application, in this case, one or more small apps. Being able to isolate the dysfunctional parts will give a better chance for the whole system to keep running and offering its services to clients while the problem is being fixed.

Another arguable concept is the method of communication between microservices. Microservices are supposed to be self-contained autonomous small apps [x] but at the same time they should offer an interface for their clients to communicate with them. Their clients are often other microservices trying to send them data or request data from them. This scenario contradicts with having microservices as independent small apps. Should developers strive to make each microservice as independent as possible or should they try to separate microservices from each other while offering a good mechanism for intercommunication?

Some microservices will, of course, need data from other sources. Maybe some of those microservices is only responsible of receiving data, processing it and moving it forward. For such microservices they should be able to communicate with other microservices to achieve their tasks but they should not need to communicate with other microservices to help them with the processing of the data. The point of argue here is whether or not processing of the data should be done completely inside on microservice and if one microservice can’t handle the processing by itself. Should it be divided into more microservices where the intercommunication will help with the processing? Or should it be combined with other microservices to handle the task inside it completely?