Concept

Application consisting of many microservices.

Microservices have interfaces.

They interact with each other using REST

One database for all the services

Each microservice can evaluate the trust value with another microservice

Such evaluation based on many factors inspired from the web content trust

The concept of microservices

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

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

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

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

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

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

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

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

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

* Small
* Independent
* Has an interface
* Easy to deploy
* Reusable

Moreover the application has overall features related to microservices as:

* Scalable
* Resilience
* Loosely-coupled

**Small**

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

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

**Independent**

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

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

**Has an interface**

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

**Reusable**

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

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

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

**Scalable**

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

**Resilience**

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

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

**Loosely-coupled**

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

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

The concept of content trust

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

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

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

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

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

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

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

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

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

**Identity verification**

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

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

**Services classifications**

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

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

**Service sensitivity measure**

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

**Past experience**

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

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

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

**Number of interacting services**

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

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

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

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

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

**Check the age of microservices**

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

**Passing of trust**

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

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

**Developers of the service**

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

**Limited services**

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

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

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

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