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

Implementation of microservices

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

Furthermore, the communication between microservices is secured using content trust. This approach helps microservices trust each other and hence enable them to exchange information safely.

The following services implements the functionality of the backend:

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

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

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

Moreover, content trust implementation will make better evaluation of trust when it has more than one option. For example, if the for one specific task there are two services handling this task, then any other service that wants to communicate with one of those two services will have to accept the one that has the highest evaluation of trust. In this case the difference between the two services may not be that big. On the other hand, if there are four or five services instead of two then the difference in trust evaluation will be bigger. As a result, other services can make better judgments when they have more options.

The following pages will discuss in details the role of each service and the underlying implementation of each one.

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

REST could simply be described in the following scenario where it involves a client and a server. On one side, the server is running a resource (files, database records…etc. stored in the server). On the other side, a client that requests these resource. REST stands for Representational State Transfer. The client asks for a representation of the resource, basically the data. In the case of the implemented Blog, the data mostly represents posts made by the users. The client doesn’t care about how the data is stored on the server side (what technologies are used…etc.). What it receives from the server is a representational state of the data. JavaScript Object Notation (JSON) is a common format for resource representation in REST architecture [47 wrong I show not add it]

C:\Users\Abid\Downloads\rest.png

Figure 3.1 shows this scenario with the time line. The user interacts with the Blog via the micro frontends. Once the user requests to read a post on the Blog, the frontend sends this request to the backend. The backend is composed of many microservices. One of them is responsible for handling requests related to the posts. This microservice will receive the request, then initiate the connection with the database and request this specific post from it. The database will answer the request and sends the post. The post will be sent in the same format as it is stored in the database. The responsible microservice will receive the data, then transform it into JSON format and sends it back to the frontend. The frontend will show the post to the client.

This transferring of data between the client and the server is represented in the third word in the REST architecture which is transfer.

JSON format is commonly used in web services as well as other formats such as xml. JSON format has a key-value representation. For example, When the microservice sends the post back to the frontend, it could have the following format:

{  
 “title” : ”Lorem ipsum dolor sit amet, consetetur sadipscing elitr”,  
 “body” : ”Nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At…”  
}

The above scenario doesn’t show the actual communication of the implemented Blog, it only shows how REST architecture is employed in the Blog.

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

1. Client-Server
2. Stateless
3. Cache
4. Uniform Interface
5. Layered system
6. Code-On-Demand
7. Client-Server

As described in [47] “Separation of concerns is the principle behind the client-server constraints”. Such separation allows the client as well as the server to develop separately making the system loosely-coupled when it comes to the connection between the client and the server. Moreover, the client now has more portability where it can be a web application, a mobile application, or an application operating on any platform. As long as it respects the API offered by the server.

1. Stateless

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

1. Cache

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

1. Uniform Interface

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

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

1. Layered system

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

1. Code-On-Demand

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

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

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

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

Microservices details

1. ContactUs

This service provides the users of the Blog 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. This service provides only one API. This API help 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 1: ContactUs API

Listing 4.1 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 where the service is dyploed in. This service contact other services to make sure that users are not submitting invalid information or to protect itself from spam attacks. To contact other services ‘ContactUs‘ uses Axios to make HTTP calls. Axios is Promise based HTTP client for the browser and node.js [49]. Promise simply means the final result of an asynchronous operation. While HTTP stands for Hypertext Transfer Protocol. The result could have one of three values: the request is fulfilled, the requested is denied or the request is still pending. A callback 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 callback function and act accordingly.

Depending on the results received from the services that are called by ‘ContactUs‘ 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.

ContactUs, has its ow database where it stores the messages submitted to the admins. MongoDB is the selected database management system to help store data for services in the Blog.

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

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

name :{

type: String,

required: true

},

email :{

type: String,

required: true

},

content :{

type: String,

required: true

}

});

Listing 2 : Structure of contact document

Listing 4.2 shows the structure of a document that’ll be stored in the contact database. It has a JSON-like structure. It has attributes as well such as if a certain field is required or not or if it has a default value.

{

"name": "Lorem ipsum",

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

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

tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,

quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo.”

}

Listing 3: Contact data example

Listing 4.3 shows an example of how data can be stored in the Contact database. To retrieve the name, one show uses the key “name”. Same applies to the email and the content of the message.

1. User Registration

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

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

1. Login

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

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

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

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

npm install jsonwebtoken

Listing 4: Installing jsonwebtoken using npm

Listing 4.4 shows how package “jsonwebtoken“ can be installed from the command line using Node.JS Package Manager. This package will do all the heavy lifting to generate and sign the JWT before sending it to the client using the microservice Login.

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

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

Listing 5: 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. A JWT could have the following shape:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.

TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

Listing 6: Generated JWT [52]

Listing 4.6 shows an example of what JWT would look like after it has been generated.

1. Comment

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

1. Search

This service helps the user to search the posts for a specific term.

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. Moreover, one database will exist to serve the implementation of content trust. This database is accessible to all the microservices of the Blog. This database will be using MongoDB as its Database management system. It will have two collections:

* Services
* Relations

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

* ID of the microservice
* Name of the microservice
* Sensitivity of the microservice
* Development source
* Start date of operation
* Difference

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 come from the nature of the task each microservice performs. For example, microservices that handles clients logins have a higher sensitivity than microservices that handles bringing comments from the database to the frontend. Sensitivity of the microservices has three classes:

* High
* Medium
* Low

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

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

The starting date of operation will also be stored in the database for each service. This will help to calculate the age of each microservice. The content trust implementation for the Blog will consider if a microservice has an old age then its trust evaluation will be more positive than if it has a young operation age. Essentially, this is a design decision and depends on the context of where the content trust implementation is implemented. 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 be stored also and 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. Services collection will only store one value. This value represents the difference between the successful interactions and the failed interactions. If this value was positive then the number of successful interactions is higher than the number of failed interactions. While if this value was negative then the number of failed interactions is higher than the number of successful interactions. Depending of the value has negative or positive value the trust evaluation will be affected accordingly.

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 will contain an ID 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 IDs 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 a document, in the relationships collection. Each document contains the ID of a microservice and evaluations of trust from other microservices to this microservice. Those trust evaluation represents the overall trust evaluation that this microservice gained from interacting with other microservices. If the value is null, then no previous interaction between those two microservices happened in the past.

47 Architectural Styles and the Design of Network-based Software Architectures Roy Thomas Fielding

48 <https://nodejs.org/en/about/> 14-11-2018

49 <https://www.npmjs.com/package/axios 2-12-2018>

50 <https://www.mongodb.com/what-is-mongodb 11-02>

51 SQL and NoSQL Databases Vatika Sharma, Meenu Dave

52 The JWT Handbook Sebastián E. Peyrott, Auth0 Inc

should not be mentioned THE LITTLE BOOK ON REST SERVICES by Kenneth Lange