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
* Spamprotector
* 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. 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 each service and how they’re implemented.

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. 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.

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

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