Université Catholique de Louvain

Samir Marini Mohamed
SALEY ABDOU Djafarou

LINGI2145 *: Cloud Computing*
Project *: Mirco-Service Architecture*

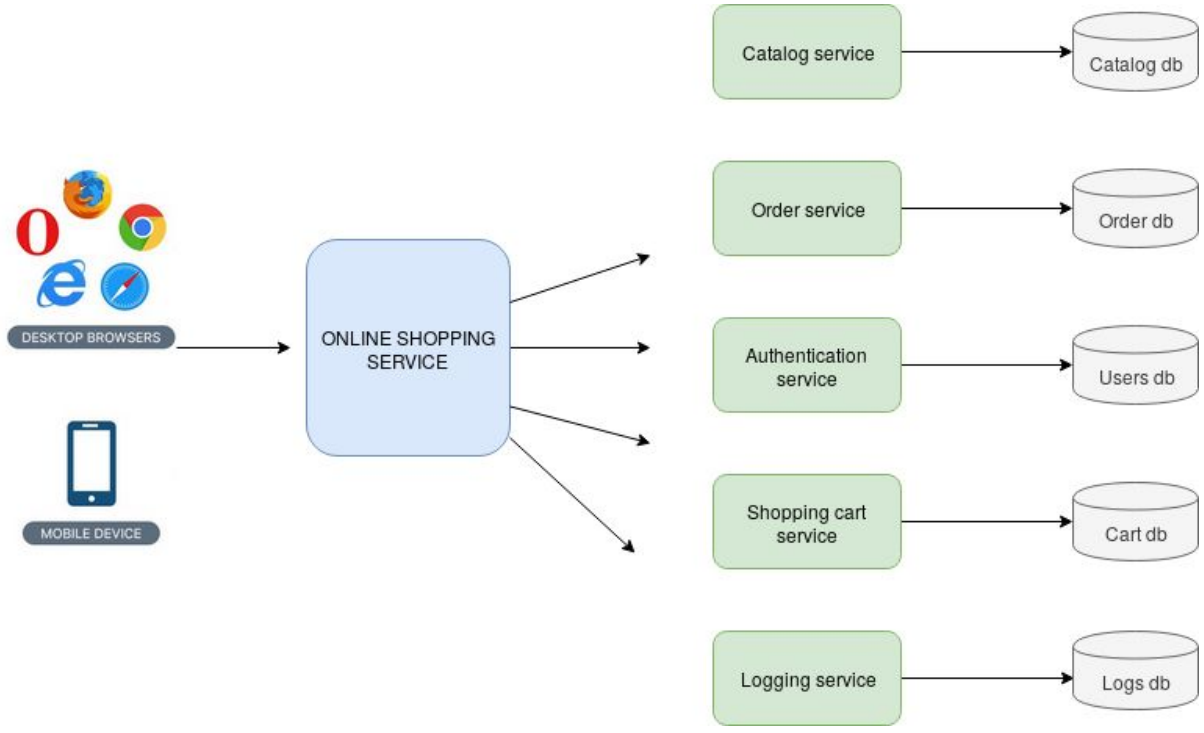
2019-2020
Group 12

Table of Content

1. Introduction

As part of this project, we were asked to take on the role of a back-end development team for a fruit and vegetable online ordering website. This site contains the basic elements of client authentication with username and password. A customer can then place products in his basket to finally make the actual order. We must take into account the fact that Admins will have to maintain this site, so it is essential at our level to distinguish between a normal user and an admin who is trying to make changes to the website. We first worked on virtual machines locally within our company and then deployed our work on the Microsoft Azure platform.

2. Microservices structure



2.1 Catalog service *:*

The first micro-services that we have decided to present to you is the catalog service, this micro-services is responsible for listing in its associated database all the products (fruit and vegetables) that will be intended to be posted by the front-end team on the website. It lists for each fruit/vegetable its name, price, a link to its corresponding image and finally its category.

Logging :
We believe that there is no point in keeping in logs the products we have previously added to the catalogue, as it does not give us any useful information about the user's behaviour.

Limitations :
We were unable to store the images in Microsoft Azure.

Code structure :
src/back-end/catalog-service/
```
├── Dockerfile    << Docker file
├── boot-in-order.sh  << script to automate database creation
├── gulpfile.js   << configuration file of development tasks
├── package.json  << node.js configuration file
└── src/
    ├── app.js    << REST API allowing POST, GET, DELETE, PUT
    ├── daemon.js << launch the HTTP server, will help to make request at port 4000
    └── utils/  << utils for the database
```

2.2 Cart service :

The second micro-service is the cart service, this micro-service aims to list what the customer decides to list in his basket in order to buy in the near future. It lists in its database the user's username as well as the name of the selected product, the selected quantity of the latter and its category.

We believe that the user will benefit from a persistent shopping cart even after a disconnection, that's why we decided to make it a micro-service with its own database.

Technologies : We use axios to make requests to the logging service.

Logging :
Each time a basket is created, it is sent to the micro logging service which will store it in the database.

Limitations :
We didn't have time to properly format the JSONs sent to the database, we send simple enough JSONs of the style {"product: nameOfProduct", "amount : nb", ..} but we wanted to have something more developed like :
{"products :
nameofProduct1, qt, price
nameofProduct2, qt, price
....
}

Code structure :
src/back-end/cart-service/
├── Dockerfile << Docker file
├── boot-in-order.sh << script to automate database creation
├── gulpfile.js << configuration file of development tasks
├── package.json << node.js configuration file
└── src/
  ├── app.js << REST API allowing POST, GET, DELETE, PUT
  ├── daemon.js << launch the HTTP server, will help to make request at port 4000
  └── utils/ << utils to create the database

2.3 Order service :

The third micro-service implemented is the history order. This microservice lists on the basis of a username the history of all its purchases since the creation of its account on the site

Technologies :
We use axios to make requests to the logging service.

Logging :
Each time a purchase is effectued, it is sent to the micro logging service which will store it in the database.

Limitations :
Same problem here, we didn't have time to properly format the JSONs sent to the database, we send simple enough JSONs of the style {"product: nameOfProduct", "amount : nb", ..} but we wanted to have something more developed like :

{"Oders :
nameofProduct1, qt, price
nameofProduct2, qt, price
....
}

Code structure :

```
src/back-end/order-service/
├── Dockerfile    << Docker file
├── boot-in-order.sh  << script to automate database creation
├── gulpfile.js   << configuration file of development tasks
├── package.json  << node.js configuration file
└── src/
    ├── app.js    << REST API allowing POST, GET, DELETE, PUT
    ├── daemon.js << launch the HTTP, will help to make request at port 4000
    └── utils/  << utils to create the database
```

2.4 Logging service :

The fourth microservice implemented is the microservice logging, it collects a maximum of information on the behavior of the user, so that we use it in the future to create a recommendation service.

Technologies : Each time another micro-service will make a request to insert something into the database, a post request (via the use of axios) will be made to the logging service that will record this insertion there.

API :

Limitations :
We were unable to record the response times of the requests.

```
src/back-end/logging-service/
├── Dockerfile    << Docker file
├── boot-in-order.sh  << script to automate database creation
├── gulpfile.js   << configuration file of development tasks
├── package.json  << node.js configuration file
└── src/
    ├── app.js    << REST API allowing POST, GET, DELETE, PUT
    ├── daemon.js << launch the HTTP, will help to make request at port 4000
    └── utils/  << utils to create the database
```

2.5 A word on our .yml file :

We have also made changes to the configuration file of our docker swarm from the following link =>
LINGI2145-2019-2020/project/src/

src/
├── back-end/ << contains the back-end
├── front-end/ << contains the front-end
├── scalability/ << related to scalability
├── curl-test.sh << script used for automated testing (not important here)
├── deploy.sh << the script to automate docker deployment
├── scapp.yml << THIS ONE

here is a snippet of the following file :

You can see here a tag called "scapp-fe", it is indeed the front end which contains the url followed by the ports to the different services. We have incremented by 1000 at the port level by added micro-service, the name of the variable contains a word allowing each time to guess which micro-service it is

You can see here still in the yml file the declaration of two tags. Indeed by microservice, we have each time a tag xxx-db and xxx-service, the first one corresponds to the correlation with the image of the db couch database and the second one corresponds to the HTTP server that will receive the requests

```
orders-db:
  image: meroxbe/storage-service:shopapp
  ports: [ "6001:5984" ]
  deploy:
    replicas: 1
    restart_policy:
      condition: on-failure
  networks: [ "scapp-net" ]
order-service:
  image: order-service:shopapp
  depends_on: [ "orders-db" ]
  ports: [ "6000:80" ]
  deploy:
    replicas: 1
    restart_policy:
      condition: on-failure
  networks: [ "scapp-net" ]
```

2.6 A word on the front-end:

Our team was not part of this project to deal with front-end changes but we did our best to provide as detailed an API as possible so that the front-end team would have the light task. Developing the back-end has also led us to a good understanding of the front-end and we have seen several possible correlations, so the way is open for further improvements with the front-end team in the future

3. REST API

3.1 MicroService : catalog-service

The catalog microservice (catalog-service) exposes a REST API over HTTP. As a reminder, this micro-service is used to store product information

Method	Uniform Resource Name (URN)	Required parameters	Output	Description
POST	/catalog/:username	name=[string]&price=[double]&image=[string]&category=[string]	catalog-token	Add a new product in the databse
GET	/catalog/:name/:username		catalog-token	get product information via it's username
PUT	/catalog/:name/:username	price=[double]&image=[string]&category=[string]	catalog-token	update product information
DELETE	/catalog/:name/:username		catalog-token	delete product

How can you test this API with curl ?

Please, follow the step in the order otherwise you'll potentially request empty object in the database. You'll notice that we always use .../admin at the end of our request for the simple reason that it was requested for this project to allow product modification only to administrators and no one else

Please don't forget to replace :
-> IP_ADDRESS = the ip address of the machine running the container
-> PORT_AUTH = the port on which the container is running

Add a new product in the database using the curl command-line HTTP client

curl -X POST --data "name=kiwi&price=1.74&image=https://www.alimentarium.org/fr/system/files/thumbnails/image/alimentarium_kiwis.jpg&category=fruit" IP_ADDRESS:PORT_AUTH/catalog/admin

Get the product information via it's username using the curl command-line HTTP client

curl -X GET IP_ADDRESS:PORT_AUTH/catalog/kiwi/admin

Update product information using the curl command-line HTTP client

curl -X PUT --data "price=2.74&image=https://www.alimentarium.org/fr/system/files/thumbnails/image/alimentarium_kiwis.jpg&category=fruit" IP_ADDRESS:PORT_AUTH/catalog/kiwi/admin

remark : in this example, we've changed the value of the field 'price' from 1.74 to 2.74

Delete product using the curl command-line HTTP client

curl -X DELETE IP_ADDRESS:PORT_AUTH/catalog/kiwi/admin

3.2 MicroService : cart-service

The cart-service microservice (cart-service) exposes a REST API over HTTP. It stores information on a couch db regarding the shopping-cart of the current user, his basket in other words

Method	Uniform Resource Name (URN)	Required parameters	Output	Description
POST	/cart	username=[string]&products=[double]&totalItems=[int]&totalAmount=[double]&category=[string]	cart-token	Add a new cart in the databse related to a client

GET	/cart/:username		cart-token	get cart information via it's username
PUT	/cart/:username	products=[string]&totalItems=[int]&totalAmount=[double]&category=[string]	cart-token	update cart information of a client
DELETE	/cart/:username		cart-token	delete cart

How can you test this API with curl ?

Add a new cart in the database using the curl command-line HTTP client

curl -X POST --data "username=kevin&products=pomme&totalItems=4&totalAmount=6,78&category=fruit" IP_ADDRESS:PORT_AUTH/cart

Get cart information via it's username using the curl command-line HTTP client

curl -X GET IP_ADDRESS:PORT_AUTH/cart/kevin

Update cart information using the curl command-line HTTP client

curl -X PUT --data "products=pomme&totalItems=5&totalAmount=7,78&category=fruit" IP_ADDRESS:PORT_AUTH/cart/kevin

remark : in this example, we've changed the value of the field 'totalAmount' from 6.78 to 7.78

Delete product using the curl command-line HTTP client

curl -X DELETE IP_ADDRESS:PORT_AUTH/cart/kevin

3.3 MicroService : order-service

The order history microservice (order-service) exposes a REST API over HTTP. It stores information on a couch db regarding the history of every purchase of every client of the website

Method	Uniform Resource Name (URN)	Required parameters	Output	Description
POST	/history	username=[string]&orders=[string]&totalItems=[int]&totalAmount=[double]	order-token	Add a new order in the databse for a client history
GET	/history/:username		order-token	get client history information via it's username
PUT	/history/:username	orders=[string]&totalItems=[int]&totalAmount=[double]	order-token	update history information

How can you test this API with curl ?

Add a new order in the database using the curl command-line HTTP client

curl -X POST --data "username=kevin&orders=pomme&totalItems=1&totalAmount=1" IP_ADDRESS:PORT_AUTH/history

Get order information via it's username using the curl command-line HTTP client

curl -X GET IP_ADDRESS:PORT_AUTH/history/kevin

Update order information using the curl command-line HTTP client

curl -X PUT --data "orders=kiwi&totalItems=1&totalAmount=2" IP_ADDRESS:PORT_AUTH/history/kevin

remark : in this example, we've changed the value of the field 'orders' from pomme to kiwi

3.4 MicroService : logging-service

The logging microservice (logging-service) exposes a REST API over HTTP. This micro-service is responsible for tracking every times a client decides to put sth in his back in order to know better about him.

Method	Uniform Resource Name (URN)	Required parameters	Output	Description
POST	/logs	name=[string] &price=[double]&image=[string]&category=[string]	pro-token	Add a new log in the database

How can you test this API with curl ?

Add a new log in the database using the curl command-line HTTP client

This service is intended to be called only by another service, so for example if you want to add a log, you can do it through the catalog service (by adding a product for example).

curl -X POST --data "name=admin&price=1.74&image=https://www.alimentarium.org/fr/system/files/thumbnails/image/alimentarium_kiwis.jpg&category=fruit" IP_ADDRESS:PORT_AUTH/catalog

3.5 MicroService : auth-service

The order auth microservice (auth-service) exposes a REST API over HTTP. It stores information on a couch db regarding username and password for every client registered on the website

Method	Uniform Resource Name (URN)	Required parameters	Output	Description
POST	/user	username=[string]&password=[string]	user-token	Add a new user in the database f
GET	/user/:username/:password		user-token	get user information via it's username

How can you test this API with curl ?

Add a new user in the database using the curl command-line HTTP client

curl -X POST --data "username=kevin&password=bob" IP_ADDRESS:PORT_AUTH/user

Get user information via it's username using the curl command-line HTTP client

curl -X GET IP_ADDRESS:PORT_AUTH/user/kevin/bob

4. What about scalability?

After deploying all the services, we have made available in the folder LINGI2145-2019-2020 a script called scale.sh that will ensure scalability on all services running in the docker swarm.

Unfortunately, we did not have enough time at our disposal to create a test script simulating a massive influx of HTTP requests in order to show you how scalability works at work, but this option is of course part of a similar development

5. Script and automatic deployment of docker

5.1 Automation in image building with Dockerfile

Browse through any file tree to find the corresponding Dockerfile for the microservice and then build the image is time-consuming and cumbersome. We have therefore created a script that automates the creation of images for all our micro services and ensures that the old ones are deleted so as not to create empty images.

We have hosted all the images of our containers on DockerHub, so you will unfortunately not be able to test this script directly in the azure virtual machine, but for curiosity's sake, you can test it on a local virtual machine.

You can also use this script in case of problem, we have made a script to build all the images locally, this one is called build-image.sh.

This file is located:

src/back-end/
├── history-order/ << a micro-service
├── product-catalog/ << a micro-service
├── shopping-cart/ << a micro-service
├── storage/ << a micro-service
├── users/ << a micro-service
├── **build-image.sh** << the script that will automate the docker build

You can easily see that this script does the work for you to build all the images of our micro-service

You can run this script by typing on your command line ./build-image.sh like this :

This script is really suitable especially for local testing virtual machine in a docker swarm environment because our team has created another script to automate the deployment of docker swarm and this same script will use build-image.sh to build the images before deploying the docker swarm but all these works locally.

5.2 Automation in Docker deployment

As mentioned above, our team has designed a script to automate the deployment of docker swarm, this is located in :

src/
├── back-end/ << contains the back-end
├── front-end/ << contains the front-end
├── scalability/ << related to scalability
├── curl-test.sh << script used for automated testing (not important here)
├── **deploy.sh** << the script to automate docker deployment
├── scapp.yml << file used to deploy docker swarm

To run this script, you will first have to go in the folder mentioned above, then you can run it like this : ./deploy.sh

You should see at the end of the execution the message "Docker is ready" like this.

There it is, you've successfully deploy the docker swarm, you can see all the services running by typing 'docker stack services scapp' on your terminal. This script is not really needed when hosted on Microsoft Azure VM because images are already hosted and deploying the docker swarm is reduced to a simple command line which is equivalent to run the script.

5.3 Automation in HTTP request testing

The last script that our team implemented is the script that will make all HTTP requests with the curl tool on all our micro services by scrupulously following their respective REST API

This script can be found here :

src/
├── back-end/ << contains the back-end
├── front-end/ << contains the front-end
├── scalability/ << related to scalability
├── **curl-test.sh** << THIS ONE
├── deploy.sh << the script to automate docker deployment
├── scapp.yml << file used to deploy docker swarm

You can run this script by typing on your terminal './curl-test.sh' like this :

```
user@my-debian:~/LINGI2145-2019-2020/project/src$ ls
back-end  curl-output.txt  curl-test.sh  deploy.sh  front-end  scalability  scapp.yml
user@my-debian:~/LINGI2145-2019-2020/project/src$ ./curl-test.sh
```

By inspecting the code more carefully, we can see very clearly that it is a set of curl requests respecting the REST API of each micro-service, the outpout of all these commands will be redirected in a structured way in an output file that will be put in the same folder and that will be called "curl-output.txt"

This system allowed us to demonstrate that all our micro services were working as intended :

6. Project deployment, Microsoft Azure and tests : instructions

Deploy project (local) :

1. Before starting you need to create a swarm and a network called scapp-fe.
2. Then clone our Github repository on your VM
3. Go to the folder LINGI2145-2019-2020/project/src
4. Deploy the stack using the command:
   docker stack deploy -c scapp.yml scapp
5. This command will download all the docker images from our Dockerhub and run them with the right parameters.

In case of problem: We have created a script that can do all the previous commands for you. To run it, simply write **./deploy.sh** in the current folder.

From now on, all services are running.

Deploy project (cloud) :

We will discuss here the steps to be taken in order to test the deployment of our microservices on Microsoft Azure.

1. The first step is to create two different virtual machines using the necessary commands, the first one will act as a "manager" and the second one as a "worker".
2. Try to make sure that the virtual machine manager can connect to the worker virtual machine via an SSH connection without having to use a password
3. Select the terminal corresponding to the manager's virtual machine and initialize the docker swarm using the command :
   docker swarm init
4. You can add the worker as a docker node using the command below:
   docker swarm join --token <TOKEN_ID> <IP_OF_YOUR_LOCAL_HOST>:2377
5. Clone our git repo **on your local vm**
6. Go to the LINGI2145-2019-2020/project/src/ folder
7. Copy the .yml file to the azure vm :
   scp LINGI2145-2019-2020/project/src/scapp.yml managerID@managerDNS-yourGithub.westeurope.cloudapp.azure.com: ~/

8. You can now deploy the docker stack with the command :
   docker stack deploy -c scapp.yml scapp

Enjoy ! All our services are running in the cloud (We didn't manage to save the images to Azure storage)

Run tests (local and cloud) :

Go to : /LINGI2145-2019-2020/project/src
Run ./curl-test.sh

Conclusion

During this project we developed our skills in the development of back-end websites. We have deployed our micro-services on the Microsoft Azure platform to test the proper functioning of all functionalities. The way is clear for many improvements regarding the current state of our development but if time allows, we will work on it again to propose something even better.