Université Catholique de Louvain

Samir Marini Mohamed
SALEY ABDOU Djafarou

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

*2019-2020 - Group 12*

# *Project Part 2 :*

# *1 The role of the service*

Through this section, we will introduce you to micro-services that have undergone changes within our back-end in order to implement a personalized recommendation service for each user.

In this introduction, I explain how our recommendation engine works:

We decided to base our recommendation engine on the purchase history of the connected user because, as Wiston Chruchill said, “The farther back you can look, the farther forward you are likely to see”. In the folder containing the relative files in the order-service micro-service, I implemented a map-reduce to retrieve the history of each user based on a key that corresponds to his username. In this way, it is sufficient at the front-end level to provide this micro-service with the string corresponding to the user name of the connected client to retrieve its information.

I then implemented a map-reduction in the folder containing the micro-service catalog-service in order to retrieve without providing a key the entire catalogue of fruits and vegetables present on our website. In a third and final step, the micro-service logging service cross-references the information collected on the user's history with the information collected on the appearance of the products offered on our website, then looks at a map reduction where the key is the name of the fruit/vegetable to see if it is a fruit or vegetable and will count everything in order to know if the user is more likely to buy fruit or vegetables.

In the case where the connected user has more purchased fruit in his history than purchased vegetables, it will then be decided to offer this customer to buy a fruit that he would not have bought yet in his history and vice versa if it is vegetables. The information will then be presented to him as a pop-up message on the front-end of the application.

Here is a graphical representation of the file organization at the back-end level
You will notice that each micro-service involved in the management of the recommendation engine contains a folder named xxx_init_db, this folder contains the declaration of the map-reduce function used for queries on the database as well as a script that will automatically fill each database with values encoded in a json

```
src/back-end/
├── cart-service/   <<  a micro-service
├── catalog-service/  << a micro-service
  ├─ catalog_init_db/
     ├─ dataset/  << contains data to fill the database
```

```
    ├─ views/    << implement the map-reduce function
    ├─ Dockerfile  << build the image
    ├─ clean-output.sh  << automate some request on database using the views
    ├─ run-curl.sh  << script involved in recommendation engine, more explanation below
├── logging-service/   << a micro-service
  ├─ logging_init_db/
    ├─ dataset/  << contains data to fill the database
    ├─ views/    << implement the map-reduce function
    ├─ Dockerfile  << build the image
    ├─ clean-output.sh  << automate some request on database using the views
    ├─ preference.sh  << the heart of the recommendation engine, more explanation below
    ├─ recommendation.sh  << keeps updating preference of users
├── order-service/  << a micro-service
  ├─ order_init_db/
    ├─ dataset/  << contains data to fill the database
    ├─ views/    << implement the map-reduce function
    ├─ Dockerfile  << build the image
    ├─ clean-output.sh  << automate some request on database using the views
├── users/   << a micro-service
├── build-image.sh  << the script that will automate the docker build
```

## 1.1 Catalog-service

We can first look in the folder catalog-service/ on the script run-curl.sh. This script runs on the basis of this command ./run-curl.sh $1 $2 where $1 is the ip address of the vm and $2 is the name of the connected user in order to create a file to protect his output name. This script will generally allow us to know on the basis of a user's history whether the products he has bought are fruits or vegetables. Let's look in this script at the command **curl admin:admin@$1:4001/catalog/_design/queries/_view/catalog?key='"'"${line}"'"'** , it's indeed this command that will use the map reduce function implemented at the catalog-service level to know if a product given as a key corresponds to a fruit or vegetable. In this command, the **key='"'"${line}"'"'** corresponds to the line of the fruit or vegetable read in the file containing the user's history and after translation, the result is put which will correspond either to the string "fruit" or "vegetable" in an output file in order to be able to count in a second time if the user consumes more vegetables or fruits.

We will now talk about the map-reduce function implemented in the user_queries.js file located at catalog_init_db/views/. Here is its content :

```
const viewDescriptor = {
  'views': {
        'catalog': {
                'map': 'function (doc) { \
                        if (doc.name && doc.category) { \
                                emit(doc.name, doc.category) \
                        } \
                }',
                'reduce': 'function(key, values) { \
                        return values; \
                }'
                }
        }
}
module.exports = { viewDescriptor }
```

We have decided to use as the key for our map function the name of the fruit we want to study within the database, it corresponds to the doc.name. We then set as value the category corresponding to the name of the fruit given in key which corresponds to the doc.category.

Our reduce function simply returns the category of the name of the given fruit in key.

This choice of implementation allowed us in our recommendation engine to know which category of each fruit or vegetable was given as a key to the request, it was almost essential to make the computer understand later that this or that user was consuming more or less fruit or vegetable

A quick reminder, at the output.json file located in the dataset/ folder still in the catalog_init_db/, this file serves as data to be injected into the database via a script when it is created.
It is a json file and we remind you that an element in the catalog database must contain the following elements, name, price, image, category and we decide to customize the id with the name of the element as realized in this example:

**{"name":"kiwi","price":1,"image":"https://www.alimentarium.org/fr/system/files/thumbnails/image/alimentarium_kiwis.jpg","category":"fruit","_id":"kiwi"}**

## 1.2 Order-service

the order-service micro-service intervenes in an underlying way to the recommendation engine by providing the history for a given user at a particular time

We can look at the clean-output.sh script of this micro-service, more particularly at the command **curl admin:admin@$1:6001/order/_design/queries/_view/order?key=$2** in this script. This command will use the map-reduce functions implemented in its database to provide a user's history based on a key given as a parameter that is its identifier. In this command, $1 corresponds to the ip address of the virtual machine and $2 to the user name

this script is executed by typing the command **./clean-output.sh $1 '"claude"'**

"claude" is a random user name

We can now look at the user_queries.js file in the order_init_db/views folder, which implements the map-reduce functions to make it easier for us to use the recommendation engine. Here is its content:

```
const viewDescriptor = {
  'views': {
        'order': {
                'map': 'function (doc) { \
                        if (doc.username) { \
                                emit(doc.username,doc.orders) \
                        } \
                }',
                'reduce': 'function(key, values) { \
                        return values; \
                }'
        }
  }
}
module.exports = { viewDescriptor }
```

Our map function takes the user name as the entry key and highlights his shopping history basket. The reduce function simply returns this shopping history basket.

We decided to implement this function in this way because the identifier is unique on our application and it is therefore a good key for a request and the purchase history basket is well structured so that it is easy to manipulate what is returned.

A quick reminder, at the output.json file located in the dataset/ folder still in the order_init_db/, this file serves as data to be injected into the database via a script when it is created.
It is a json file and we remind you that an element in the catalog database must contain the following elements, username, orders (You will notice that the elements are separated by '_'), totalItems, totalAmount and we decide to customize the id with the username of the element as realized in this example:

**{"username":"pierre","orders":"kiwi_orange","totalItems":2,"totalAmount":3,"_id":"pierre"}**

# *2 Logging service*

The folder associated with the micro-service logging-service/ contains a script called preference.sh which is the core of our recommendation engine. The different steps of this script are summarized in the following points:

- -the script will go to the order-service/ folder to execute the clean-output.sh script in order to retrieve the history of the connected user. Then move the file containing the history of the formatted user to the logging-service folder
- The script then goes to the catalog-service/ micro-service folder to execute the run-curl.sh script which will return it for each product purchased by the user if it is a fruit or vegetable and the output file will be moved to the logging-service/ folder for future processing
- Using a condition, the script will check if the file file fileOfCategory.txt containing the category of products purchased by the user contains more fruit or vegetables
- Depending on the previous step, let's take the case where the file fileOfCategory.txt contains more fruit than vegetables, it means that our user consumes more fruit than vegetables, we will then use the script clean-output-fruit.sh which will use the map-reduce implemented in the database containing the catalog of our application to output the entire list of fruits with the following curl request **curl admin:admin@$1:4001/catalog/_design/queries/_view/catalog?group=true | grep 'key' | grep 'fruit' | cut -d ',' -f 1 | cut -d ':' -f 2**
- The script will then compare the differences between the file containing all the catalogue fruits to find those that the user has not yet purchased. After finding them, it groups them together in a txt file with the user's name so as not to create confusion and we have our recommendation engine customized by user. The opposite will of course be done if the user consumes more vegetables than fruit

2.1 Link with the front-end :

We have implemented at the front end a code to deploy a pop-pup message window in order to propose on the basis of our recommendation engine a product to be consumed by the customer. This is an axios method which makes a get to know if the user is a new user or not, if this one is a new user, we can then propose him the pop-up, if not, it's useless because we won't know his preferences.

If the user clicks on cancel in the pop up menu, he will no longer be offered this product. We will wait before the next re-run of the search engine to offer him a product even more adapted.

# *3 Microsoft Azure storage blob*

As requested in the instructions, we have added the possibility to add the images from the catalogue in azure blob

How does it work? The site administrator has at his disposal an API which is as follows:

| **Method** | **Uniform Resource Name (URN)** | **Required parameters** | **Output** | **Description** |
|---|---|---|---|---|
| POST | /catalog/:username | name=[string]&price=[double]&image=[string]&category=[string] | Azure URI link | Add a new product in the database |

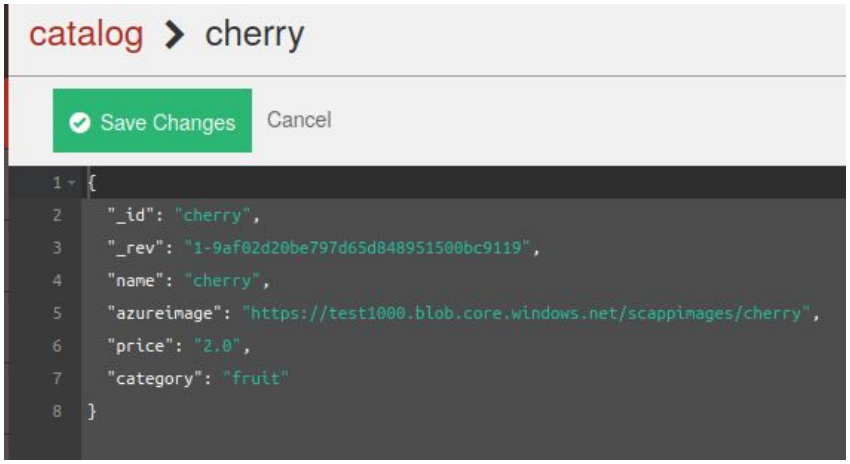
When the admin adds a product to the catalog, the website will store the image of the product in Azure blob and will generate a link that will be inserted in the database. This is the link of the image in an Azure blob.

This link can then be retrieved via the standard output or via the GET method of the same API and then it can be used to display the product in the front end.

Let’s see an example

1. I add a cherry to the catalogue:
   *curl -X POST --data "name=cherry&image=https://www.julieandrieu.com/245-book_product_image_size_default/cerise.jpg&price=2.0&category=fruit" 192.168.56.102:4000/catalog/admin*

2. This is the result in the database :




3. As you can see, an azure link has been generated, just retrieve it with a get call (*curl -X GET 192.168.56.102:4000/catalog/cherry/admin*) : https://test1000.blob.core.windows.net/scappimages/cherry
4. The image is on Azure blob!

**Bonus : resizing**

We implemented the resizing via the use of the jimp module (https://www.npmjs.com/package/jimp), before sending the image on azure blob, we modify its size.

We based ourselves on this microsoft documentation to do so: https://docs.microsoft.com/en-us/samples/azure-samples/storage-blob-resize-function-node/storage-blob-resize-function-node/

# *4 Deployment*

***we have tried as much as possible to make our program customizable but please try to use the ip adress 192.168.56.102***

-First thing to do, in the preference.yml file located in projet>src>back-end> logging-service, cange all occurrences of the address 192.168.56.102 by the ip address of your VM so that it can work.

So in order not to waste your time when deploying at all, we invite you not to use this script as indicated in the report and to build the images manually by following the preference file.yml located in the LINGI2145-2019-2020/project/src/back-end folder, the image names of the form xxx-db should be built with the docker file located in back-end/storage, the image names of the form xxx-services should be built with the dockerfile located in back-end/xxx-service and the image names of the form xxx_init_db should be built with the dockerfile located in back-end/xxx-service/xx_init_db. This structure follows the one you used in the tutorials.

You can then execute the command

**chmod -R +xr .**

in the folders

back-end/logging-service/

back-end/order-service/

back-end/catalog-service/

in order to make all scripts executable.

You can deploy the docker swarm with the command

**docker stack deploy -c preference.yml preference**

with the preference.yml file which is tortuous for recall
LINGI2145-2019-2020/project/src/back-end

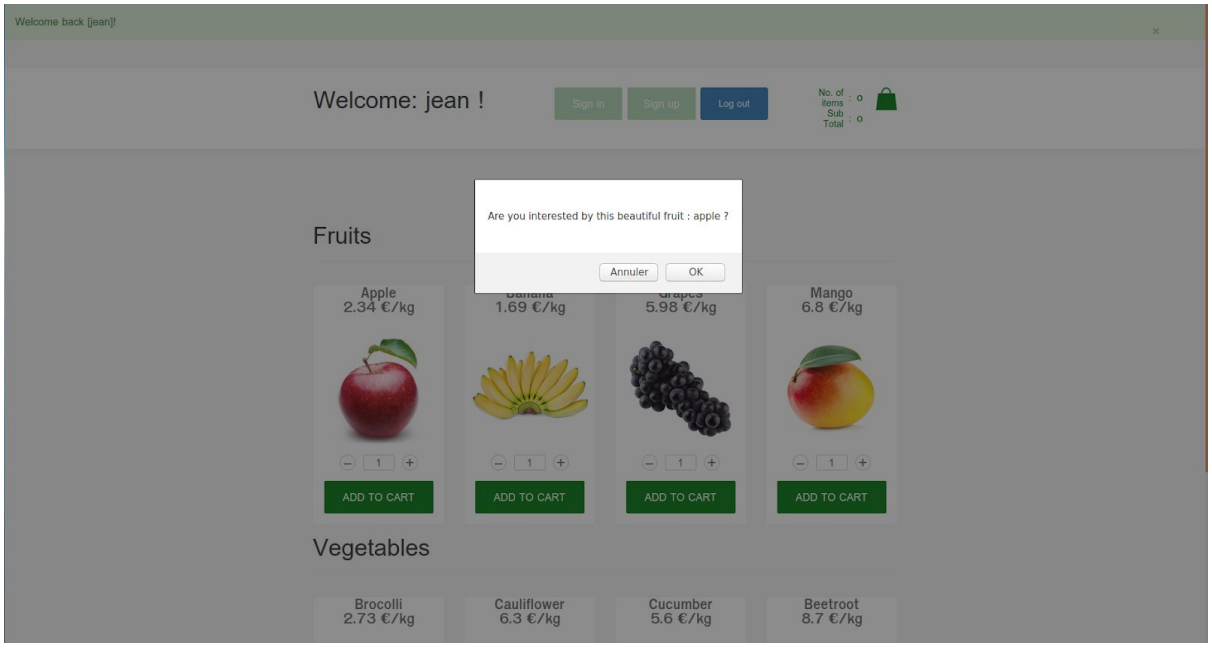
the last step will be to go to the back-end/logging-service folder and launch the recommendation engine with the command

**(./recommendation.sh $1 -i 1 -d 10 &> tmp.txt)&**

There you go,

***1 create an account***
***2 log out***
***3 log back in and you'll get a pop up like this giving you a recommendation***

result :



# *5 Choice of implementation for the recommendation engine*

I decided for the recommendation engine to work mainly with bash scripts for the simple reason that I was screwing around with a history that had its own micro-service and the logging-service was already almost a recommendation since it was already tracking user preferences. That's why we didn't work that much in the recommendation micro-service, which was already grouped together in all the others.

could better structure this output file.

rgeted advertising engine for the simple reason that our program will know if the user is more likely to buy fruits or vegetables, if one has the advantage over the other, this way we can

# *5 API Update*

The main Rest Api having been modified is that of logging-service, here are the modifications

| **Method** | **Uniform Resource Name (URN)** | **Required parameters** | **Output** | **Description** |
|---|---|---|---|---|
| POST | /catalog/:username | name=[string]&price=[double]&image=[string]&category=[string] | Azure URI link | Add a new product in the database |

| | | | | |
|---|---|---|---|---|
| POST | /logs | username=[string]&logs=[string] | Succes confirmation | Add new logs related to the user account “username” |
| DELETE | /logs/admin | x | Succes confirmation | Delete logs |
| GET | /logs/admin | | Succes confirmation | Get logs |

# *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.*