UCLouvain

LINFO 2145 : Cloud Computing

Project report

Groupe 30

December 2021

Charles Lohest
Romain Tourpe

Année 2021-2022

Professeur :
Etienne Riviere

Table des matières

1 Introduction **1**

2 Back-end **1**

2.1 Catalog . . . 1
2.2 Shopping-kart . . . 2
2.3 Cart-History . . . 2
2.4 Users . . . 3
2.5 Storage . . . 3
2.6 Scalability . . . 3
2.7 Logs . . . 3
2.8 Map-Reduce . . . 4
2.9 Implementation . . . 4
2.9.1 Catalog . . . 4
2.9.2 Shopping-cart . . . 5
2.9.3 Cart-history . . . 5
2.9.4 Logs . . . 6
2.9.5 Views . . . 6

3 Improvement on first deliverable **6**

4 Front-end **7**

5 Conclusion **8**

1 Introduction

For this project, we have been tasked with the design and implementation of the backend of a shopping-cart application. In order to match the specifications, we developped differents microservices based on RESTfull APIs. Each microservice has his own database, based on couchDB, and every microservice is run with docker.

2 Back-end

2.1 Catalog

This service stores all the item of the catalog with a name a price an image a category and an id. When the admin add a product it is added by the microservice to the database and each product can also be removed or modified. This services uses a database which contains all the items stored in the catalog in a specific format in order to fetch the catalog with the front-end see[Figure 1]. When a modification occurs in the database, the service realize a call to the logs database in order to post the new modification in this database.

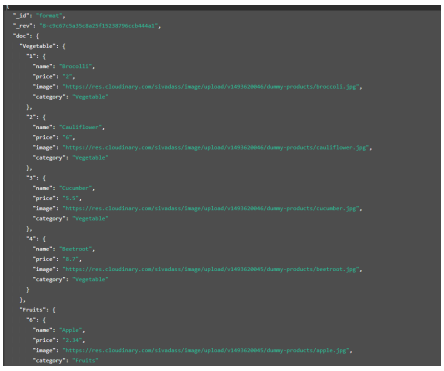


FIGURE 1 – format database

2.2 Shopping-kart

This service create a shopping-cart for every user that add item in his shopping cart. It stores the products added, the quantities of each product added, the id of each product, and the image URL of each product. We know it isn't the best option and we wanted to call other microservice in order to recover that kind of information in the front-end, but due to concurency problems, we had no choice. The best option would have been to only keep the name of the product and the quantity in the database. When a user is login the service get the shopping cart corresponding to this user. This service handle the basket of a user with a RESTfull API.

To do so, it has been implemented with one database, based on couchdb. You can acces his database with curl, with the address http ://cloud-romtourpe.westeurope.cloudapp.azure.com :3006 .

The image of the container is automaticaly build run with the command "make swarm" in the Makefile.

This container contains "boot-in-order.sh". His goal is to create the database in order that the microservice run properly.

In App.js, you can found the GET, POST and DELETE operations. You can use them as follows : - POST : you have to give him as argument the name of the product, a username, a quantity, the price of the product and finaly the id of the product. It will add your product with the given quantity to your basket. - GET : you have to give him as argument the username. The reponse from the database will be the basket of the user. -DELETE : you have to give him as argument a username and a product (who already is in the basket). It will remove the given product from the basket. If you only give as argument the username to the delete function, it will delete the entire basket of the specified user.

2.3 Cart-History

When the user checkout an history of his purchase is uploaded to the database by the history service. It gets the actual shopping cart of the user and store for the user a list of each shopping cart that he bought.It also has a list of each items that the user has ever bought and the quantity of each of those product. This service is hardly dependent to the shopping cart service. It uses a single database history with a post and a get method. The database create a document for each user where there is an history of all the shopping cart that they purchased, an history of every item that they ever purchased and the quantity of each item those items are stored in lists. Each history is a list and is updated at each checkout. For example if the item Apple is stored in the 'items' list at position 1 the the total quantity is stored in the 'quantity' list and is updated at every checkout. The post method will take a username in argument and will get the shopping cart of the user. The service check if the user

has already an history. If it succeed the cart is push in the shopping cart history and if he buys a new item that he never bought this item is added to the list 'items' or the quantity is updated if it was already bought. The get method takes a user as argument and return the whole history document corresponding to the correct user.

2.4 Users

This is the authentification service. This service is used to create and get users by clicking on the button register or login and giving username and password. The user is added to the service and a database is created. When a user is created he is posted to the logs service.

2.5 Storage

It is the service that creates the databases on couch db every services use this storage service to create a couch db and each service is link to its own storage image. The API can link the microservice with the couchdb tool.

2.6 Scalability

You can test the scalability of two containers with the command "make artillery". Theses containers are the user container and the catalog container. The only container for wich the scalability was usefull was the user one. After many test on the others, (up to 20 000 requests in less than 2 min), we conclude that there was no need to scale them, because the load never exceed 30It will, if the load is superior to 70% of the capacity, increase the number of replicas of the service by 3. The tests take place on two services : the authentification service and the catalog service. In order to launch the scalability script of the differents daemon, use the 'make scalability' command.

2.7 Logs

The logs service is very useful because it memorize the catalog, the users registered on the site the id of the last item added and many other information as the purchases history. When we need to make a call on the catalog we can just call this database so that there is no access to the real database which fetch the products. So if there is a fail in the security and a hacker try to modify the database he will only modify the logs. Those modifications will not impact the website. The logs does not contains any important information as passwords. Only a list of user is stored. For the example, this list is used when we need to check the shopping cart of each user. Each other services that need an information from another service will make a call on the logs database. This will impact this database because it will have to handle a lots of calls but it reduces the possible error by doing calls in different services and preserve the website security. Finally the logs service contains the recommendation service. Indeed we didn't create a separate service for the recommendation because we already store at each purchase an history of each products purchased with the list of the rest of the cart. There is two type of purchase history. The first one is a specific history to each user so when a user conclude the purchase the service will add the history in the user's history.

```
{
  'user':{
     'item1':{
        'name':'item1',
        'with':['item2'],
        'quantity':[10]
     }

```

```
  }
}
```

The second one is just a general history very similar to the first one but not specific to a user.

```
{
    'item1':{
       'name':'item1',
       'with':['item2'],
       'quantity':[10]
    }

}
```

With those information we simply apply a map reduce in order to get the recommendations.

2.8 Map-Reduce

The Map reduce is used in order to create recommendation. the return values is a list with items associated to a list of three items. The function gets the purchase history and iterate in this data to get each item and the lists 'with' and 'quantity' of each item then sort the both lists by checking the highest quantities. When the list is sort we take the three first elements and we create a new data

```
{
    'item1':['item2','item3','item4']

}
```

At the end we have the bests purchase association and we can return this new data. There is two map-reduce views. The first one returns the user's recommendations and the second the general recommendations

2.9 Implementation

2.9.1 Catalog

For each databases there is a POST GET and DELETE methods implemented in the API. The post method try to get the database called in order to update it. If the document is not created the method create a new document and insert it into the couchdb database. The GET method returns the document into the database and the delete methods remove an item in each databases.
Each method is called by the API with a specific path.
GET :
To get the catalog in the good format the path is url :[DBPORT]/format.

POST :
To post an item in the catalog with the right format the path is url :[DBPORT]/format with providing a data with the name,price,image,category and id of the product for example data = {'name' :'Apple','price' :2,'image' :'image.png','category' :'Fruit','id' :1}.

DELETE :
To delete the catalog in the good format the path is url :[DBPORT]/format/[name].

When the get is called by the api we return the function addformat. This function first get the catalog in order to check if the document already exists. If the response is successful we check in the catalog if the category of the product to add exists. Then we create a new category if it does not exist or we check if the product is already stored in the catalog in order to updates the data of the product or to create a new one in the right category.
We use the same principle to remove the item from the catalog. We had to check if the item is the last of his category to also remove the category. When an item is post in the catalog the services makes a POST call on the logs micro-service in order to store the information of the product.

2.9.2 Shopping-cart

For this database, there is a POST GET and DELETE method implemented in the API. The post method try to get the database called in order to update it. If the document is not created the method create a new document and insert it into the couchdb database. The GET method returns the document into the database and the delete methods remove an item in each databases.
Each method is called by the API with a specific path.
GET :
To get the basket of an user, the path is url :[DBPORT]/shopping-kart/username.
POST :
To post an item in the basket the path is url :[DBPORT]/shopping-kart. You have to provide data such as the username, the name of the product, the quantity, the price and the url of the image.An example of data xould be data = {'name' :'Apple', 'username' : test, 'price' : 1.73, 'quantity' : 4, 'url' : 'http ://myImageOfApple.com' }

The API uses differents fonction, specific to their needs. We wrote four differents functions in order to satisfy the API : AddToBasket,getBasket, removeFromBasket, removeAllBasket. The first one calls the catalog microservice to get differents informations about the specific product, and fetch the old basket of the user. The second-one fetch the basket of a user, the third one remove a specific item from an user's basket, and the last one completely remove all the items present in the basket of a specific user.

DELETE :
To remove a specific item from the basket the path is url :[DBPORT]/shopping-kart/username/name. 'name' is the name of the product to remove, and username is the name of the user.
To clear the basket of a specific user, the path is url :[DBPORT]/shopping-kart/username.

2.9.3 Cart-history

For this database, there is a POST GET and DELETE method implemented in the API. The post method try to get the database called in order to update it. If the document is not created the method create a new document and insert it into the couchdb database. The GET method returns the document into the database and the delete methods remove an item in each databases.
Each method is called by the API with a specific path.
GET :
To get the basket history of an user, the path is {url :[DBPORT]/history/ :name}.
POST :
To post a new shopping kart : {url :[DBPORT]/history/ :name}

The post method call a function which get the shopping cart of the user corresponding to the name given in argument then put in the db in the user part the entire cart and a list of each items bought with the quantity updated at each purchase.

2.9.4 Logs

For this database, there is a POST GET and DELETE method implemented in the API. The post method try to get the database called in order to update it. If the document is not created the method create a new document and insert it into the couchdb database. The GET method returns the document into the database and the delete methods remove an item in each databases.
Each method is called by the API with a specific path.
GET :
To get the logs of a certain type(id,user,catalog,recommendation), the path is {url :[DBPORT]/logs/ :type}.
POST :
In the database each document correspond to a different log so there is a post path for each type of logs.
To post a new user : {url :[DBPORT]/logs/user} with a data containing "name" :${user} This will add the user in a list of all users registered on the website.
To post a new id : {url :[DBPORT]/logs/id} with a data containing "id" :${number} This will compare the id to post with the actual id. If the id to post is greater than the actual one it will update the id in the database
To post a new product : {url :[DBPORT]/logs/product} with a data containing {name, price, image , category, id} This will add or update the product in the document
To post a new user's purchase history : {url :[DBPORT]/logs/recommendation} with a data containing the user, the item to add, a list with the items purchased and a list with the quantity of each items purchased.
To post a new purchase history : {url :[DBPORT]/logs/recommendation2} with a data containing the item to add with a list of the items purchased and a list of the quantities.
The history is matching for a specific user or in general, an item with his name, a list of product purchased with this item and the quantity of each product. The two lists are updated at each post when the item is already in the history.

2.9.5 Views

The views are created in the logs database. The API handles the get call of those views. There are two views which are very similar. One is suppose to create recommendation specific to a specific user and the other is creating a general recommendation. Each method is called by the API with a specific path.
GET :
To get the view for the user's recommendation the path is {url :[DBPORT]/logs/view1}
To get the view for general recommendation the path is {url :[DBPORT]/logs/view2}
There is no POST or DELETE concerning the views.

3 Improvement on first deliverable

In the first part of the project the catalog service was creating 3 different databases in order to use it in the other services. With the creation of logs service we could remove 2 databases. Now the catalog service consist only on the creation of the format database.
We also improved the front end , especially concerning the purchase system. Now we can add items in the cart and purchase them. The calls to the different databases are directly made by the website by using axios. We also modified the back end calls in the back end. Indeed we were using nano to get some information from the different databases which was not secure at all. Now we only use axios the proceed micro-services calls.

4 Front-end

The Front-end of the website is partially implemented and linked to the back-end. When you enter the website all the catalog is fetched from the back-end and displayed. You can click on each items to have a little overview and some general recommendations for the item if they exists. You can register or log in to add some products in your shopping cart. they will be added correctly to the cart history. You can also remove them from your cart. The items that are already in your shopping-cart will not be displayed in the recommendations. When you want to finalize your purchases you can click on proceed to checkout and then confirm. If you disconnect and reconnect you will keep your shopping cart with its content. When you finish the checkout you will be redirected on the main page but the recommendations will be updated. If you click on an item that you've already bought you will see the items that the user is used to buy with this item but if the user choose an item that he never bought he will see the suggestions made by other clients. Sometimes the recommendation takes a bit of time to appear. It is due to the fact that we had some concurrency problems so we had to wait for an automatic update and it may takes few seconds so you have to stay on the quick view page and wait the apparition of the recommendation. If you connect as admin you will be redirected on the admin page but this one is not functional because it is not linked to the back-end.

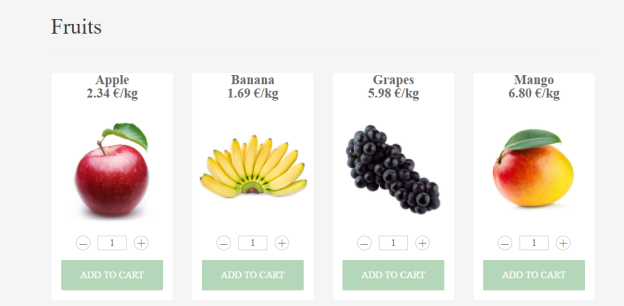


FIGURE 2 – homepage view

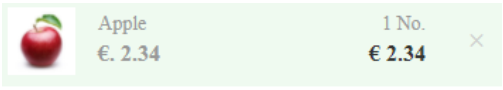
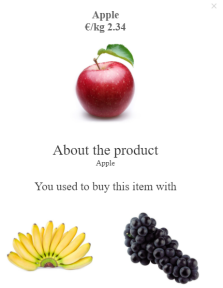


FIGURE 3 – cart view

((a)) General recommendations



((b)) Personalized recommendations

FIGURE 4 – recommendations

5 Conclusion

During this project we learned a lot about the cloud and the conception of REST API. We are now able to produce our own API and display it on a cloud website with manager and workers node. Unfortunately we ran out of time so we did not finish the full implementation of the front end with the admin functions. We also didn't handle the image upload with the blob storage because we had a lots of problems with versions of node and we couldn't resolve those problems so in the continuity of this project we could focus on those points in order to provide a fully finish version of the website. Tough, we are proud of our work and what we learned.