École Polytechnique de Louvain

LINFO2145 Cloud computing

Report : Cloud Computing Project

Second deliverable: recommendation service

Members

Joachim Renard - 67952200
Maxime Delacroix - 31632000

Group

19

Contents

1 List of Services

1.1 Frontend Service

Role of the Service: The Frontend service is designed to provide a responsive and intuitive user interface, essential for engaging user interaction and a seamless user experience.

Associated Technologies:

- **Svelte:** A JavaScript framework for building dynamic and efficient user interfaces
- **TypeScript:** Enhances JavaScript with type safety, leading to more robust and maintainable code.
- **Axios:** Facilitates efficient HTTP requests to the backend, ensuring smooth data exchange.
- **Docker:** Provides a consistent environment for development and deployment, ensuring application stability across various platforms.

1.2 Backend Microservices

Each microservice in the backend plays a crucial role in the application's overall functionality.

Common Technologies Across Backend Microservices:

- **Express:** Provides the framework for building each microservice's API.
- **JWT:** Ensures secure authentication and session management across services.
- **CouchDB:** Used for storing various types of data, from user credentials to product and order details, etc.
- **Nano:** Facilitating seamless interaction with the database.
- **Docker:** Facilitates consistent deployment and scaling of microservices.

1.2.1 Users Service

Role of the Service: Manages user authentication, ensuring secure access to the application.

1.2.2 Products Service

Role of the Service: Manages the application's product inventory, including creation, retrieval, update, and deletion of product data.

1.2.3 Orders Service

Role of the Service: Handles all aspects of order processing, from creation to retrieval.

1.2.4 Shopping-Carts Service

Role of the Service: Manages the shopping cart functionality, including adding, updating, and deleting cart items.

1.2.5 Recommendations Service

Role of the Service: The Recommendations Service is tasked with providing personalized recommendations to each user for every product in the shop. These recommendations are based on the overall purchase history and the specific buying patterns of the user. The service also considers factors like frequently bought together items and user's preferred product categories, enhancing the shopping experience through tailored suggestions.

Specific Associated Technologies:

- **node-cron:** Utilized for scheduling periodic tasks within the Recommendations Service. This tool is instrumental in automating the process of generating daily product recommendations, ensuring that users receive timely and relevant suggestions based on their preferences and purchasing history.

1.2.6 Logging Service

Role of the Service: Captures and stores logs from various services, aiding in debugging and monitoring.

1.3 Complete API

1.3.1 Frontend Service

Note: The Frontend Service interacts with backend services to facilitate user interface functionality.

1.3.2 Users Service API

- **POST /user**: Creates a new user.
  Request body: { username: [string], password: [string] }.
  Response: { status: [string], token: [string] } (token valid for 14 days).
- **GET /user/:username/:password**: Authenticates a user and retrieves their information.
  Response: { status: [string], data: { token: [string], role: [string] } }.

1.3.3 Orders Service API

- **POST /orders**: Creates a new order for the authenticated user.
  Request body: { items: [array of { id: [string], quantity: [number], extras: [object], totalQuantity: [number], totalPrice: [number], date: [string] }] }.
  Response: { status: [string], message: [string] }.
- **GET /orders**: Retrieves all orders associated with the authenticated user.
  Response: { status: [string], orders: [array of order objects] }.

1.3.4 Shopping-Carts Service API

- **POST /cart**: Creates a new cart for the authenticated user.
  Request body: { items: [array of { id: [string], quantity: [number] }] }.
  Response: { status: [string], _id: [string], _rev: [string], items: [array of products] }.
- **GET /cart**: Retrieves the cart of the authenticated user.
  Response: { status: [string], cart: [cart object] }.

- **DELETE /cart**: Deletes the entire cart of the authenticated user.
  Response: { status: [string], message: [string] }.
- **DELETE /cart/:itemId**: Deletes a specific item from the user's cart.
  Response: { status: [string], message: [string] }.

1.3.5 Products Service API

- **GET /products**: Retrieves all products.
  Response: { status: [string], data: [array of product objects] }.
- **GET /products/id**: Retrieves specific products by ID.
  Request: { productsId: [array of string] }.
  Response: { status: [string], data: [array of product objects] }.
- **POST /products**: (Admin only) Creates a new product.
  Request body: { product: { name: [string], price: [number], image: [string], category: [string] } }.
  Response: { status: [string], data: [product object] }.
- **PUT /products**: (Admin only) Updates a product.
  Request body: { product: [product object with updates] }.
  Response: { status: [string], data: [product object] }.
- **DELETE /products**: (Admin only) Deletes a product.
  Request body: { product: { _id: [string], _rev: [string] } }.
  Response: { status: [string], message: [string] }.

1.3.6 Recommendations Service API

- **GET /launchDailyRecommendations**: Force daily generation of recommendations. This route is purely useful for testing development and is intended to be removed in production.
  Response: On success, returns { status: 'success', message: "Daily recommendations generated" }. On error, returns { status: 'error', message: [string] }
- **GET /recommendations/:productId**: Retrieves personalized recommendations for a given product.
  Request: { productId: [string] }.
  Response: On success, returns { status: 'success', data: [array of recommended products] }. On error, returns { status: 'error', message: [string] }

1.3.7 Logging Service API

- **POST /logger/:name_of_the_service/info**: Logs an informational message from the specified service.
  Request body: { message: [string], data: [object], userId: [string, optional] }.
  Response: { status: [string], message: [string] }.
- **POST /logger/:name_of_the_service/error**: Logs an error message from the specified service.
  Request body: { message: [string], data: [object], userId: [string, optional] }.
  Response: { status: [string], message: [string] }.
- **GET /logger/user/info/:username**: Retrieves all informational logs for a specific user.
  Response: { status: [string], data: [array of log objects] }.

1.4 List of items logged in the logging service

The logging service is store inside multiple databases (one container with multiple database). Each service has it's own database that logs all the request done by the service, see subsection 1.3. All the requests answer by the services are sent to the logging service with different informations:

- **level:** If the entry is a info or a error
- **message:** A message that summarize the entry information, for example: Product created, Product sent, User logged in...
- **userID:** The id of the user that made the request, only use when the user need to authenticate for a action, such as admin actions, otherwise anonymous.
- **timestamp:** The timestamp of the request
- **data:** Data specific to the request
- **request_time_ms:** The time the service took to answer the request from the user sending the request to the response

2 Deployment

2.1 Overview

Scalable Application: This project leverages Docker containers and a Makefile to simplify the build process and ensure consistency across environments.

2.1.1 Getting Started:

Instructions provided here will help set up the project for development and testing purposes.

Note: If you are using Windows the .sh files are written in CRLF and not in LF. You have to change the line endings in VSCode by clicking on the CRLF in the bottom right corner and selecting LF for every .sh files.

Prerequisites:

- Docker
- GNU Make

2.2 Build Process

2.2.1 Building the Images

Important: Before running the swarm on azure or on your local VMs you need to run one of these two command to update the scapp.yml file, do not forget to set the variables inside the makefile or in your bash environment:

```
# To update the ip address for local use, do not forget to set the ip in the
    make file
make update-service-url-local

# To update the ip address for azure use, do not forget the set the domain name
    of the azure VM
make update-service-url-azure
```

To build all Docker images in the project, execute:

```
# To build the images
make full-build

# To push the images to docker hub
make full-push

# To do both
make full
make
```

Alternatives:

```
make backend-build
make backend-push

make frontend-build
make frontend-push
```

2.3 Swarm Management

2.3.1 Creating the Swarm

Initialize the swarm with:

```
make swarm-init
```

2.3.2 Running the Swarm

Deploy the swarm using:

```
make swarm-deploy
or
make swarm-deploy-with-scaling
```

Reload the swarm with:

```
make swarm-reload
or
make swarm-reload-with-scaling
```

2.3.3 Stopping the Swarm

To stop the swarm and the scaling services (if any):

```
make swarm-remove
```

2.3.4 Leaving the Swarm

To leave the swarm:

```
make swarm-leave
```

2.4 Scalability and Testing

2.4.1 Scalability Approach

Scalability is achieved through monitoring CPU usage and dynamically scaling services. Services are replicated three times when the demand is high and is scaled down when the demand is low.

2.4.2 Testing Scalability

To test scalability using workload injection at the front-end level :

```
make elastic-scaling-users-daemon-test
make elastic-scaling-orders-daemon-test
make elastic-scaling-products-daemon-test
make elastic-scaling-shopping-carts-daemon-test
make elastic-scaling-recommendations-daemon-test
```

Important: Start scalability processes before testing:

```
make swarm-deploy-with-scaling
or
make launch-elastic-scaling
\end{verbatim}
To stop scaling processes:
\begin{lstlisting}[language=bash]
make stop-elastic-scaling
```

3 Recommendations

The Recommendations service in our application is designed to provide personalized product suggestions for each user using CouchDB's Map/Reduce functionality to efficiently process and analyze large datasets.

3.0.1 Recommendations Generation

The recommendation generation process is multifaceted, taking into account not only user purchases but also their current shopping cart contents. This dual approach ensures that the recommendations are not only based on historical data but are also dynamically responsive to the user's immediate shopping context.

To facilitate this recommendation mechanism, a scheduled task, specifically a cron job using node-cron, plays a crucial role. This cron job is programmed to execute daily at 03:00 AM. During this time, the system generates personalized recommendations for each user and for each product in the recommendations database.

A small note on this scheduled task: if the elastic scaling is launch and decides to create several recommendations micro-services, there will be several scheduled tasks doing exactly the same job, which is a loss of efficiency that should be corrected in future versions [1].

Use of Thresholds in Orders Data Analysis Several thresholds are employed to fine-tune the recommendation process:

- **Frequency Threshold:** Determines how often a product needs to be purchased before it's considered popular enough for recommendation.
- **Bought Together Threshold:** Assesses the frequency with which products are purchased together, allowing the service to identify items that are often purchased in the same order.
- **Category Preference Threshold:** Analyzes the frequency of purchases within specific categories, helping to tailor recommendations based on the user's category preferences.

[1] We wanted to use zookeeper to coordinate the different service so only one of them do the cronjob, but this approach is not worth it for only one service to coordinate, because it means we have to run another container

These thresholds are very low for the moment, as we are in a test environment, and will have to be adapted in the future when the application goes into production, depending on the number of users and shopping orders.

3.1 Principles and Implementation of Our Map/Reduce Queries

3.1.1 Map/Reduce Query for Product Frequency

Purpose: The ‘productsFrequency‘ view is designed to calculate the frequency of each product purchased by users. This data forms the foundation of understanding popular products and user-specific buying patterns.

Map Function: The map function iterates over each order. For every item in an order, it emits a composite key comprising the user ID and the product ID, along with the quantity of the product ordered. This structure allows us to track how often each product is purchased by individual users.

```
map: function (doc) {
    doc.items.forEach(function (item) {
        emit([doc.userId, item._id], item.quantity);
    });
}
```

Reduce Function: The reduce function sums up the quantities of each product for each user, giving us a total purchase frequency.
We are utilizing the built-in ’_sum’ reduce function of CouchDB, which employs the rereduce mechanism efficiently, making it optimal for handling large datasets.

```
reduce: '_sum'
```

3.1.2 Map/Reduce Query for Frequently Bought Together

Purpose: This view helps in identifying products that are frequently purchased together, which is a critical aspect of our recommendations.

Map Function: The map function processes each order and creates pairs of products that were bought together within the same order.

```
map: function (doc) {
    var sortedItems = doc.items.map(item => item._id).sort();
    for (var i = 0; i < sortedItems.length; i++) {
        for (var j = i + 1; j < sortedItems.length; j++) {
            emit([doc.userId, sortedItems[i], sortedItems[j]], 1);
            emit([doc.userId, sortedItems[j], sortedItems[i]], 1);
        }
    }
}
```

Reduce Function: It sums the counts for each pair of products, providing insights into how often two products are purchased together. We are using the built-in ’_sum’ reduce function of CouchDB aswell.

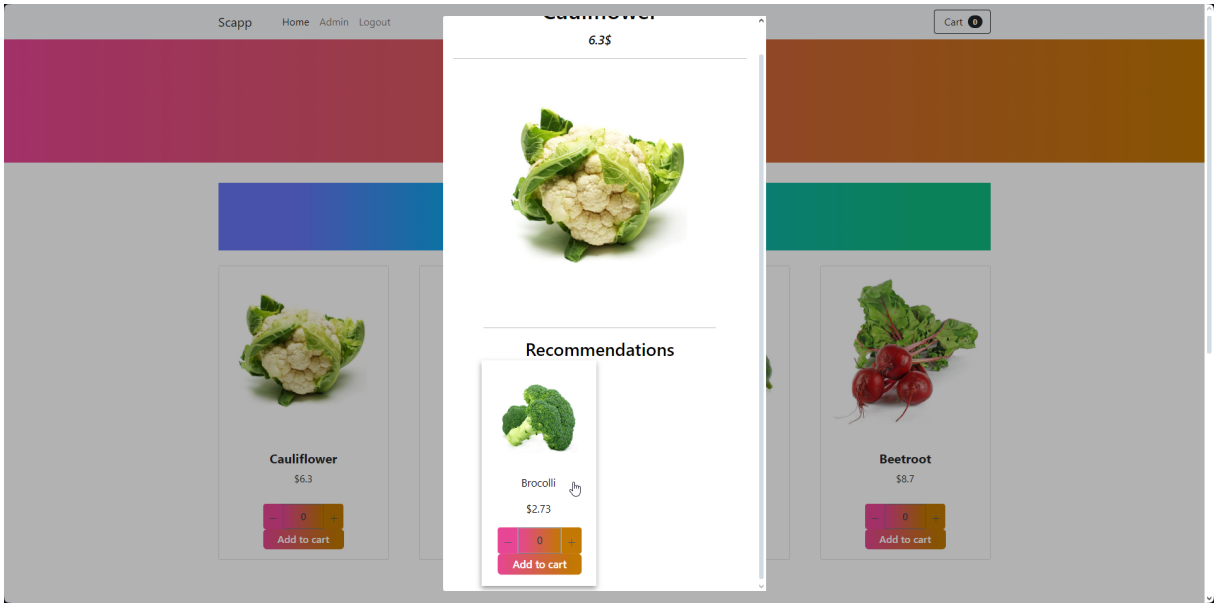
```
reduce: '_sum'
```

The aggregated data from these Map/Reduce queries allows the Recommendations service to create a nuanced understanding of user behavior. By analyzing product frequencies and combinations of frequently bought items, the service can tailor recommendations to align closely with each user’s preferences and current shopping trends.

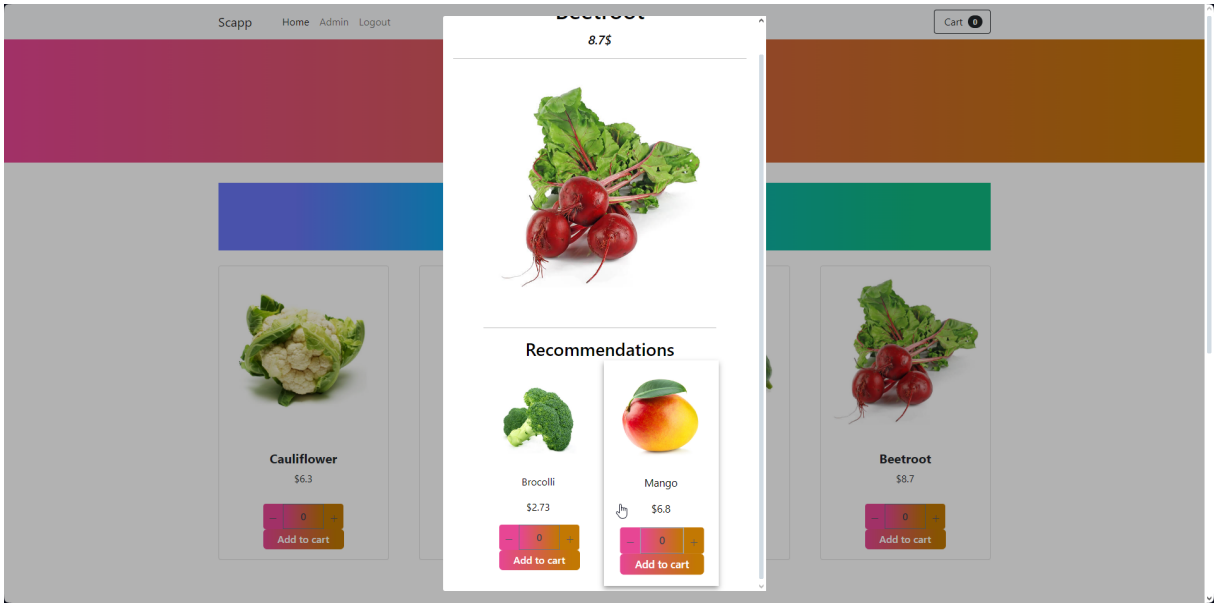
3.2 Screenshots in action

Here are a few examples of recommendations.

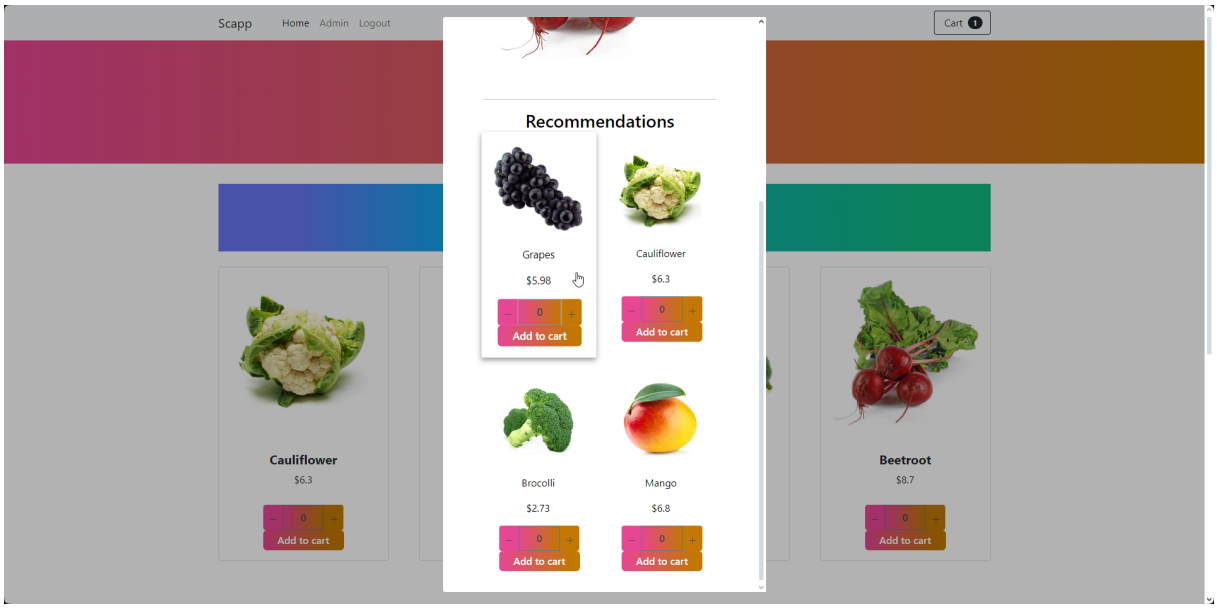
- Let's imagine that broccoli is a hit on the app, and that the vegetables category is also a hit for the user in question, so when he clicks on cauliflower, broccoli will be recommended.



- Now let's imagine that in addition to the previous example, there's a trend in the application that when a user buys beetroot, he also buys mangoes. In this case, the mango will be recommended when a user clicks on the beetroot.



- As a final example, let's take the previous ones and add that cauliflowers are a hit and also that there's a trend that when users buy apples, they buy grapes. Since the user in question has apples in his basket, grapes will be offered in addition to the other recommendations.



Of course, there are plenty of other examples that could be presented, such as the fact that if a recommended product is already in the shopping cart, it will no longer be shown, etc.
As you can see, our recommendation system gives users the opportunity to add their recommended products directly to the shopping cart, making the purchase as simple as possible. We've also set a limit of 20 recommended products so as not to overwhelm the user. This limit is, of course, easily modifiable. Finally, we've introduced a few animations to make the UX experience more enjoyable.
If you want to test the recommendations on your side, we have set the global thresholds to these values:

- thresholdFrequency = 30
- thresholdBoughtTogether = 10,
- thresholdCategoryPreferences = 40

and the thresholds for a specific user:

- thresholdFrequency = 5
- thresholdBoughtTogether = 2
- thresholdCategoryPreferences = 5

This means that to create a cauliflower recommendation, your user must purchase at least 6 to activate the recommendation by category and frequency. These cauliflowers will then be recommended to the other vegetables. If you want to activate the frequentlyBougthTogether recommendation, you need to buy several products at least 3 times at the same time.
If you buy 31 beetroots and the vegetables category has at least 41 total purchases in the application, beetroots can be recommended to other users for all vegetables.
Don't forget to force the launch of daily recommendations after buying products, because in a test environment there's no point in waiting until 3 a.m. or letting services run !

4 Azure

4.1 Prerequisites

For the azure deployment, there are a few prerequisites. You need to have your azure VMs setup with a manager of the swarm and the network interface setup. In the makefile, you need

to specify 3 variables, the docker id which is the id of your docker account, the azure manager address which is the dns address of your azure VM manager and the user name of the admin in the VM. You might need to give your password, but it is recommended to have your ssh keys setup.

4.2 Azure deployment

To deploy the app on azure you need to run this command, it will update the url in the scapp.yml file which will be set to the azure manager ip and send the scapp.yml file to the VM.

```
make azure-push
```

Finally to deploy to application:

```
# Remove the old swarm and deploy the new one
make azure-deploy

# execute all of the above, azure-push and azure-deploy and build the entire
    project with make full
make azure
```

4.3 Azure blob

All the images are store inside an azure blob. The base images are already in the blob when the user wants to add a product the image url is sent to the product service, which will download the image and sent it to the blob and then finaly store the product with the new image uri (from the azure blob). So even though the image came from another server it will be saved forever inside our azure blob.