UNIVERSITÉ LIBRE DE BRUXELLES
Université Catholique de Louvain

Project report

UCLouvain: LINFO2145 - Cloud Computing
ULB: INFO-Y025 Cloud computing

Samir Azmar

November 30, 2024

Contents

1 Context

My task is to design and implement the backend for a shopping-cart application (SCApp). The frontend, created with Svelte, is functional but relies solely on local storage for browsing items and managing the cart and checkout. This is not suitable for real-world use, as critical data like product details, availability, and user activity must be managed dynamically in the Cloud.

My role involves developing cloud-based micro services to support the backend. These services will dynamically provide item data (e.g., characteristics, photos, availability) and store user activity, such as cart contents, in the Cloud. This backend will integrate with the existing frontend to create a fully functional SCApp.

I encountered an obstacle while preparing to deploy the backend for the shopping-cart application on Microsoft Azure. Due to a Cross-Origin Resource Sharing (CORS) issue during deployment, the management team has decided that, for the initial phase, the backend should be deployed locally using 2 VMs and swarm. This adjustment allows development to proceed without delays, while a resolution for the CORS issue is being investigated.

2 What's implemented

The backend:

- Use sets of microservices (RESTful APIs over HTTP)
- Each microservice have its own database
- Have Each microservices packaged as a Docker container and use the Docker network.

An authenticated user can:

- Add/Remove items to/from a shopping cart.
- Perform a purchase (checking out)
- Can check his purchases history even after logging out/leaving the application.
- Can see the content of his cart even after logging out/leaving the application.

An administrator can:

- Use an interface to update an item (price, description). Refresh the tab to see the updated item.
- Use the interface to add new items (description, photo,..) stored in an Azure Blob. Refresh the tab to see the added item.

Logging:

- Other microservice will asynchronously send logs to the log service, which will store them in a CouchDB database as JSON objects.
- Logs user actions.
- Logs performance measurement.
- Each microservice has its own table in the logging database.

API gateway: There is an API gateway in the backend named api-gateway.

Scalability: Users, Cart, Checkout, Product and logs microservice are scalable using the "test_scalability.sh" script.

Persistent authentication: The user is permanently logged on the tab. The token is stored in the session of the tab.

Bearer Authentication: To send a request to the backend, the header must contain the Authentication. Without Authentication in the header, the request is rejected.

Authorization verification: Product, Cart, and Checkout microservices are calling the users microservice to verify the authentication token provided by the user in the Authentication header.

3 List of services

3.1 Users microservice

Role:

This microservice handles user authentication and management. It provides endpoints for user registration, login, token verification, logout, and user deletion.

Technologies:

The microservice uses the following technologies:

- **Node.js**
- **Express.js**
- **Apache CouchDB**
- **JWT**
- **Docker**

How to build the container:

The command to build this container is found in the script **build_containers.sh** that builds all containers.

How to run a swarm of instances of the container:

The file **init_swarm.sh** creates a swarm and deploy all containers of my microservice from **scapp.yml**.

Complete API of the service:

- **Register a New User**
  - **Endpoint:** `POST /register`
  - **Description:** Creates a new user account.
  - **Request Body:**
    - `username` (string, required): The username for the new user.
    - `password` (string, required): The password for the new user.
    - `isAdmin` (boolean, optional): Indicates if the user has admin privileges.
  - **Response:**
    - **200 OK:** Registration successful. Returns:
      - `status` (string): `"success"`
      - `token` (string): The authentication token for the new user.
    - **409 Conflict:** Registration failed. Returns:
      - `status` (string): `"error"`

        - `message` (string): Error details.
    - **Additional Actions:**
        - Logs the performance for debugging purposes.

- **User Login**
  - **Endpoint:** `POST /login`
  - **Description:** Authenticates a user and provides an authentication token.
  - **Request Body:**
    - `username` (string, required): The user's username.
    - `password` (string, required): The user's password.
  - **Response:**
    - **200 OK:** Login successful. Returns:
      - `status` (string): `"success"`
      - `token` (string): The user's authentication token.
      - `isAdmin` (boolean): Indicates if the user has admin privileges.
    - **404 Not Found:** Login failed. Returns:
      - `status` (string): `"error"`
      - `message` (string): Error details.
    - **Additional Actions:**
      - Logs the performance for debugging purposes.

- **Verify Authentication Token**
  - **Endpoint:** `GET /verify/:token`
  - **Description:** Validates an authentication token and retrieves user details.
  - **URL Parameter:**
    - `token` (string, required): The token to be verified.
  - **Response:**
    - **200 OK:** Verification successful. Returns:
      - `status` (string): `"success"`
      - `...userData`: Details of the authenticated user (e.g., username, roles).
    - **401 Unauthorized:** Verification failed. Returns:
      - `status` (string): `"error"`
      - `message` (string): Error details.
    - **Additional Actions:**
      - Logs the performance for debugging purposes.

- **User Logout**
  - **Endpoint:** `POST /logout`
  - **Description:** Ends the user session.
  - **Request Body:**
    - `username` (string, required): The username of the user logging out.

    - `token` (string, required): The user's authentication token.
  - **Response:**
    - **200 OK:** Logout successful. Returns:
      - `status` (string): `"success"`
    - **401 Unauthorized:** Logout failed. Returns:
      - `status` (string): `"error"`
      - `message` (string): Error details.
    - **Additional Actions:**
      - Logs the performance for debugging purposes.

- **Delete a User**
  - **Endpoint:** `DELETE /delete/:username`
  - **Description:** Removes a user account from the system.
  - **URL Parameter:**
    - `username` (string, required): The username of the user to be deleted.
  - **Response:**
    - **200 OK:** Deletion successful. Returns:
      - `status` (string): `"success"`
    - **401 Unauthorized:** Deletion failed. Returns:
      - `status` (string): `"error"`
      - `message` (string): Error details.
    - **Additional Actions:**
      - Logs the performance for debugging purposes.

3.2 Cart microservice

Role:

The role of this microservice is to manage user cart operations. It provides endpoints for saving, retrieving, and removing items from a user's cart.

Technologies:

The microservice uses the following technologies:

- **Node.js**
- **Express.js**
- **Apache CouchDB**
- **Docker**

How to build the container:

The command to build this container is found in the script **build_containers.sh** that builds all containers.

How to run a swarm of instances of the container:

The file **init_swarm.sh** creates a swarm and deploy all containers of my microservice from **scapp.yml**.

Complete API of the service:

- **Save Cart Data**
  - **Endpoint:** `POST /cart`
  - **Description:** Saves the current cart data for a user.
  - **Request Headers:**
    - `Authorization` (string, required): Bearer token for authentication.
  - **Request Body:**
    - `username` (string, required): The username of the user.
    - `cart` (array, required): An array of items representing the cart contents.
  - **Response:**
    - **200 OK:** Cart saved successfully. Returns:
      - `status` (string): `"success"`
    - **500 Internal Server Error:** Saving the cart failed. Returns:
      - `status` (string): `"error"`
      - `message` (string): Error details.
  - **Additional Actions:**
    - Logs the last item added to the cart for tracking purposes.

    * Logs the performance for debugging purposes.

- **Retrieve Cart Data**
  - **Endpoint:** `GET /cart/:username`
  - **Description:** Retrieves the cart data for a specific user.
  - **Request Headers:**
    * `Authorization` (string, required): Bearer token for authentication.
  - **URL Parameters:**
    * `username` (string, required): The username of the user whose cart is being retrieved.
  - **Response:**
    * **200 OK:** Cart retrieved successfully. Returns:
      · `status` (string): `"success"`
      · `cart` (array): The user's cart data.
    * **500 Internal Server Error:** Retrieving the cart failed. Returns:
      · `status` (string): `"error"`
      · `message` (string): Error details.
    * **Additional Actions:**
      · Logs the performance for debugging purposes
- **Save Removed Items**
  - **Endpoint:** `POST /cart/remove`
  - **Description:** Saves an item that was removed from the user's cart.
  - **Request Headers:**
    * `Authorization` (string, required): Bearer token for authentication.
  - **Request Body:**
    * `username` (string, required): The username of the user.
    * `item` (object, required): Details of the item removed from the cart.
  - **Response:**
    * **200 OK:** Removed item saved successfully. Returns:
      · `status` (string): `"success"`
    * **500 Internal Server Error:** Saving the removed item failed. Returns:
      · `status` (string): `"error"`
      · `message` (string): Error details.
  - **Additional Actions:**
    * Logs the removed item for tracking purposes.
    * Logs the performance for debugging purposes.

3.3 Checkout microservice

Role:

The role of this microservice is to manage the checkout process. It provides endpoints for saving and retrieving checkout data for users.

Technologies:

The microservice uses the following technologies:

- **Node.js**
- **Express.js**
- **Apache CouchDB**
- **Docker**

How to build the container:

The command to build this container is found in the script **build_containers.sh** that builds all containers.

How to run a swarm of instances of the container:

The file **init_swarm.sh** creates a swarm and deploy all containers of my microservice from **scapp.yml**.

Complete API of the service:

- **Save Checkout Data**
  - **Endpoint:** `POST /checkout/save`
  - **Description:** Saves the checkout data for a user.
  - **Request Headers:**
    - `Authorization` (string, required): Bearer token for authentication.
  - **Request Body:**
    - `username` (string, required): The username of the user.
    - `checkout` (object, required): Details of the checkout data to be saved.
  - **Response:**
    - **200 OK:** Checkout data saved successfully. Returns:
      - `status` (string): `"success"`
    - **500 Internal Server Error:** Saving checkout data failed. Returns:
      - `status` (string): `"error"`
      - `message` (string): Error details.
  - **Additional Actions:**
    - Logs the checkout data for tracking purposes.

    * Logs the performance for debugging purposes.

- **Retrieve Checkout Data**
  - **Endpoint:** `GET /checkout/load/:username`
  - **Description:** Retrieves all saved checkout data for a specific user.
  - **Request Headers:**
    * `Authorization` (string, required): Bearer token for authentication.
  - **URL Parameters:**
    * `username` (string, required): The username of the user whose checkout data is being retrieved.
  - **Response:**
    * **200 OK:** Checkout data retrieved successfully. Returns:
      · `status` (string): `"success"`
      · `checkouts` (array): The list of checkout data for the user.
    * **500 Internal Server Error:** Retrieving checkout data failed. Returns:
      · `status` (string): `"error"`
      · `message` (string): Error details.
  - **Additional Actions:**
    * Logs the retrieval action for tracking purposes.
    * Logs the performance for debugging purposes.

3.4 Products microservice

Role:

The role of this microservice is to manage product-related operations. It provides endpoints for creating, updating, deleting, and retrieving products.

Technologies:

The microservice uses the following technologies:

- **Node.js**
- **Express.js**
- **Apache CouchDB**
- **Docker**
- **Azure Blob**
- **multer**

How to build the container:

The command to build this container is found in the script **build_containers.sh** that builds all containers.

How to run a swarm of instances of the container:

The file **init_swarm.sh** creates a swarm and deploy all containers of my microservice from **scapp.yml**.

Complete API of the service:

- **Save a New Product**
  - **Endpoint**: `POST /product/save`
  - **Description**: SSaves a new product to the database and asynchronously uploads its image.
  - **Request Body**:
    - `name` (string, required): Name of the product.
    - `price` (number, required): Price of the product.
    - `category` (string, required): Category of the product.
    - `url` (string, null): URL of the product image.
    - `image` (file, required): Image file to be uploaded.
  - **Response**:
    - **200 OK**:
      - `status` (string): `"success"`
      - `product` (object): The saved product details.

    - **409 Conflict**:
        - `status` (string): `"error"`
        - `message` (string): Error details.
  - **Additional Actions:**
    - Logs the product details for tracking purposes.
    - Asynchronously uploads the product image to Azure Blob Storage.
    - Logs the performance for debugging purposes.

- **Update an Existing Product**
  - **Endpoint**: `PUT /product/update/:id`
  - **Description**: Updates details of an existing product.
  - **URL Parameters**:
    - `id` (string, required): ID of the product to update.
  - **Request Body**:
    - `name` (string, optional): Updated name of the product.
    - `price` (number, optional): Updated price of the product.
    - `category` (string, optional): Updated category of the product.
    - `url` (string, optional): Updated URL of the product image.
    - `image` (file, optional): Updated image file to be uploaded.
  - **Response**:
    - `200 OK`:
      - `status` (string): `"success"`
      - `product` (object): The updated product details.
    - `409 Conflict`:
      - `status` (string): `"error"`
      - `message` (string): Error details.
  - **Additional Actions:**
    - Logs the update operation for tracking purposes.
    - Logs the performance for debugging purposes.

- **Delete a Product**
  - **Endpoint:** `DELETE /product/delete/:id`
  - **Description:** Deletes an existing product.
  - **Request Headers:**
    - `Authorization` (string, required): Bearer token for authentication.
  - **URL Parameters:**
    - `id` (string, required): ID of the product to delete.
  - **Response:**
    - **200 OK:** Product deleted successfully. Returns:
      - `status` (string): `"success"`

        - `message` (string): `"Product deleted"`
    - **409 Conflict:** Deleting the product failed. Returns:
        - `status` (string): `"error"`
        - `message` (string): Error details.
  - **Additional Actions:**
    - Logs the delete operation for tracking purposes.
    - Logs the performance for debugging purposes.
    - Asynchronously deletes the product image from Azure Blob Storage.

- **Get All Products**
  - **Endpoint:** `GET /product`
  - **Description:** Retrieves all available products.
  - **Response:**
    - **200 OK:** Products retrieved successfully. Returns:
        - `status` (string): `"success"`
        - `products` (array): List of all products.
    - **500 Internal Server Error:** Retrieving products failed. Returns:
        - `status` (string): `"error"`
        - `message` (string): Error details.
  - **Additional Actions:**
    - Logs the performance for debugging purposes.

3.5 Logs microservice

Role:

The role of this microservice is to handle logging operations. It provides endpoints to save performance logs for microservices and to save user action logs. This helps in monitoring, auditing, and debugging the application by keeping track of performance metrics and user activities.

Technologies:

The microservice uses the following technologies:

- **Node.js**
- **Express.js**
- **Apache CouchDB**
- **Docker**

How to build the container:

The command to build this container is found in the script **build_containers.sh** that builds all containers.

How to run a swarm of instances of the container:

The file **init_swarm.sh** creates a swarm and deploy all containers of my microservice from **scapp.yml**.

Complete API of the service:

- **Save Performance Logs for a Microservice**
  - **Endpoint:** `POST /logs/microservice/:microservice`
  - **Description:** Saves performance logs for a specified microservice.
  - **URL Parameters:**
    - `microservice` (string, required): Name of the microservice for which logs are being saved.
  - **Request Body:**
    - `logs` (object, required): Performance log data to be saved.
  - **Response:**
    - **200 OK:** Logs saved successfully. Returns:
      - `status` (string): `"success"`
    - **500 Internal Server Error:** Saving logs failed. Returns:
      - `status` (string): `"error"`
      - `message` (string): Error details.
- **Save User Action Logs**

- **Endpoint:** `POST /logs/user-action`
- **Description:** Saves logs for user actions across microservices.
- **Request Body:**
  - `logs` (object, required): User action log data to be saved.
- **Response:**
  - **200 OK:** Logs saved successfully. Returns:
    - `status` (string): `"success"`
  - **500 Internal Server Error:** Saving logs failed. Returns:
    - `status` (string): `"error"`
    - `message` (string): Error details.

3.6 Api-gateway

Role:

The role of this microservice is to act as an API Gateway. It routes incoming HTTP requests to the appropriate backend microservices. This API Gateway simplifies client interactions with multiple microservices by providing a single entry point. This API gateway is currently in version 1.

Technologies:

The microservice uses the following technologies:

- **Node.js**
- **Express.js**
- **Docker**
- **http-proxy-middleware**

How to build the container:

The command to build this container is found in the script **build_containers.sh** that builds all containers.

How to run a swarm of instances of the container:

The file **init_swarm.sh** creates a swarm and deploy all containers of my microservice from **scapp.yml**.

Complete API of the service:

- **Save Product**
  - **Endpoint:** `POST /api/v1/product/save`
  - **Proxy to:** `/product/save` on `http://products-service:80`
  - **Description:** Proxies the request to save a product to the product service.
  - **Error Handling:** Returns `500 Internal Server Error` in case of proxying issues.
- **Get Products**
  - **Endpoint:** `GET /api/v1/product`
  - **Proxy to:** `/product` on `http://products-service:80`
  - **Description:** Proxies the request to fetch all products from the product service.
  - **Error Handling:** Returns `500 Internal Server Error` in case of proxying issues.
- **Update Product**
  - **Endpoint:** `PUT /api/v1/product/update`
  - **Proxy to:** `/product/update` on `http://products-service:80`
  - **Description:** Proxies the request to update a product in the product service.

  - **Error Handling:** Returns `500 Internal Server Error` in case of proxying issues.

- **Delete Product**
  - **Endpoint:** `DELETE /api/v1/product/delete`
  - **Proxy to:** `/product/delete` on `http://products-service:80`
  - **Description:** Proxies the request to delete a product from the product service.
  - **Error Handling:** Returns `500 Internal Server Error` in case of proxying issues.

Checkout Service Routes

- **Save Checkout**
  - **Endpoint:** `POST /api/v1/checkout/save`
  - **Proxy to:** `/checkout/save` on `http://checkout-service:80`
  - **Description:** Proxies the request to save checkout data to the checkout service.
  - **Error Handling:** Returns `500 Internal Server Error` in case of proxying issues.
- **Load Checkout**
  - **Endpoint:** `GET /api/v1/checkout/load`
  - **Proxy to:** `/checkout/load` on `http://checkout-service:80`
  - **Description:** Proxies the request to load checkout data from the checkout service.
  - **Error Handling:** Returns `500 Internal Server Error` in case of proxying issues.

Cart Service Routes

- **Save Cart**
  - **Endpoint:** `POST /api/v1/cart/save`
  - **Proxy to:** `/cart/save` on `http://cart-service:80`
  - **Description:** Proxies the request to save a cart for a user in the Cart Service.
  - **Error Handling:** Returns `500 Internal Server Error` in case of proxying issues.
- **Get Cart by Username**
  - **Endpoint:** `GET /api/v1/cart/:username`
  - **Proxy to:** `/cart` on `http://cart-service:80`
  - **Description:** Proxies the request to retrieve the cart data for a specific user identified by `username`.
  - **Error Handling:** Returns `500 Internal Server Error` in case of proxying issues.
- **Remove Item from Cart**
  - **Endpoint:** `DELETE /api/v1/cart/remove`
  - **Proxy to:** `/cart/remove` on `http://cart-service:80`
  - **Description:** Proxies the request to remove an item from the user's cart.
  - **Error Handling:** Returns `500 Internal Server Error` in case of proxying issues.

User Service Routes

- **User Registration**
  - **Endpoint:** `POST /api/v1/register`
  - **Proxy to:** `/register` on `http://users-service:80`
  - **Description:** Proxies the request to register a new user in the authentication service.
  - **Error Handling:** Returns `500 Internal Server Error` in case of proxying issues.
- **User Login**
  - **Endpoint:** `POST /api/v1/login`
  - **Proxy to:** `/login` on `http://users-service:80`
  - **Description:** Proxies the request to log in a user to the authentication service.
  - **Error Handling:** Returns `500 Internal Server Error` in case of proxying issues.
- **User Verification**
  - **Endpoint:** `GET /api/v1/verify/:token`
  - **Proxy to:** `/verify` on `http://users-service:80`
  - **Description:** Proxies the request to verify a user's authentication token.
  - **Error Handling:** Returns `500 Internal Server Error` in case of proxying issues.
- **User Logout**
  - **Endpoint:** `POST /api/v1/logout`
  - **Proxy to:** `/logout` on `http://users-service:80`
  - **Description:** Proxies the request to log out a user from the authentication service.
  - **Error Handling:** Returns `500 Internal Server Error` in case of proxying issues.
- **Delete User**
  - **Endpoint:** `DELETE /api/v1/delete/:username`
  - **Proxy to:** `/delete` on `http://users-service:80`
  - **Description:** Proxies the request to delete a user from the authentication service.
  - **Error Handling:** Returns `500 Internal Server Error` in case of proxying issues.

4 Technology justification

- Node.js
  - **Description**: A JavaScript runtime built on Chrome's V8 engine that allows developers to execute JavaScript on the server side.
  - **Justification**: It enables full-stack JavaScript development, offering non-blocking, event-driven architecture ideal for scalable and high-performance applications.
- Express.js
  - **Description**: A minimalist web application framework for Node.js used to build web applications and APIs.
  - **Justification**: Simplifies the creation of server-side applications by offering robust routing, middleware integration, and efficient request/response handling.
- Docker
  - **Description**: A platform that enables developers to build, ship, and run applications in isolated containers.
  - **Justification**: Provides consistency across environments, improves scalability, and simplifies deployment and dependency management by packaging applications with their runtime environment.
- http-proxy-middleware
  - **Description**: A middleware for Node.js that acts as a reverse proxy to forward HTTP requests to a different server or endpoint.
  - **Justification**: Ideal for development environments to avoid CORS issues and dynamically route requests without needing a separate server like Nginx. Its simplicity and easy configuration make it suitable for low-traffic applications.
- Apache CouchDB
  - **Description**: A NoSQL database that uses JSON to store data and JavaScript for querying and indexing, with multi-master replication and offline-first support.
  - **Justification**: Stores data in a JSON format, making it easy to work with modern web applications.
- Azure Blob
  - **Description**: A Microsoft Azure storage solution for managing unstructured data like images, videos, documents, and backups.
  - **Justification**: Offers scalable, secure, and cost-effective cloud storage for handling large volumes of unstructured data, with integration into Azure's ecosystem.
- multer
  - **Description**: A Node.js middleware for handling multipart/form-data, primarily used for uploading files.
  - **Justification**: Simplifies file upload processing in Node.js applications by parsing incoming files and making them accessible in the request object.

- JWT
  - **Description**: A compact, self-contained token format for securely transmitting information between parties as a JSON object.
  - **Justification**: Provides a secure and stateless authentication mechanism, commonly used for API authorization and session management.

5 List of logged items

Performance log:

- **timestamp**: The current date and time in ISO 8601 format when the performance data was recorded.
- **endpoint**: The specific endpoint of the request being measured.
- **duration**: The time taken to handle the request, in milliseconds.
- **statusCode**: The HTTP status code of the response.
- **requestSize**: The size of the incoming request in bytes, derived from the `Content-Length` header (or `0` if not present).
- **responseSize**: The size of the outgoing response in bytes, retrieved from the `Content-Length` header of the response (or `0` if not present).
- **serviceName**: The name of the microservice handling the request.
- **metadata**:
  - **queryParams**: The query parameters included in the request, captured from `req.query`.
  - **headers**: The headers from the incoming request, captured from `req.headers`.

User action log

- **timestamp**: The current date and time in ISO 8601 format when the action was logged.
- **username**: The name of the user performing the action.
- **action**: The specific action performed by the user.
- **endpoint**: The endpoint where the user action occurred.
- **details**: The information sent by the user in the body of the request.

6 Deployment

Before following one of these deployments, make sure to log in with your Microsoft Azure account. If you are not logged in, type in your terminal:

```
az login
```

Make sure to open a terminal in the project folder and that the terminal user owns the files.

6.1 Locally without VMs

1. If you do not possess a **group resource**, execute this command:

   ```
   $ az group create --name linfo2145.weu --location westeurope
   ```

2. Create an **Azure Blob** by executing this shell script `create_azure_blob.sh` with this command and follow the instructions:

   ```
   $ source create_azure_blob.sh
   ```

3. To build and run the docker files of each microservice, execute this command:

   ```
   $ ./build_and_run_containers.sh
   ```

6.2 Locally with 2 VMs + swarm

You can start from step 3 if you already created Azure blob from step 2 in section Locally without VMs

1. If you do not possess a **group resource**, execute this command:

   ```
   $ az group create --name linfo2145.weu --location westeurope
   ```

2. Create an **Azure Blob** by executing the shell script `create_azure_blob.sh` with this command and follow the instructions:

   ```
   $ source create_azure_blob.sh
   ```

3. Start 2 VMs by executing this command:

   ```
   $ ./init_vms.sh
   ```

4. Create a docker swarm by executing this command:

   ```
   $ ./init_swarm.sh
   ```

6.3 On Microsoft Azure

Since the first delivery is not required to run on Microsoft Azure. I did not provide a deployment script. I could have provided one, but I spent my time on another course.

However, I managed to make my backend run on Microsoft Azure without issue.

7 What is the required configuration on Microsoft Azure

None