**Car Parts eCommerce Shop**

Members: Dany Solh, Arman Nadjarian

Submitted on: 01/12/2024

**Table of Contents**

1. Introduction
   1. Overview
   2. Objectives
2. System Architecture
   1. Customers
   2. Inventory
   3. Reviews
   4. Sales
3. Implementation Details
   1. Customers
   2. Inventory
   3. Reviews
   4. Sales
4. Database Design
5. Error Handling and Validation
6. Testing
7. Deployment and Integration
8. Documentation and Profiling
9. GitHub and Version Control
10. User Authentication
11. Moderation

**I-** **Introduction**

1- Overview:

This project is designed as a backend system comprising a set of independent services that could be used for a variety of eCommerce shops. These services cover critical functionalities, including sales, inventory management, customer handling, and reviews operations.

The purpose of the project is to deliver a robust backend interface that ensures seamless integration with various frontend platforms, accommodating diverse client requirements.

The scope of the project includes the implementation of all essential backend components, from database creation scripts to comprehensive API endpoints. These APIs facilitate communication and ensure efficient data exchange between the services and the database, supporting the full range of operations for the eCommerce platform.

2- Objectives:

The main goal of this project is to build a backend system that can easily work with different frontend and backend systems by using JSON to transfer data.

Another key objective is to create a system that can handle errors effectively and keep running smoothly. To achieve this, we implemented error control mechanisms and tested all the services thoroughly using Postman for API testing and pytest for automated testing.

**II- System Architecture:**

1- Customers:

The customer service manages and updates the shop's user information. It allows new customers to register in the database, update their details, or be entirely removed from the records.

This service also includes features to retrieve information for either all customers or a specific customer. Additionally, it enables adding or deducting funds from a customer’s wallet.

2- Inventory:

The inventory service manages and updates the shop’s product and stocks information. It offers the possibility to add a new product to the shop’s system, update its information, and reduce its stocks.

3- Reviews:

The review service manages, updates and retrieves the products’ reviews. It provides the possibility to submit, update, and delete a user’s review of a product given that the user provides valid credentials when attempting to perform these operations. Additionally, this service provides features to let admins delete a user’s review, costumers and admins to flag a review, and any user to retrieve submitted reviews.

4- Sales:

The sales service is mainly responsible for managing users’ purchases and retrieving products’ information.

**III- Implementation Details:**

1- Customers:

The customer service provides a total of seven API endpoints that enable a variety of functionalities such as registering or deleting a user, modifying his information, updating his wallet, and retrieving a single or all customers’ information.

Here is a list of the implemented APIs with a description of each:

* /get\_all\_customers: this endpoint is responsible for retrieving all the registered customers in the database
* /get\_customer/<string:username>: this endpoint is responsible for retrieving a particular user’s information. It takes as a parameter the username of the customer as it is unique.
* /register\_customer: this endpoint accepts only POST requests and is responsible for registering new customers to the system. It receives the data as a JSON dictionary that includes all the user’s information and transfers it to the appropriate database API where the registration gets completed.
* /update\_customer/<string:username>: this endpoint accepts only PATCH requests and is responsible for updating a customer’s information. It receives the data as a JSON dictionary that includes the modified information and transfers it to the appropriate database API. It also takes the customer’s username as a parameter to identify which user’s information is getting updated. It is crucial here that the name of the dictionary’s keys match the names of the database table’s columns as they are used in the update mechanism.
* /delete\_customer/<string:username>: this endpoint accepts only DELETE requests and is responsible for deleting a customer. It takes as a parameter the customer’s username.
* /charge\_wallet/<string:username>: this endpoint accepts only PATCH requests and is responsible for adding funds to a customer’s wallet. It receives the amount to add in a JSON dictionary and takes the customer’s username as a parameter. It transfers both to the appropriate database API to complete the operation.
* /deduct wallet/<string:username>: this endpoint accepts only PATCH requests and is responsible for deducing funds to a customer’s wallet. It receives the amount to deduce in a JSON dictionary and takes the customer’s username as a parameter. It transfers both to the appropriate database API to complete the operation.

2- Inventory:

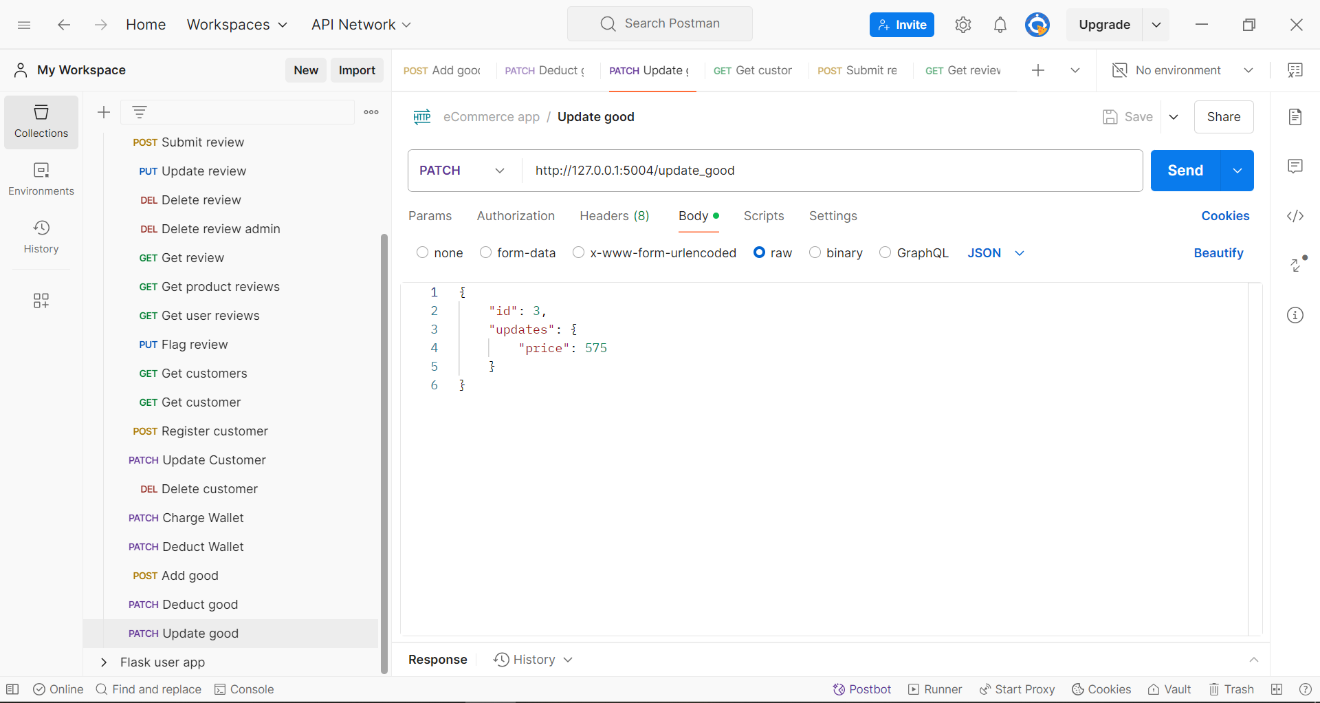
* /add\_good: this endpoint accepts only POST requests and is responsible for adding a product to the shop’s database. It receives all the product’s information in a JSON dictionary that it transfers to the appropriate database API to complete the operation.
* /deduct\_good: this endpoint accepts only PATCH requests and is responsible for reducing the stock amount of a product. It receives the product’s ID and the quantity to deduct in a JSON dictionary and transfers both to the appropriate database API to complete the operation.
* /update good: this endpoint accepts only PATCH requests and is responsible for updating a product’s information. It receives the product’s ID and the updated information in a JSON dictionary consisting of two key-value pairs: *id* and *updates*. The second key has for a value a dictionary that indicates which fields got updated. It is crucial for the keys of that dictionary to have the same name as the database table’s column names as they are directly used in the update mechanism.

3- Reviews:

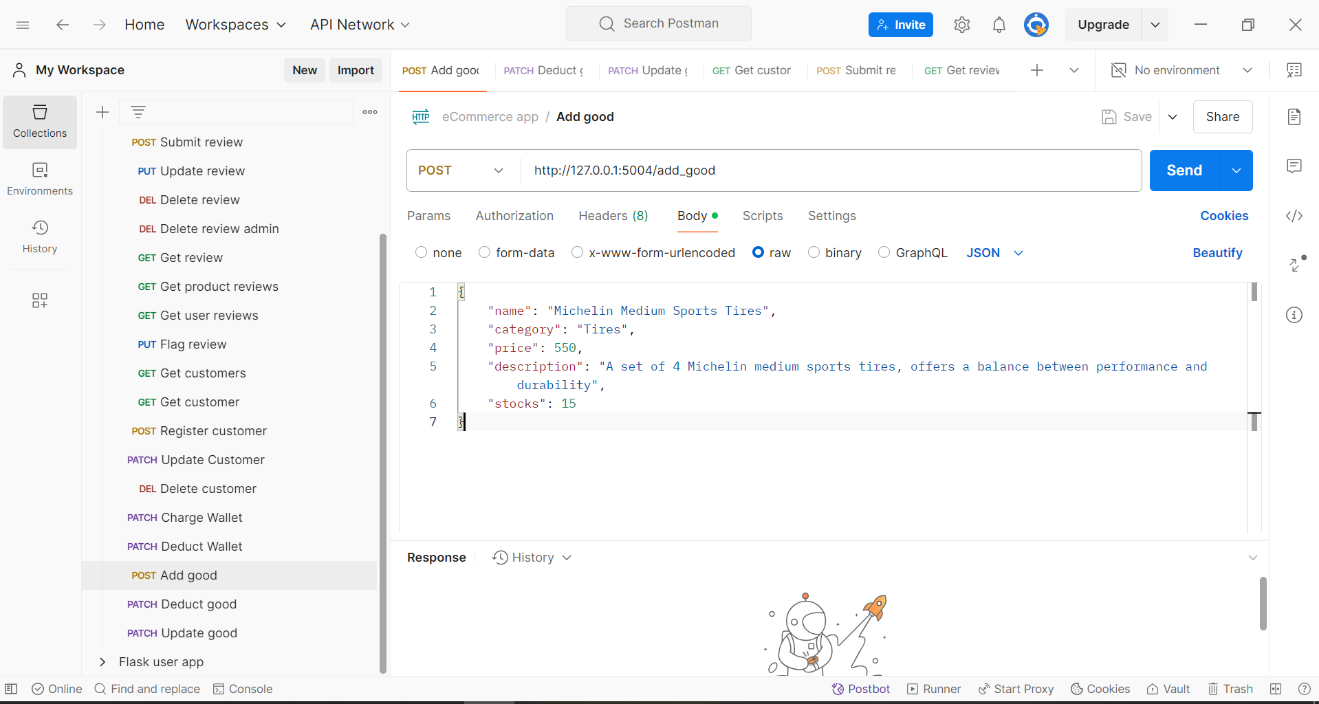
* /submit\_review: this endpoint accepts POST requests and is responsible for adding a customer’s review of a product to the system. The review’s information is contained in a JSON dictionary. As this operation can only be performed by an authenticated user, the service will attempt to authenticate the user by using the credentials included in the received dictionary. If this process is successful, the service proceeds to send the data to the appropriate database API to complete the operation.
* /update\_review: this endpoint accepts PUT requests and is responsible for updating a customer’s review of a product. The review’s information is contained in a JSON dictionary. As this operation can only be performed by an authenticated user, the service will attempt to authenticate the user by using the credentials included in the received dictionary. If this process is successful, the service proceeds to send the data to the appropriate database API to complete the operation.
* /delete\_review: this endpoint accepts DELETE requests and is responsible for deleting a customer’s review of a product. As this operation can only be performed by an authenticated user, the service will attempt to authenticate the user by using the credentials included in the received JSON dictionary. If this process is successful, the service proceeds to send the data to the appropriate database API to complete the operation.
* /admin\_delete\_review: this endpoint accepts DELETE requests and is responsible for deleting a customer’s review of a product. As this operation can only be performed by an authenticated user, the service will attempt to authenticate the user by using the credentials included in the received JSON dictionary. If this process is successful, the service proceeds to send the data to the appropriate database API to complete the operation.
* /get\_review/<string:user>/<string:good>: this endpoint is responsible for retrieving a user’s review of a particular product. It takes as parameters the customer’s username and the good’s name as both are unique.
* /get\_product\_review/<string:good>: this endpoint is responsible for retrieving a product’s reviews. It takes as a parameter the good’s name as it is unique.
* /get\_user\_review/<string:user>: this endpoint is responsible for retrieving a user’s reviews. It takes as parameters the customer’s username as it is unique.
* /flag\_review: this endpoint allows customers and admins to flag a customer’s review if considered inappropriate. It receives data such as the flag’s value, the reviewed good and the customer that made the review in a JSON dictionary which is transferred to the appropriate database API to complete the operation.

4- Sales:

* /get\_available\_goods: this endpoint is responsible for retrieving all the goods that have stock levels greater than 0.
* /get\_good\_details/<int:id>: this endpoint is responsible for retrieving a particular good’s details. It takes as a parameter the product’s ID which it passes to the appropriate database API to complete the operation
* /record\_sales: this endpoint accepts POST requests and is responsible for managing purchases. It receives all the information related to the transaction such as the costumer’s username, the product’s name and the desired quantity in a JSON dictionary. It proceeds to check that the user has the required amount of money in his wallet and that the desired quantity is available. If all is successful, the transaction gets performed and recorded in a *PURCHASES* table by transferring the data to the appropriate database API to complete the operation.

****

*Screenshot of the /update\_good endpoint request in Postman*

****

*Screenshot of the /add\_good endpoint request in Postman*

**A screenshot of a computer

Description automatically generated**

*Screenshot of the /delete\_review endpoint request in Postman*

**IV- Database Design:**

We opted to use the SQLite database system for our project as it doesn’t require the advanced and powerful features of advanced database management systems such as MySQL and PostgreSQL.

Our database consists of five tables interconnected through primary and foreign keys. Here is the entity relationship diagram of our database that explains the relationship between the different tables:

A diagram of a company

Description automatically generated

**V- Error Handling and Validation:**

The error handling mechanisms employed in our project consist mainly of try-catch statements that will ensure the system’s operation while protecting the database from problematic changes and returning an appropriate error message in case of any failure.

In terms of validation, we verify in all endpoints that the targeted resource exists before trying to access it to prevent receiving database error messages.

**VI- Testing:**

We tested all our services through two approaches: Postman API requests and pytest.

1- Postman API:

The Postman API allowed us to test with ease all the endpoints with different inputs to check that they are all operational and doesn’t present any errors or failures. Additionally, the Postman API helped us in adjusting the format of the data that needs to be sent to the backend systems which in turn helped us avoid any syntax or format related errors.

2- pytest:

We have employed pytest as an official and final testing approach to verify that all services are working properly and are generating acceptable outputs. The test cases used for each endpoint consists of inputs that would be usually generated by a front-end interface, or some other backend system and we verify that the generated output corresponds to one of the expected outcomes.

**VII- Deployment and Integration:**

We deployed our application using Docker containers. Each service, including the database API, is deployed in a container of its own. This required then the development of five separate Dockerfiles, each indicating to the built image the service to run.

As the database API will be running in a container of its own, the requests made in the services’ codes cannot be directed anymore to the IP address 127.0.0.1, otherwise the container will believe the request is meant to a service running in it. We had to modify these requests and address them to the name databaseAPI, which is name of the container that will be running the database API. This will make the container resolve this name to get the correct IP address and send the request to it.

Before running any of the images in a container, we needed to define a network that will include each launched container to allow communication between the containers, specifically between a service and the database API. The command used to define the network is the following: *docker network create -d bridge my-net*

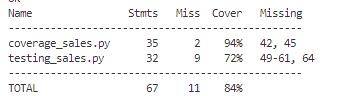
Finally, when we want to run an image, we must indicate to the container that it is part of the created network using the following command (here for the review service for example): *docker run --network my-net -itd --name reviews -p 5002:5002 review\_service*

**VIII- Documentation and Profiling:**

We have compiled a thorough documentation of our main services and database API using the Sphinx Python library. You will find in our GitHub repository the index.html file in the docs folder that will allow you to investigate the project’s documentation.

We have also tested our services’ performance through memory and coverage profiling. The results we obtained were very positive. Indeed, all the endpoints had very low memory usage, mainly attributed to the fact that no heavy computations need to be performed: we are simply retrieving or writing data from/to the database (non-volatile storage) which doesn’t require keeping the data stored in memory.

Additionally, the coverage profiling generated good results with most service scripts having a coverage greater than 80% which indicates that there were few redundant or useless instructions.



*Screenshot of the coverage profiling for the sales service*

A screenshot of a computer

Description automatically generated

*Screenshot of the memory profiling for the /get\_all\_customers endpoint*

A white text box with black text

Description automatically generated

*Screenshot of the memory profiling for the /charge\_wallet endpoint*

**IX- GitHub and Version Control:**

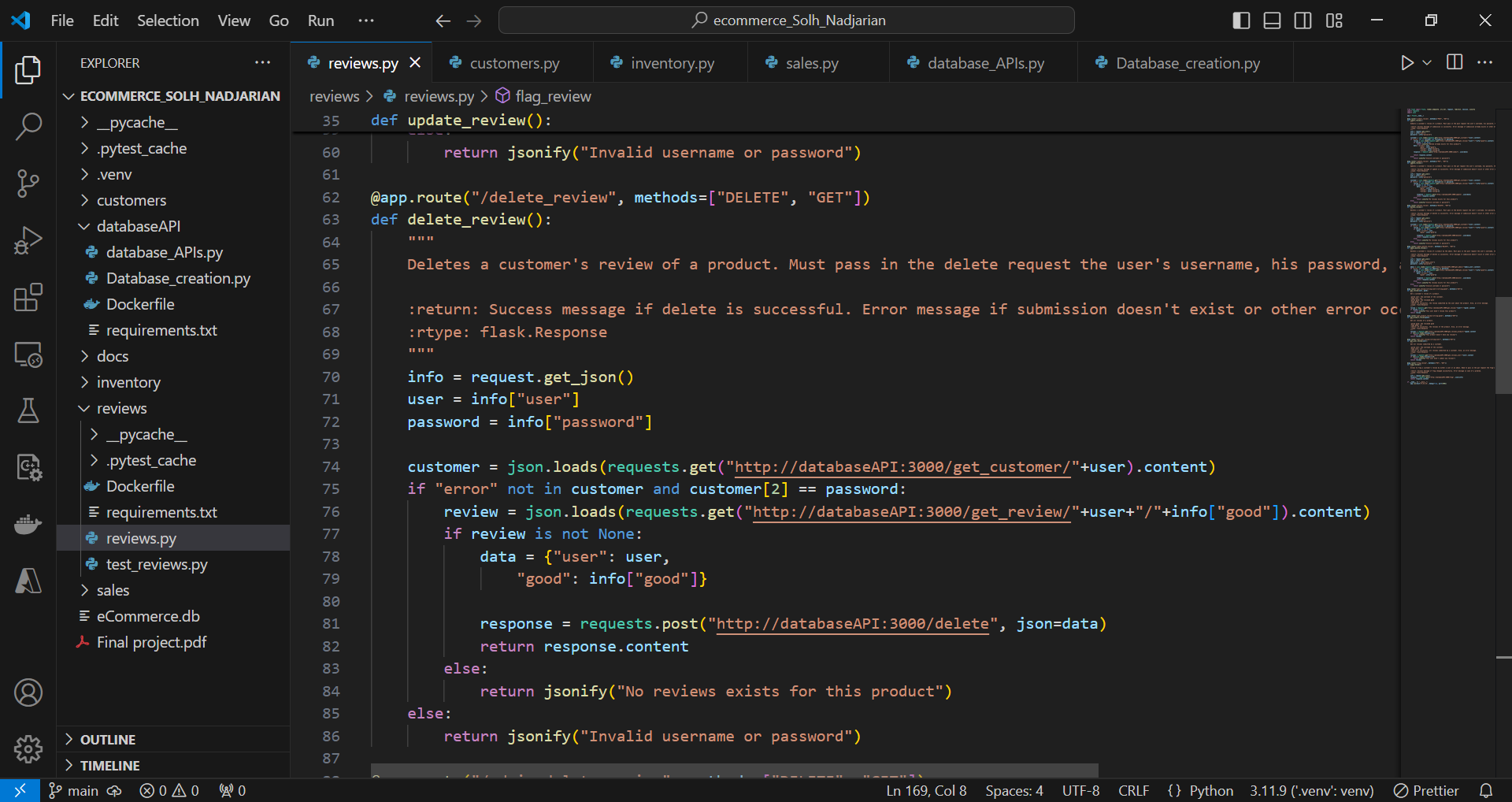
Throughout the project, we heavily relied on GitHub to make sure that we were working on the same version of the code and to share any updates or new files with ease. You will find here the link to the GitHub repository we used and that includes all the files and scripts needed:

<https://github.com/Das301/ecommerce_Solh_Nadjarian>

**X- User Authentication:**

Due to the lack of a front-end interface, which in turn makes it difficult to use a session-oriented system where we would make use of a JWT token approach, we resorted to the simple authentication method where the user submits his credentials which are then passed in a database query. If the query’s output is not null, the user is authenticated and can proceed with the operation.

This mechanism is used in the submit, update and delete review endpoints. On top of passing the review’s information, the user must also submit his credentials at each operation to authenticate him first, then perform the operation.



*Screenshot of the user authentication mechanism in the /delete\_review endpoint*

**XI- Moderation:**

If a review is deemed inappropriate, customers and admins can flag this review by setting its flag attribute in the database to True. Admin users can then decide to delete this review, but they will need to provide their credentials in order to perform this operation.