Chapter 4

Users microservice

4.1 Description

The microservice contains endpoints for user login, obtaining and refreshing JSON Web Tokens (JWT) for authenticated users, and revoking tokens on user logout. The endpoints for JWT generation and refresh utilize Django's built-in TokenObtainPairView and CustomTokenRefreshView classes, respectively. Additionally, there is an OAuthTokenObtainPairView for obtaining JWTs using Google authentication.

The microservice also includes several endpoints for user management, including registering new users, updating user accounts, retrieving user information, and adding reviews for farmers. These endpoints are grouped together under a single UsersView class and utilize various HTTP methods, such as GET, POST, and PATCH. There are also endpoints specifically for managing farmer and rider profiles, including updating the number of insertions for farmers and changing the status of riders.

Overall, this microservice appears to be a key component of a larger web application that allows for user authentication and authorization, as well as user-related operations such as user registration, profile management, and review posting.

4.2 JWT Authentication

JWT stands for *JSON Web Token*, which is a standard for securely transmitting information between parties as a JSON object. In the context of authentication, JWT is used to authenticate and authorize users, and it allows clients to obtain a token that can be used to access protected resources.

In our implementation, we use the third-party Django library *rest_framework_simplejwt*, as it provides a simple and powerful implementation of JWT authentication.
Here is how it works:

1. A user logs in using their username and password. The Django view authenticates the user and generates a JWT token.
2. The JWT token is then returned to the client as a response to the login request.
3. The client can then use this JWT token to access protected resources by including it in the Authorization header of subsequent requests (this is done automatically by the front-end).
4. When the server receives a request with a JWT token, it verifies the token's signature and decodes the token to extract the user's information.
5. If the token is valid and has not expired, the server allows the request to proceed and grants the user access to the protected resource.
6. If the token is invalid or has expired, the server denies the request and returns an error response.

In particular, there are generated two types of tokens the *access* token and the *refresh* token.

The *access* token is a short-lived token that is used to authenticate a user for a limited time period, usually for a few minutes (5 minutes in our case). This token contains the user's identity and permissions encoded within it. When a user logs in or authenticates for the first time, an access token is issued to them, and this token is sent along with every subsequent request to the server to verify their identity and permissions.

The *refresh* token, on the other hand, is a long-lived token that is used to obtain a new access token when the current one expires. When the *access* token expires, the user can send the refresh token to the server to obtain a new access token without having to log in again.

By using two types of tokens, the JWT authentication system can achieve better security and usability. Short-lived access tokens reduce the risk of token hijacking or misuse, while long-lived refresh tokens provide a seamless and user-friendly authentication experience. Additionally, the use of refresh tokens also helps reduce the load on the server, as the user can obtain a new access token without having to make a new login request.

In addition, the library provides features such as token refreshing, blacklisting, and custom claims that can be used to customize the authentication process.

4.2.1 Google OAuth

The system also allows to authenticated using a Google account. An OAuth is an open authorization service that allows websites or applications to share user information with other websites without being given a user's password. Users can sign in to multiple sites using the same account without creating other credentials. Therefore, this feature allows to register to our website without an explicit users' password.

In this case, the user has his own *access* token different from the JWT token, which is sent to the user once he has been authenticated to Google. The front-end sends that token to the back-end, which uses it to fetch user's information from the Google account. Now, it is verified whether there exists an account with that email in our system: if there is then is simply generated a JWT token using the *rest_framework_simplejwt* library; otherwise, it is first created an account to our website using a fake password and then it is genereted the JWT token.

The advantage of this method it that, in the end, the user will always use the JWT token, as if he accessed through credentials, so we do not need to handle particular cases.

4.3 API

4.3.1 Resource: Token

Method	Endpoint	Request body	Description
POST	token/	JSON: { “email”: “xxx” “password”: “xxx” }	Generates the *access* and the *refresh* tokens associated to user with those email and password.
POST	token/refresh/	JSON: { “refresh”: “xxx” }	It takes as input a *refresh* token and outputs the two new *access* and *refresh* tokens. It also returns a new *refresh* token since the first one is backlisted.
POST	token/verify/	JSON: { “user_id”: “xxx” }	In this case the token is passed in the Authorization header. This API checks whether the user id passed as input is the one associated with the token. If it is not the case, the access is denied.
POST	oauth/token/	JSON: { “access_token”: “xxx” }	The *access* token in input is the token generated by the Google OAuth service. The API then returns the usual *access* and *refresh* JWT tokens of our system.
POST	logout/blacklist/	JSON: { “refresh_token”: “xxx” }	The *refresh* token is blacklisted. This will make sure that the refresh token cannot be used again to generate a new token.

4.3.2 Resource: User

Method	Endpoint	Request body	Description

POST	login/	JSON: {
		“email”: “xxx”
		“password”: “xxx”
		}
			It checks if there exist a user with that email and password in the DB. If it is not the case an exception is thrown.

GET	users/		Return all users that are stored in the database.

POST	users/	JSON: {
		”name”:”xxx”,
		”email”:”xxx”
		”password”:”xxx”
		”account_type”:”xxx”
		}
			This creates a new instance of the user only when you satisfied all the requirements needed in order to register to this application. If you don’t respect some of this requirements: like unique email and so on, u can’t able to register to the site.

GET	users/<int:user_id>/	JSON : {
		”user_id” : ”xxx”
		}
			Get the specific user which match the user_id passed through the request. If the user has some special extra information, that comes from special kind of account, the request retrieve either the user information and the user extra info.

GET	users/<int:user_id>/name/		Get the name of the user with that id.

PATCH	users/<int:user_id>/	JSON: {
		”account_type”:”xxx”,
		}
			This request modify the field of the user in the db by updating the type of the users account. This is a special operation that is handled if cause some error by administrator.

PATCH	users/<int:user_id>/changes/	JSON: {
		”name”:”xxx”,
		”email”:”xxx”
		”account_type”:”xxx”
		”billing_address”:”xxx”
		”shipping_address”:”xxx”
		”phone_number”:”xxx”
		”bio”:”xxx”
		”billing_address”:”xxx”
		}
			Updates the user’s account information, such as name, billing, shipping address and so other usefully information. It doesn’t require that all field are filled up with some context.

POST	users/<int:user_id>/<int:type>/	Farmer
		JSON: {
		”bio”:”xxx”,
		”farm_location”:”xxx”
		}
		Rider
		JSON: {
		”available”:”xxx”,
		”bio”:”xxx”
		}
			Updates user extra information based on type, by adding some new field in the database. In this case gives two additional information for the specific case of rider and farmer.

4.3.3 Resource: User.Farmer

Method	Endpoint	Request body	Description
GET	farmers/<int:user_id>/		Get the instance of farmer which has the sames user_id, then retrieve all the extra information about it.
PATCH	farmers/<int:user_id>/ id	JSON: { "farmer_id":"xxx" }	Once a farmer publish his insertion, this request is called in order to increase the global counter of published insertion for the specific farmer

4.3.4 Resource: User.Rider

Method	Endpoint	Request body	Description
GET	riders/		Search through the database to find the first raider available in order to assign him to a new delivery.
GET	riders/<int:user_id>/		Get the instance of rider which has the same user_id, then retrieve all the extra information about it.
PATCH	riders/<int:user_id>/	JSON: { "available":"xxx" }	Special operation for the rider. It change only the status of the rider by putting him available or not available,which means being able to handle a delivery and not being able to handle a delivery, respectively

4.3.5 Resource: Review

Method	Endpoint	Request body	Description
GET	review/<int:user_id>/		Return the last review written for the specific farmer and display it when u visit the Farmer profile page.
POST	review/<int:user_id>/	JSON: { "rating":"xxx", "comment":"xxx", "farmer_user":"xxx", }	User can create a review about the order. This review is applied both to the shipping and to the quality of the product selled by the farmer. Once the user create a review it can be available on the specific farmer's page.

Chapter 5

Insertion microservice

5.1 Description

This microservice implements all the functionalities related to insertions. Creation, deletion and update of insertions and boxes are all treated here. The booking of products is also treated by this microservice.

5.2 API

5.2.1 Resource:Insertion

Method	Endpoint	Request body	Description
GET	insertions/	JSON: {"search":"xxx", "expiring":"xxx", "farmer":"xxx"}	List all or a subset of the insertions.
POST	insertions/	JSON: {"title":"xxx", "description":"xxx", "expiration_date":"xxx", "gathering_location":"xxx", "image":"xxx", "reported":"xxx", "farmer":"xxx", "related_name":"xxx", "private":"xxx", "request":"xxx"}	Creates a new insertion.
GET	insertions/<int:id>/		Retrieve a specific insertion. If the insertion has expired, it is deleted from the database.
GET	insertions/<int:id>/image/		Retrieve the image of the specified insertion.
PUT	insertions/<int:id>/	JSON: {"title":"xxx", "description":"xxx", "gathering_location":"xxx", "image":"xxx", "farmer":"xxx", "related_name":"xxx", "private":"xxx", "request":"xxx"}	Create new insertion
DELETE	insertions/<int:id>/		Delete the specified insertion.

5.2.2 Resoure: Box

Method	Endpoint	Request body	Description
GET	insertions/<int:id>/boxes/		Retrieve all the boxes related to an insertion.
POST	insertions/<int:id>/boxes/	JSON: {"insertion":"xxx", "weight":"xxx", "validators":"xxx", "size":"xxx", "price":"xxx", "number_of_available_boxes":"xxx", }	Create a new box for an insertion.
PATCH	boxes/<int:box_id>/decrease		Decrease the number of boxes of a given type.

5.2.3 Resoure: Booking

Method	Endpoint	Request body	Description
GET	booking/<int:request_id>/		Returns the specified request
PUT	booking/<int:request_id>/	JSON: {"insertion":"xxx"}	As a farmer, accept a request by publishing a private insertion.
GET	booking/requests/<int:user_id>/		Returns the list of a user's requests (booked products).
GET	booking/inbox/<int:farmer_id>/		Returns the list of a farmer's requests received by users.
POST	booking/	JSON: {"user":"xxx", "farmer":"xxx", "title":"xxx", "comment":"xxx", "weight":"xxx", "deadline":"xxx", "insertion":"xxx"}	Creates a new request.
DELETE	booking/<int:id>/		Deletes the specified request.

Chapter 6

Subscriptions microservice

6.1 Description

This service implements the subscription functionality: a customer can subscribe to a farmer in order to be notified when that farmer publishes new insertions.

6.2 API

6.2.1 Resource: Queue

Method	Endpoint	Request body	Description
PUT	customer/<int:user_id>/	JSON: {"farmer_id":"xxx"}	Creates a new binding between the customer's queue and the farmer's exchange. If the queue or the exchange do not exist, they are created. Each customer has its own queue.
PATCH	customer/<int:user_id>/	JSON: {"farmer_id":"xxx"}	Deletes a binding between a customer's queue and a farmer's exchange.
GET	customer/<int:user_id>/		Reads all the messages in the queue.
DELETE	customer/<int:user_id>/		Deletes the queue.

6.2.2 Resource: Exchange

Method	Endpoint	Request body	Description
POST	farmer/<int:farmer_id>/	JSON: {"message":"XXX"}	Passes a message to the exchange relative to the farmer. If the exchange does not exists, it is created. The exchange will deliver the message to all the queues it is bound to.
DELETE	farmer/<int:farmer_id>/		Deletes the exchange.

Chapter 7

Payments & Orders microservice

7.1 Description

The microservice is responsible for handling requests related to order and payment management in this application. The payment are handled by the API given by Stripe, a payment processing company.

It provides a set of RESTful endpoints for processing payments, saving stripe information, getting specific orders, updating orders, updating the status of riders, and retrieving orders by email. The NewView class is used as the view for handling incoming requests, and it has various methods associated with different HTTP verbs and actions, including test_payment for processing payments, in particulary save_stripe_info for saving stripe information that is handled by STRIPE API, and getSpecificOrder for retrieving specific orders.

As a microservice, it provides an independent and modular solution for order and payment management, which can be easily scaled and updated without affecting other components of the application. It encapsulates the order and payment logic, providing a clear separation of concerns between different parts of the system. This allows for better maintainability, security, and flexibility in managing orders and payments within the application. Thanks to stripe it also provides robust security features, including data encryption and tokenization, to help protect sensitive payment information.

7.2 API

7.2.1 Resource: Orders

Method	Endpoint	Request body	Description
GET	getSpecificOrder/<str:payment_method_id>/		Reeturn the information about the specific order with the same unique payment_method_id.
GET	getSpecificOrderByRider/<int:rider_id>/		Get the instance of the order that is handled by the rider which has the same value of the parameter rider_id.
PATCH	update-order/	JSON: { "payment_method_id": "riderId":"xxx" , }	Update the status of the order when is in the process of delivering. The shipping can be handled by a rider or the costumer can choose to pickup at the warehouse.
PATCH	update-status-rider/	JSON: { "riderUserId":"xxx" }	Special operation of the rider and change the status of a specific order,which has the same rideerUserId, once the delivery is completed. It can be generated by the rider itself directly from it's account.
GET	get-orders/<str:email>/		Retrieve all the orders by passing the email address of the users as a parameter.

7.2.2 Resource: Payments

Method	Endpoint	Request body	Description

POST	save-stripe-info/	JSON: {
		"email":"xxx",
		"payment_method_id":
		"price":"xxx",
		"box_names":"xxx",
		"farmer":"xxx"
		}
			Creates an Intent of the payment with the total price of the items in the cart and the user, then the intent is handled in order to create the payment. Futhermore it create and save the information about the order early created.

Chapter 8

Shopping Cart microservice

8.1 Description

The Shopping Cart microservice is responsible for handling the requests that regards the shopping cart management, is linked to the checkout process and take care of checking the information passed from the Insertions microservice.
Inside the microservice two main models have been defined to manage the requests and to define the single, definitive source of information about the data: the Cart and the CartItem. Each of these model maps to a single database table and the attributes defined inside the model outline the column in the table.
The microservice provides a fixed amount of RESTful endpoints for the following functionalities:

- Create the shopping cart
- Retrieve the cart
- Delete the cart
- List the cart items
- Add the insertion boxes to the shopping cart
- Remove the insertion boxes
- Checkout the shopping cart

The main concept of the Cart model is that each shopping cart is linked to a user and both to a farmer, so it's possible to only add products to the shopping cart that have been created by a single farmer. For each product that doesn't belong to the same farmer, upon request made to the user, the shopping cart is deleted and a new one is created with the products of the new farmer.

8.2 API

Here it's possible to see a review of all the APIs built for the Shopping Cart microservice. These APIs cover both the Cart and the CartItem logic.

8.2.1 Resource: Cart

Method	Endpoint	Request body	Description
GET	users/<int:user_id>/cart/	The user id is extracted from the endpoint	Obtains the shopping cart given the user ID

POST	users/<int:user_id>/cart/	JSON:{
		 "user": user_id
		 "current_farmer": farmer_id
		 }
			Create a new shopping cart that is linked to a user by its user id and a farmer by its farmer id

DELETE	users/<int:user_id>/cart/	The user id is extracted from the endpoint	Delete the shopping cart linked to the user by its user id

8.2.2 Resource: CartItem

GET	users/<int:user_id>/cart/items/	The user id is extracted from the endpoint	Obtains the boxes saved in the shopping cart given the user ID

PUT	users/<int:user_id>/cart/items/	JSON:{
		"cart": cart_id
		"box_id": box_id
		"name": boxName
		"size": boxSize
		"weight": boxWeight
		"price": boxPrice
		}
			Create a new CartItem based on the info of the insertion box and it's added to the shopping cart

DELETE	users/<int:user_id>/cart/items/	JSON:{
		"box_id": box_id
		}
			Delete the CartItem linked to the box id passed in the request