POLITECNICO
MILANO 1863

Iqueue Project
Implementation Document
Software Engineering for Automation (2022-2023)

Giacomelli Gianluca, 10615105
Gottardini Andrea, 10617000
Veronese Niccolò Enrico, 10620278

Professors: Rossi Matteo Giovanni
Lestingi Livia

Summary

This document aims to provide a detailed guideline for implementing the Iqueue app. It will offer a clear and concise overview of the Django framework's architectural components that make up the app. Then, a detailed and schematic overview of the code is proposed. Next, an outline of the implemented requirements and their corresponding functions is presented, along with a user manual for the app. The objective is to present the information in a more comprehensible and articulate manner.

1. Frameworks

In our IQueue App project, we have utilized the popular Django framework to develop an application that addresses specific requirements. Django is an open-source web framework based on Python that simplifies the development of complex and scalable web applications. Django promotes a modular approach to designing and building web applications by allowing developers to split the main features of the application among different files that interact together.

The main files in Django are models.py, views.py, templates, urls.py and forms.py, settings.py and admin.py. These components allows the developer to design the backend of the application meeting the desired requirements.

Manage.py

Manage.py runs the development server, create new applications, perform database migrations, interact with the database, create administrative accounts, run tests, and more. It serves as a convenient interface for executing common tasks and automating processes within a Django project.

Settings.py

The settings file serves as the central configuration hub for our Django project. Here, we have specified various settings that define the behavior of our application. These settings include database connections, middleware configurations, static file paths, authentication settings, and much more. By customizing these settings according to our specific project requirements, we can tailor the behavior of our Django application to suit our needs.

Views.py

In our project, the views are defined in the views.py file. Views play a crucial role in handling the logic behind serving HTTP requests and generating appropriate responses. They are responsible for processing user input, querying the database, and interacting with other components of our application. By defining view functions or classes, we have determined how our application responds to different URLs and user actions. Within the views, we have implemented logic to process incoming requests and generate the necessary data or render appropriate templates.

Templates

Templates, residing in the templates directory, represent the presentation layer of our application. They are HTML files that incorporate the Django template language (DTL) tags for dynamic content rendering. By separating the presentation logic from the business logic, we have been able to create visually appealing and interactive user interfaces. In our views, we have utilized templates to render the final HTML response, passing dynamic data as context variables. This way, we have achieved a clean separation between the backend processing and the frontend representation of our application.

Models.py

Models, defined in the models.py file, form the backbone of our application's data structure. They define the schema for the database tables and represent the entities and relationships within our system. Utilizing Django's Object-Relational Mapping (ORM) capabilities, we have defined Python classes that seamlessly translate into database tables and simplify database interactions. The models provide a high-level API for managing data, allowing us to perform common database operations using Python code instead of writing complex queries.

Urls.py

URL configuration is handled in the urls.py file. This file acts as a central routing mechanism that maps incoming URLs to their corresponding views. By defining URL patterns and associating them with specific views, we have determined the flow and navigation within our application. When a user accesses a specific URL, Django's URL resolver matches it to the appropriate view function or class defined in urls.py, enabling us to direct the request to the intended functionality.

In our Django project, the flow typically starts when a user makes a request by accessing a specific URL in their browser. The URL is then matched against the defined URL patterns in urls.py. Once a matching URL is found, the associated view function is invoked. This view processes the request, leveraging the models to retrieve or modify data as required. The retrieved data is then passed as context variables to the appropriate template, which handles the rendering of the final HTML response. Finally, the response is sent back to the user's browser, completing the request-response cycle.

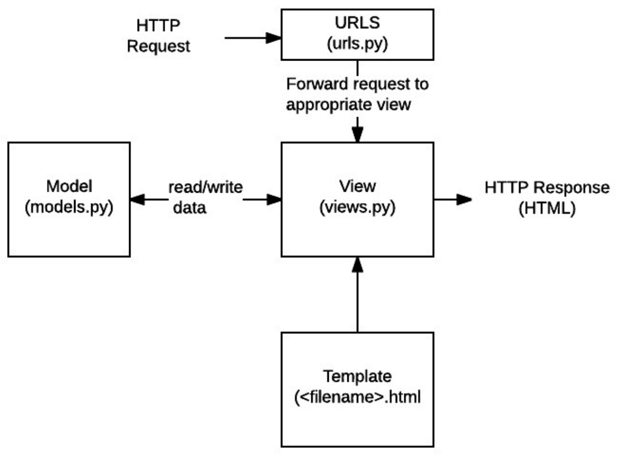


Figure 1: Django interaction scheme

Forms.py

Forms provide a structured way to gather user input, such as text, numbers, selections, and files, which can be processed and utilized by the server-side application.

Admin.py

The main purpose of "admin.py" is to allow developers to easily define how the models in their application should be presented and manipulated in the Django administration interface. By customizing the "admin.py" file, you can control what models are displayed, how they are displayed, and which fields are editable. It enables you to tailor the administration interface to suit the specific needs and requirements of your application.

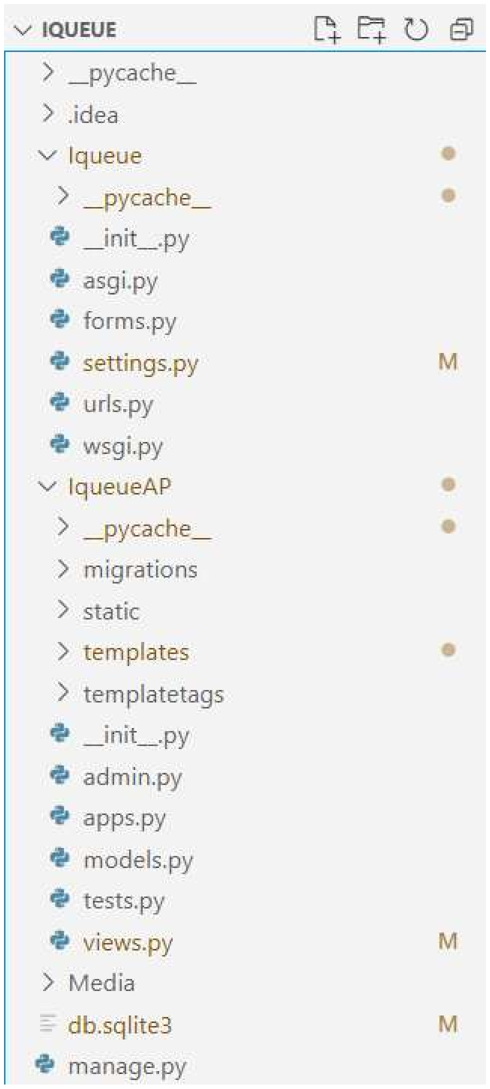
Manage.py

Manage.py runs the development server, create new applications, perform database migrations, interact with the database, create administrative accounts, run tests, and more. It serves as a convenient interface for executing common tasks and automating processes within a Django project.

2. Code structure

In this chapter a detailed overview of the code is provided, describing how the different Django components have been developed in our case.

First, a screenshot of the folder organization of the Iqueue application is reported:



```
IQUEUE
  > __pycache__
  > .idea
  v Iqueue                      ●
      > __pycache__             ●
        __init__.py
        asgi.py
        forms.py
        settings.py             M
        urls.py
        wsgi.py
  v IqueueAP                    ●
      > __pycache__             ●
      > migrations
      > static
      > templates               ●
      > templatetags
        __init__.py
        admin.py
        apps.py
        models.py
        tests.py
        views.py                M
  > Media
    db.sqlite3                  M
    manage.py
```

Figure 2: folder organization

In the folder Media, the shop images are saved.

Now we will provide a deep and exhaustive analysis of the relevant code components of the Iqueue application.

2.1 Settings.py

The settings.py file in Django contains various configurations and settings for the Django project. Now we will go through a detailed description of each section in our settings.py file:

1. Base Directory and Project Information:
   - "**BASE_DIR**": this variable specifies the base directory of your Django project.
   - Project Information: The initial comments describe the project's name and how the file was generated using '**django-admin startproject**'. It also provides links to the official Django documentation for further information on settings and topics.

2. Development Settings:
   - "**SECRET_KEY**": This setting is used for cryptographic signing and should be kept secret. It is used to secure cookies, sessions, and other sensitive data.
   - "**DEBUG**": When set to **True**, this enables Django's debug mode, which provides detailed error messages during development. It should be set to **False** in production for security reasons.
   - **ALLOWED_HOSTS**: This list specifies the valid host/domain names that can be used to access the Django application. For development, an empty list allows all hosts.

3. Application Definition:
   - "**INSTALLED_APPS**": This setting lists all the Django apps installed in the project. Apps provide specific functionalities and can be both built-in Django apps and custom apps created for the project.
   - "**MIDDLEWARE**": Middleware is a series of hooks that process requests and responses. It includes security middleware, session middleware, authentication middleware, and more.

4. URL Configuration:
   - **"ROOT_URLCONF":** This setting points to the URL configuration module for the project. It specifies the Python module that contains the URL patterns.

5. Media Configuration:
   - **"MEDIA_ROOT":** This setting defines the absolute filesystem path to the directory where uploaded media files will be stored.
   - **"MEDIA_URL":** This setting specifies the base URL for serving media files.

6. Database Configuration:
   - **"DATABASES":** This setting defines the configuration for the project's database connection. The given configuration uses the SQLite database engine and sets the database name to **db.sqlite3**.

7. Password Validation:
   - **"AUTH_PASSWORD_VALIDATORS":** This setting lists the password validation backends to be used for validating user passwords.

8. Internationalization and Localization:
   - **"STATIC_URL":** This setting specifies the base URL for serving static files (CSS, JavaScript, images, etc.).

   - **"DEFAULT_AUTO_FIELD":** This setting defines the default primary key field type for models.

9. Static Files Directories:
   - **"STATICFILES_DIRS":** This setting lists the directories from which Django collects static files.

2.2 Views.py

The views.py file in Django contains the views or functions that handle HTTP requests and generate HTTP responses. Each view corresponds to a specific URL pattern and performs certain actions based on the request parameters and data.

Here are reported the most relevant functions among those we had implemented.

2.2.1 Select Role

The **selectRole** function handles the display of the "SelectRole" page, which allows the user to choose a specific role. Here's a detailed description of the function:

1. The function receives a **request** object as a parameter, which represents the HTTP request received from the client.
2. The **request.session.get** method is used to retrieve the values of the 'name', 'idc', and 'idso' keys from the session. If any of the keys are not present in the session, an empty value is assigned to the corresponding variable.
3. An empty list **idss** is initialized to hold the advertisement IDs.
4. A **for** loop is executed over all **Advertisement** objects in the database to retrieve the IDs and add them to the **idss** list.
5. During the **for** loop, it checks if the current date is equal to the end date of the advertisement. If so, the advertisement is deleted from the database using the **adv.delete()** method.
6. The **shop_advertised** variable is initialized with a value of **None**.
7. A check is performed to see if the **idss** list contains at least one element.
8. If the **idss** list contains elements, a random ID is selected from the list using **random.choice(idss)**.
9. A query is executed on the **Shop** model to retrieve the first object that has the selected ID.
10. Finally, the "SelectRole.html" page is rendered, passing the values of the **name**, **idc**, **idso**, and **shop** variables as context to be used in the template.

In summary, the **selectRole** function retrieves session values and advertisements from the database, randomly selects an advertisement, and then renders the "SelectRole.html" page, passing the retrieved values as context to be displayed in the template.

2.2.2 Registration View

The **registration_view** function handles the registration process for new users. Here's a breakdown of how the function works:

1. The function checks the HTTP method of the request. If it's a POST request, it means that the user has submitted the registration form.

2. Inside the **if request.method == 'POST'** block, a **RegistrationForm** instance is created using the data from the request's POST parameters.

3. The form's validity is checked using the **form.is_valid()** method. If the form is valid, the registration process proceeds.

4. The function extracts the cleaned data from the form fields, such as **first_name**, **last_name**, **password**, **email**, and **birthday**.

5. It checks if the **birthday** date is in the future. If so, it renders the "ErrorBirthdayAccount.html" template, indicating an error with the birthday date.

6. It checks if an **Account** object already exists with the provided email by querying the database. If a matching account is found, it renders the "ErrorEmailAccount.html" template, indicating an error with the email address.

7. If the birthday date is valid and the email is not already associated with an account, unique identifiers (**idso** and **idc**) are generated using **uuid.uuid4()**.

8. An **Account** object is created using the extracted form data and the generated identifiers.

The **account** object is saved to the database using the **save()** method.

9. Additionally, a **PurchaseList** and a **WishList** associated with the new account are created and saved to the database.

10. Finally, the user is redirected to the 'success' page.

11. If the request method is not POST, indicating that the user is accessing the registration page for the first time, an empty **RegistrationForm** instance is created.

12. The 'formRegistration.html' template is rendered, passing the **form** object as context to be used in the template.

In summary, the **registration_view** function handles the registration form submission. It validates the form data, creates an **Account** object with unique identifiers, saves the account and related models to the database, and redirects the user to a success page. If the form is not submitted, it renders the registration form template with an empty form instance.

2.2.3 Login view

The **login_view** function handles the user login process. Here's an explanation of how the function works:

1. The function checks the HTTP method of the request. If it's a POST request, it means that the user has submitted the login form.

2. Inside the **if request.method == 'POST'** block, a **LogIn** form instance is created using the data from the request's POST parameters.

3. The form's validity is checked using the **form.is_valid()** method. If the form is valid, the login process proceeds.

4. The function extracts the cleaned data from the form fields, such as **email** and **password**.

5. An attempt is made to retrieve an **Account** object from the database that matches the provided email and password.
6. If a matching account is found, the user's name, **idc**, and **idso** are stored in the session for future use.
7. Next, an **idss** list is created to store advertisement IDs.
8. All advertisements are retrieved from the database, and their IDs are added to the **idss** list.
9. If there are any advertisement IDs available, one is randomly chosen.
10. The corresponding **Shop** object is fetched from the database based on the randomly chosen advertisement ID.
11. Finally, the user is rendered the 'SelectRole.html' template, passing the user's name, **idc**, **idso**, and the **shop** object (if available) as context.

If no matching account is found, an error message is added to the form, indicating that the provided login credentials are invalid.

If the request method is not POST, indicating that the user is accessing the login page for the first time, an empty **LogIn** form instance is created.

The 'login.html' template is rendered, passing the **form** object and the **error** message (if any) as context.

In summary, the **login_view** function handles the login form submission. It validates the form data, checks the provided credentials against the database, stores user information in the session, retrieves advertisements and a corresponding shop (if available), and renders the appropriate template.

2.2.4 Booking view

The **Booking_view** function is the view function that handles the booking process for a selected category of shops. Here is a step-by-step description of how it works:

1. The function loads the Django forms **Shop_and_day_selectionForm** and **TimeSlot_selectionForm**.
2. It filters the shops based on the selected category passed as a parameter.
3. The function initializes empty lists, **queues** and **reviews**, to store the queue and rating of each shop, respectively.
4. For each shop in the selected category, it retrieves the associated time slots using **TimeSlot.objects.filter(shop=shop)**.
5. The **checkQueue** method of the **Shop** model is called to calculate the available slots and update the shop's queue count (**shop.queue**).
6. The updated queue count and the shop's rating are appended to the **queues** and **reviews** lists, respectively.
7. Lists of addresses and names are created by extracting the corresponding attributes from the shop objects.

8. If the request method is POST, the function checks for specific conditions:

    - If the first form (**Shop_and_day_form**) is submitted (identified by the presence of 'btnform1' in the request POST data), the selected shop and date are retrieved.

        - The selected shop is obtained using **Shop.objects.get(ids=shop_ids)**.

        - The associated time slots for the selected shop and date are fetched using **TimeSlot.objects.filter(shop=shop, date=date, available=True)**.

        - The booking.html template is rendered with the necessary data, including the shops, time slots, forms, addresses, names, queues, and reviews.

    - If the second form (**TimeSlot_form**) is submitted (identified by the presence of 'btnform2' and 'selected_slot' in the request POST data), the customer's information and the selected time slot are retrieved.

        - The customer ID (**idc**) is obtained from the session data.

        - The selected time slot is fetched using **get_object_or_404(TimeSlot, id=selected_timeslot_id)**.

        - An available slot associated with the time slot and shop is obtained using **Slot.objects.filter(TimeSlot=timeslot, available=True).first()**.

        - The selected slot is marked as unavailable, associated with the customer (**slot.idc** = **idc**), and saved.

        - If all slots associated with the time slot are now unavailable, the time slot is also marked as unavailable and saved.

        - A QR code is generated using the booking details, such as shop ID, date, time, slot number, and customer ID.

        - The QR code image is converted to a base64-encoded string for embedding in the HTML page.

        - A new QR object is created and saved in the database.

        - The qr.html template is rendered, passing the QR code image, shop details, QR object, addresses, and form instances as context.

9. If the request method is not POST, the initial state of the booking page is rendered. The booking.html template is rendered, passing the shops, forms, addresses, names, queues, and reviews as context.

In summary, the **Booking_view** function handles the filtering of shops, calculation of queues and reviews, processing of form submissions for shop and time slot selection, making reservations, generating QR codes, and rendering the appropriate templates for the booking process.

2.2.5 Reservation view

The **Reservation_view** function is the view function that handles the rendering of a customer's reservations page. Here's a breakdown of its functionality:

1. It retrieves the customer's ID (**idc**) from the session.

2. It queries the **QR** model to fetch all the QR objects associated with the customer (**idc**).
3. The **shop_names** and **shop_address** lists are initialized to store the corresponding shop names and addresses for each QR.
4. For each QR, it retrieves the associated shop object using the **ids** field of the QR, and appends the shop's name and address to the respective lists.
5. If the request contains the parameter 'Guide', it indicates that the user clicked on a link to get directions to a shop. It retrieves the shop's **ids** from the request and fetches the corresponding shop object. Then, it constructs a Google Maps URL using the shop's address and redirects the user to that URL.
6. If the request contains the parameter 'Delete_QR', it indicates that the user requested to delete a QR. It retrieves the **idQR** from the request and fetches the corresponding QR object to be deleted. Then, it redirects the user to the 'DeleteQR' view, passing the **idQR** as a parameter.
7. The **list** variable is created by zipping together the **qrs**, **shop_names**, and **shop_address** lists. This creates an iterable of tuples where each tuple contains a QR object, its corresponding shop name, and its corresponding shop address.
8. The **context** dictionary is created to hold the **list** variable.
9. Finally, it renders the 'CustomerReservations.html' template, passing the **context** dictionary to it.

Overall, this view retrieves a customer's reservations, fetches additional information about the associated shops, handles the redirection for getting directions and deleting reservations, and renders the reservations page with the necessary context.

2.2.6 Delete QR

The **DeleteQR** function is the view function that handles the deletion of a QR reservation. Here's a breakdown of its functionality:

1. It takes two parameters: **request** (the HTTP request object) and **idQR** (the ID of the QR to be deleted).
2. It uses the **get_object_or_404** function to retrieve the QR object with the given **idQR**. If the QR object does not exist, it raises a 404 error.
3. If the request contains the parameter 'Choice', it indicates that the user has submitted a confirmation for deleting the QR.
4. It retrieves the value of the 'Choice' parameter to determine whether the user selected 'Yes' or 'No' for deletion.
5. If the user selected 'Yes' for deletion, it retrieves the necessary information from the QR object, such as the start time, end time, date, and associated shop.
6. It uses this information to query the **Slot** model and retrieve the corresponding slot object for the QR. It ensures that the slot is currently unavailable (**available=False**) and associated with the customer (**idc**) from the session.
7. It sets the slot's availability to **True** to mark it as available again and saves the slot.

8. It deletes the QR object.

9. It attempts to find a matching **TimeSlot** object based on the start time, end time, date, and shop, where the timeslot is currently unavailable (**available=False**).

10. If a matching timeslot is found, it sets its availability to **True** to mark it as available again and saves the timeslot.

11. If the **TimeSlot.DoesNotExist** exception is raised (i.e., no matching timeslot is found), it simply continues without raising an error.

12. It redirects the user to the 'Reservation_view' view.

13. If the request does not contain the 'Choice' parameter, it renders the 'DeleteQR.html' template, allowing the user to confirm the deletion.

Overall, this view handles the deletion of a QR reservation, updates the corresponding slot and timeslot availability, and redirects the user back to the reservations page after deletion.

2.2.7 Write review

The **write_review** function is the view function that handles the process of writing a review for a shop. Here's a breakdown of its functionality

1. It retrieves the **idc** (customer ID) from the session.

2. It queries the **Review** model to retrieve all reviews that are marked as not written (**written=False**) and associated with the customer (**idc=idc**).

3. If the request method is 'POST', it indicates that the user has submitted a review.

4. It retrieves the **review_id** from the POST data, which represents the ID of the review being submitted.

5. It retrieves the rating value for the review from the POST data by dynamically generating the form field name using the **review_id**.

6. It retrieves the **Review** object with the given **review_id**.

7. It retrieves the associated shop from the **ids** attribute of the review.

8. It calculates the new rating for the shop by taking the weighted average of the existing rating and the new rating submitted by the customer. The formula used is: **new_rating = ((current_rating * num_reviews) + rating) / (num_reviews + 1)**, and the result is rounded to one decimal place.

9. It increments the **num_reviews** attribute of the shop by 1.

10. It saves the updated shop object.

11. It marks the review as written by setting the **written** attribute to **True**.

12. It saves the updated review object.

The function does not have a return statement, so it does not render a specific template or redirect the user to another page.

2.2.8 Shop view

The **Shop_view** function is the view function that handles the registration of a shop. Here's a breakdown of its functionality:

1. If the request method is 'POST', it means that the form containing the shop data has been submitted.
2. It creates a new instance of the **ShopForm** using the POST data and file data (if present).
3. It checks if the form is valid.
4. If the form is valid, it retrieves the different fields provided by the form using **cleaned_data**.
5. It performs additional checks on the opening and closing times to ensure they are valid (opening time should be before closing time).
6. It constructs the complete address by concatenating the address, number, CAP (postal code), and city fields.
7. It retrieves the shop owner ID (**idso**) from the session.
8. It generates a unique ID for the shop using **uuid.uuid4**().
9. It creates a new instance of the **Shop** model with the provided data and saves it to the database. The initial rating and number of reviews are set to 0.
10. If an image file was uploaded for the shop, it assigns it to the **img** attribute of the shop object.
11. It saves the shop object in the database.
12. It creates time slots for the shop. The time slots are generated for a one-year period starting from the current date. Time slots are only created for weekdays (Monday to Friday) and not for weekends.
13. It iterates over the dates within the one-year period and, for each weekday, iterates over the time slots starting from the opening time until the closing time. It creates a **TimeSlot** object with the corresponding start time, end time, date, and availability status (initially set to True).
14. For each time slot, it creates individual slots (numbered from 1 to **max_numb_clients**) and assigns them to the time slot.
15. It updates the current date and time for the next iteration.
16. After successfully registering the shop, it redirects the user to a success page (**SuccessShopRegistration**).

If the request method is not 'POST', it means that the user is accessing the registration form for the first time. In this case, it creates a new instance of the **ShopForm** and renders the **ShopRegistration.html** template, passing the form as context.

2.2.9 Shop Queue List

The **ShopQueueList** function is the view function that displays the queue list for a particular shop. Here's a breakdown of its functionality:

1. It retrieves the shop object with the specified **ids** using the **get_object_or_404** function.
2. It retrieves all the time slots associated with the shop.
3. It calls the **checkQueue** method on the shop object to get the current customers in the queue and the actual time slot.
4. It stores the customers' IDs (**idc**) in the **customers** variable.
5. If there are customers in the queue, it performs the following steps: a. It creates an empty list **qrs** to store the QR objects associated with the customers in the queue. b. It creates a set **unique_customers** to store the unique customer objects based on their IDs. c. It iterates over each unique customer and retrieves the QR objects associated with them for the current time slot. d. It extends the **qrs** list with the retrieved QR objects. e. It sorts the **qrs** list based on the 'number' attribute of the QR objects. f. It creates an empty list **names** to store the names of the customers. g. It retrieves the names of the customers using their **idc** and adds them to the **names** list.
6. It creates an empty list **numberNOAPP** to store the numbers of customers in the non-app queue.
7. If the shop has a non-app queue (**queue_no_app** is greater than or equal to 0), it iterates over the range of **queue_no_app** and adds the corresponding numbers to the **numberNOAPP** list.
8. It creates a zip object **list** by combining the **qrs** and **names** lists.
9. It prepares the context data with the **list**, **NumberNOAPP**, and **shop** variables.
10. It renders the **ShopQueueList.html** template with the provided context.

2.2.10 Delete Shop

The **DeleteShop** function the view function that handles the deletion of a shop. Here's an overview of its functionality:

1. It retrieves the shop object with the specified **ids** using the **get_object_or_404** function.
2. It checks if a "Choice" parameter is present in the GET request.
3. If the "Choice" parameter exists and its value is "Yes", it performs the following steps: a. It retrieves all the QR objects associated with the shop and deletes them. b. It retrieves all the Advertisement objects associated with the shop and deletes them. c. It retrieves all the Product objects associated with the shop. d. It iterates over each product and performs the following steps:
    - It retrieves the WishListItem objects associated with the product and deletes them.
    - It deletes the product itself. e. It retrieves all the Review objects associated with the shop and deletes them. f. Finally, it deletes the shop itself.
4. It redirects the user to the "MyShops_view" view.
5. If the "Choice" parameter is not present or its value is not "Yes", it renders the "DeleteShop.html" template.

2.2.11 Product View

The **Product_view** function is the view function that handles the registration of a product. Here's an overview of its functionality:

1. It creates an instance of the **ProductForm** form.
2. It initializes the **selected_shop** variable to 0.
3. It retrieves the shop owner's ID (**idso**) from the session.
4. If the request method is POST:
    - It retrieves all the shops associated with the shop owner (**idso**).
    - It retrieves the selected shop's IDs (**ids_selected_shop**) from the POST data.
    - It retrieves the corresponding shop object using the **ids_selected_shop**.
    - It stores the selected shop's IDs (**ids**) in the session.
    - It retrieves the product name, quantity, price, and shop discount from the POST data.
    - It generates a unique ID for the product (**idp**) using **uuid.uuid4**().
    - It generates QR code data based on the product details.
    - It creates a QR code image using the **qrcode** library.
    - It converts the QR code image to a base64-encoded string.
    - It creates a **Product** object with the provided product details and the generated QR code.
    - It saves the **Product** object in the database.
    - It redirects the user to the "SuccessProductRegistration" view.
5. If the request method is not POST, it prepares the context data for rendering the template:
    - It retrieves the shops associated with the shop owner (**idso**).
    - It includes the **Product_form**, **selected_shop**, and **shops** in the context dictionary.
6. It renders the "ProductRegistration.html" template with the provided context.

2.2.12 QR Print

QR_print is the function which retrieves the products associated to a specific shop, in order to obtain thier corresponding QR codes. It takes a request object as a parameter and returns a rendered HTML template named "QR_print.html", which shows all the QR codes stored in the qr attribute of product variables.

Let's break down the code step by step:

1. It starts by extracting the value of the 'idso' key from the session object stored in the request using **request.session.get('idso', '')**. If the 'idso' key is not present in the session, it assigns an empty string as the default value.

2. The code then queries the **Product** model using the **idso** value obtained from the session. It filters the **Product** objects based on the **idso** field.
3. Next, an empty list named **shop_associated_name** is initialized. This list will store the names of the shops associated with each product.
4. A loop is initiated over the **products** queryset obtained in the previous step. Inside the loop, the code queries the **Shop** model using the **ids** field of each product. The **first()** method is used to retrieve the first **Shop** object matching the query.
5. If a **Shop** object is found, its name is appended to the **shop_associated_name** list. Otherwise, if no shop is found, the string 'The shop has been deleted!' is appended to the list.
6. The code then uses the **zip()** function to combine the **products** queryset and the **shop_associated_name** list. The **zip()** function pairs corresponding elements from both iterables, creating an iterator of tuples.
7. Finally, the view function returns the rendered HTML template named "QR_print.html". It passes a dictionary containing the 'list' variable, which holds the zipped data, as the context for rendering the template.

In summary, this view function retrieves products associated with a specific 'idso' from the session, fetches the names of the associated shops, combines the products and shop names using **zip()**, and renders the data in the "QR_print.html" template.

2.2.13 Advertisement view

Advertisement_view is the fucntion which handles the rendering and processing of an advertisement form for a specific shop associated with the current session.

Let's go through the code step by step:

1. It begins by retrieving the value of the 'idso' key from the session object stored in the request using **request.session.get('idso', '')**. If the 'idso' key is not present in the session, it assigns an empty string as the default value.
2. The code queries the **Shop** model using the **idso** value obtained from the session. It filters the **Shop** objects based on the **idso** field.
3. Next, an empty list named **shopadvlist** is initialized. This list will store the shops that have already been advertised.
4. A loop is initiated over the **shops** queryset obtained in the previous step. Inside the loop, the code queries the **Advertisement** model using the **ids** field of each shop. If any advertisement exists for the shop, it appends the shop object itself to the **shopadvlist**.
5. After the loop, a list named **shops_not_adv_yet** is created using a list comprehension. It includes only those shops that are not present in the **shopadvlist**, i.e., shops that have not been advertised yet.

6. If the request method is 'POST', it means that the advertisement form has been submitted. The code creates an instance of the **AdvertisementForm** with the form data obtained from **request.POST**.
7. The form's validity is checked using the **is_valid()** method. If the form is valid, the code retrieves the selected shop ID, the starting date (current date), the advertisement period, and calculates the ending date by adding the period (in months) to the starting date. A unique identifier for the advertisement (**ida**) is generated using the **uuid.uuid4()** function.
8. An instance of the **Advertisement** model is created with the obtained data and saved to the database.
9. Finally, if the form processing is successful, the code redirects to a success page named 'SuccessAdvertisementRegistration', passing the selected shop ID as a parameter.
10. If the request method is not 'POST' or the form is invalid, the code prepares the context dictionary containing the 'shops_not_adv_yet' list and renders the 'Advertisement.html' template, passing the context as the context data.

In summary, this view function retrieves a list of shops associated with a specific 'idso' from the session, checks which shops have already been advertised, and displays an advertisement form for the shops that have not been advertised yet. When the form is submitted, it processes the data, saves an advertisement instance to the database, and redirects to a success page.

2.2.14 Scan QR

The function **scan_qr** handles the scanning and processing of a QR code.

Let's go through the code step by step:

1. The function begins by checking if the request method is 'POST'. If it is, it means that a form containing the scanned QR code has been submitted.
2. The code retrieves the value of the scanned QR code from the request using **request.POST.get('qrCodeValue')**.
3. The QR code value is then split into lines using the newline character ('\n') as the delimiter, creating a list of lines.
4. The necessary information from the QR code lines is extracted. The 'ids' field represents the ID of the shop, the 'date_str' field represents the date in string format, the 'time_range_str' field represents the time range in string format, the 'number_within_slot' field represents the number within the time slot, and the 'idc' field represents the ID of the customer.
5. The extracted values are further processed and cleaned by removing leading and trailing whitespaces.
6. The date is converted from a string to a **date** object using **datetime.strptime()** with the appropriate format.
7. The start and end times are extracted from the time range string using the '-' delimiter. They are converted to **time** objects using **datetime.strptime()**.
8. The start time and date are combined to create a **start_datetime** object, and the end time and date are combined to create an **end_datetime** object.

9. If the end datetime is earlier than the current datetime, it means the QR code has expired. In such a case, the function renders the 'Expired_qr_code.html' template.
10. The function attempts to retrieve the shop object from the **Shop** model using the 'ids' field.
11. If the shop does not exist, a 'Shop.DoesNotExist' exception is raised, and the function renders the 'error.html' template.
12. The function then attempts to retrieve the time slot object from the **TimeSlot** model. It checks if a time slot with matching shop, start and end times, and date exists.
13. If the time slot does not exist, a 'TimeSlot.DoesNotExist' exception is raised, and the function renders the 'error.html' template.
14. If the time slot exists, the function attempts to retrieve the slot object from the **Slot** model. It checks if a slot with the corresponding time slot and number within slot exists.
15. If the slot is not available (i.e., already occupied), the function retrieves the QR object from the **QR** model using the shop ID, date, time start, time end, and number within slot. It sets the 'scanned' attribute of the QR object to True and deletes the QR object.
16. Additionally, a review object is created using the retrieved shop, customer ID, and shop name. The review object is saved to the database.
17. Finally, if the scanning process is successful, the function renders the 'scan_successful.html' template, passing the review object as the context data.
18. If the request method is not 'POST' or the scanning process encounters an error, the function renders the 'scan.html' template.

In summary, this view function processes a scanned QR code by extracting relevant information, checking for the existence of associated objects (shop, time slot, slot), updating the QR object, creating a review object, and rendering appropriate templates based on the scanning result.

2.2.15 Scan Product

Scan_**product** is the function which handles the scanning and processing of a product QR code for a specific customer.

Let's go through the code step by step:

1. The function first checks if the request method is 'GET'. If it is, it renders the 'Scan_product.html' template, passing the 'idc' (customer ID) as the context data.
2. If the request method is 'POST', it means that a form containing the scanned product QR code has been submitted.
3. The code retrieves the 'idc' (customer ID) from the session using **request.session.get('idc', '')**. This ensures that the customer ID is consistent throughout the process.
4. The customer object is retrieved from the **Account** model using the 'idc' field.
5. The QR code value is obtained from the request using **request.POST.get('qrCodeValue')**.
6. The QR code value is split into lines using the newline character ('\n') as the delimiter, creating a list of lines.

7. The necessary information from the QR code lines is extracted. The 'idp' field represents the ID of the product, the 'name' field represents the name of the product, the 'price' field represents the price of the product, the 'shop_discount' field represents the discount at the shop level, and the 'quantity' field represents the available quantity of the product.
8. The extracted values are further processed and cleaned by removing leading and trailing whitespaces or converting the price to a float.
9. The product object is retrieved from the **Product** model using the 'idp' field.
10. The customer's reward is increased by the price of the purchased product, and the customer object is saved.
11. The quantity of the product is decreased by 1 to reflect the purchase.
12. If the product quantity reaches 0 after the purchase, it means that the products are over, and the product object is deleted from the database.
13. The product and purchased item objects are saved to the database.
14. Finally, if the scanning and purchase process is successful, the function renders the 'scan_successful_purchase.html' template.

In summary, this view function processes a scanned product QR code by extracting relevant information, updating the customer's reward, decreasing the product quantity, and saving the changes to the database. It also handles the scenario where the product quantity reaches zero and renders appropriate templates based on the scanning and purchase result.

2.2.16 Purchase List

The function **Purchase_list** retrieves the purchase list for a specific customer and displays the purchased items along with relevant shop information.

Let's go through the code step by step:

1. The function retrieves the 'idc' (customer ID) from the session using **request.session.get('idc', '')**.
2. The customer object is retrieved from the **Account** model using the 'idc' field.
3. The purchase list object is retrieved from the **PurchaseList** model using the customer object.
4. The purchased items associated with the purchase list are retrieved from the **PurchasedItem** model using the purchase list.
5. Empty lists (**name_shops**, **address_shops**, **prices**) are initialized to store the shop name, shop address, and product prices respectively.
6. The function iterates through each purchased item in the **items** queryset.
7. For each item, the corresponding product object is retrieved from the **Product** model using the 'idp' field.
8. The shop associated with the product is retrieved from the **Shop** model using the 'ids' field.

9. The shop name is appended to the **name_shops** list, the shop address is appended to the **address_shops** list, and the product price is appended to the **prices** list.
10. The **zip()** function is used to combine the **items**, **name_shops**, **address_shops**, and **prices** lists into a single iterable.
11. The function renders the 'Purchase_list.html' template, passing the zipped iterable as the context data.

In summary, this view function retrieves the purchase list for a customer, fetches the purchased items along with associated shop information, and renders the 'Purchase_list.html' template to display the list of purchased items, shop names, shop addresses, and product prices.

2.2.17 Wish list

The **Wish_list** function handles the management of a customer's wish list, including displaying the existing wish list items, allowing the removal of items, and providing the ability to search and add similar products to the wish list.

Let's go through the code step by step:

1. The function retrieves the 'idc' (customer ID) from the session using **request.session.get('idc', '')**.
2. The customer object is retrieved from the **Account** model using the 'idc' field.
3. The wish list object associated with the customer is retrieved from the **WishList** model.
4. The wish list items associated with the wish list are retrieved from the **WishListItem** model.
5. Empty lists (**name_shops**, **address_shops**, **prods**, **prices**, **itemss**) are initialized to store shop names, shop addresses, product names, product prices, and wish list item objects respectively.
6. The function iterates through each wish list item in the **items** queryset.
7. For each item, the corresponding product object is retrieved from the **Product** model using the 'idp' field.
8. The shop associated with the product is retrieved from the **Shop** model using the 'ids' field.
9. The shop name is appended to the **name_shops** list, the shop address is appended to the **address_shops** list, the product name is appended to the **prods** list, the product price is appended to the **prices** list, and the wish list item object is appended to the **itemss** list.
10. The **zip()** function is used to combine the **prods**, **address_shops**, **name_shops**, **prices**, and **itemss** lists into a single iterable.
11. The function initializes variables (**similar_products**, **show_search_bar**, **data**) to manage the search functionality and display of similar products.
12. If the request method is 'POST', the code checks for different actions based on the submitted form data.
13. If the 'Remove' button is clicked, the product ID is retrieved from the request and all wish list items associated with that product ID are deleted from the database.

14. If the 'add_products' button is clicked, it sets the **show_search_bar** variable to **True**, which triggers the display of a search bar on the template.
15. If the 'query' button is clicked, the code retrieves the search query from the request and performs a search for similar products based on the query terms.
16. The similar products found are stored in the **similar_products** list along with their associated shop names and addresses.
17. If the 'selected_product' button is clicked, the selected product ID is retrieved from the request. The corresponding product object is retrieved from the **Product** model, and a new wish list item is created and saved to the database.
18. Finally, the function renders the 'WishList.html' template, passing the relevant context data including the zipped iterable, **show_search_bar**, **similar_products**, and **data**.

In summary, this view function handles the display and management of a customer's wish list. It retrieves existing wish list items, allows the removal of items, provides a search functionality for finding similar products, and allows customers to add selected products to their wish list.

2.2.18 Edit Product

The function **Edit_product** is responsible for editing product information in a shop's inventory.

Here's a breakdown of the code:

1. The function retrieves the 'idso' (shop ID) from the session using **request.session.get('idso', '')**.
2. The shop objects associated with the 'idso' are retrieved from the **Shop** model.
3. Two variables, **editing** and **re**, are initialized with values of 0.
4. If the request method is 'POST', the code checks for different actions based on the submitted form data.
5. If the 'btn1' button is clicked, the selected shop ID is retrieved from the request. The products associated with the selected shop are retrieved from the **Product** model, and **re** is set to 1 to indicate that the products should be displayed on the template.
6. If the 'btn2' button is clicked, **editing** is set to 1, indicating that a specific product is being edited. The selected product ID is retrieved from the request, and **re** is set to 1 to indicate that the editing form should be displayed on the template.
7. If the 'btn4' button is clicked, the code retrieves the updated product information from the request, including the new name, price, quantity, discount, and product ID.
8. The corresponding product object is retrieved from the **Product** model using the 'idp' field.
9. The product's attributes (name, price, quantity, shop_discount) are updated with the new values.
10. The product is saved to the database.
11. The function redirects the user to the 'Product_view' view, which presumably displays the updated product information.

12. If none of the above conditions are met (i.e., the request method is not 'POST'), the function renders the 'EditProducts.html' template, passing the relevant context data including the shop objects, **editing**, and **re**.

In summary, this view function handles the editing of product information in a shop's inventory. It allows the selection of a shop, displays the products associated with the selected shop, enables editing of a specific product's information, and saves the updated product information to the database.

2.3 models.py

In Django, models.py is used to define the structure and behavior of database tables. Each model class represents a table in the database, and the fields within the class correspond to the columns in the table.

All the models have been defined according to the “UML rquirements” document.

For the creation of the models, "Many-to-Many" and "Foreign Key" relations have been used.
A Many-to-Many relationship is a relationship where a record in one table can be associated with multiple records in another table, and vice versa.
For example, consider the **Shop** and **Product** models. A shop can sell many products, and a product can be sold in many shops. This is a Many-to-Many relationship.
A Foreign Key relationship is a relationship where a record in one table is linked to a record in another table through a foreign key.
For example, consider the **PurchaseList** and **Account** models. A purchase list is associated with a single customer account. This is a Foreign Key relationship where **PurchaseList** is the child model and **Account** is the parent model.

2.3.1 Account

Represents user accounts in the application, which can be either customers or shop owners. It contains fields for account information such as name, surname, password, email, birthday, and identification (idc or idso).Provides a method user_login() for user login functionality.

2.3.2 Shop

It represents a shop in the application. It contains fields for shop details such as name, image, maximum number of clients, queue information, address, rating, and category.
This class has a relationship with the Account model via idso to associate the shop with its owner (ShopOwner).
In addition, the following relationships with other models are present:

- One-to-Many relationship with TimeSlot model to define the time slots available for booking in the shop.
- One-to-Many relationship with Product model to represent the products available in the shop.
- One-to-Many relationship with Review model to store customer reviews for the shop.

2.3.3 Product

It represents a product available in a shop and contains fields for product details such as name, price, shop discount, quantity, availability, and associated shop (via ids).

It has a relationship with the Account model via idso to associate the product with its owner (ShopOwner).

2.3.4 Timeslot

The **TimeSlot** model allows you to define and manage time slots for a shop. Each time slot has a specific start and end time, a date, and an availability status. The model also maintains a relationship with the **Shop** model, enabling you to associate time slots with their respective shops.

2.3.5 Slots

The **Slot** model allows you to define and manage individual slots within a time slot. Each slot has a specific number, availability status, and is associated with a particular time slot. The **idc** field provides a way to identify or label each slot if needed.

2.3.6 QR

The **QR** model allows you to store and manage QR codes along with their relevant information such as identifiers, dates, times, and scanning status. The **img** field stores the actual image of the QR code encoded in base64 format.

2.3.7 Advertisement

The **Advertisement** model allows you to store and manage advertisements along with their relevant information such as start and end dates and identifiers. You can use this model to track and display advertisements based on their duration and associated identifiers.

2.3.8 Review

The **Review** model allows you to store and manage customer reviews for a shop. It captures the review content, date, associated shop, and other relevant information.

2.3.9 Purchase List

The **PurchaseList** model allows you to link a customer account with their purchase list, providing a way to track and manage their purchases.

2.3.10 Purchase Item

The **PurchasedItem** model represents an item that has been purchased and is associated with a specific purchase list. It has fields that store information about the purchased item, such as the date of purchase, a unique identifier, the item's name, the shop where it was purchased, the shop's address, and the price of the item.

2.3.11 Wishlist

The **WishList** model represents a wishlist associated with a specific user account. It has a field that stores the relationship between the wishlist and the user account using a foreign key.

2.3.12 Wishlist Item

The WishListItem model represents an individual item within a wishlist. Each item is associated with a specific wishlist using a foreign key relationship.

The wish_list field establishes the connection to the WishList model by utilizing a foreign key. This field links each item to its corresponding wishlist. If the wishlist is removed or deleted, the associated items will also be removed.

2.4 forms.py

The forms.py file in Django is commonly used to define form classes. These form classes provide a convenient way to handle form input and validation in web applications. They are responsible for rendering HTML forms, processing user input, and validating the submitted data.

- **The RegistrationForm** is a form for user registration, containing fields such as first_name, last_name, password, email, and birthday. It collects user information for creating a new account.
- **The LogInForm** is used for user login and includes fields for password and email. Users can input their credentials to authenticate and access their accounts.
- **The ShopForm** is a form for creating a new shop. It includes fields such as name, max_numb_clients, address, number, CAP, City, category, opening_time, closing_time, slot_duration, and img. These fields gather information about the shop, including its details, category, operating hours, and an optional image.
- **The ProductForm** represents a form for adding a product. It consists of fields such as name, price, and shop_discount. These fields collect information about the product, such as its name, price, and any applicable shop discounts.
- **The ShopCategorySelectionForm**, **Shop_and_day_selectionForm**, and **TimeSlot_selectionForm** are specialized forms with hidden fields. They are used for specific purposes related to shop and time slot selection, allowing for specific interactions and data handling within the application.
- **The AdvertisementForm** is a form used for creating advertisements. It includes a field for period to specify the duration of the advertisement.

Overall, the provided code defines various forms used for different functionalities within a Django application, allowing users to input and submit data for registration, login, shop creation, product addition, advertisement creation, and other specific interactions within the application.

2.5 Templates

The main role of templates in Django is to facilitate the rendering of dynamic data within HTML pages. Templates use a combination of HTML markup and special template syntax (such as variables, loops, conditionals, and filters) to dynamically generate the final HTML output. They allow developers to define placeholders within the HTML code, which are then populated with actual data when the template is rendered.

In our projects, we coded more than 40 html files, but for making the anlysis cleare, we decided to report only the important templates.

2.5.1 Advertisment Template

The **advertisement template** starts with the <head> section, which includes metadata and external dependencies such as stylesheets and fonts. Inside the <head>, there is a JavaScript function called Redirect() defined, which is responsible for redirecting the user to a specific URL.

Moving on to the <body> section, the template begins with two paragraphs providing instructions and information to the user. Following that, the template utilizes Django template syntax to conditionally display a form or a message based on the availability of shops.

If there are shops available, a form is rendered using the <form> tag. This form allows the user to select a shop to advertise and choose the duration of the advertisement. The shop options are dynamically generated using a loop over the shops variable, which is passed from the Django view to the template. The form includes a CSRF token for security and uses the HTTP POST method for submission.

If there are no shops available, a message is displayed indicating that there are no shops to advertise. After the form or message, a paragraph is included to inform the user that the advertisement period starts from the current time.

Towards the end of the template, there are two buttons displayed at the bottom of the page. The first button is labeled "Going back to ShopOwner view" and has a CSS class of "GoingBackButton". Clicking on this button triggers the Redirect() function, redirecting the user to the appropriate URL. The second button is labeled "LogOut" and has a CSS class of "anybutton". Clicking on this button also triggers the Redirect() function to log the user out.

This template serves as the visual representation of the advertisement functionality in the application, allowing users to select a shop, specify the advertisement period, and perform actions based on their choices.

2.5.2 Booking Template

The template is responsible for rendering a page related to slot booking functionality. Let's break down the structure and purpose of the different sections in the code:

The template starts with the <head> section, which includes metadata and external dependencies such as stylesheets, fonts, and JavaScript libraries. It loads custom tags and static files using Django template syntax.

Moving on to the <body> section, it begins with an <h1> heading displaying "Slot Booking". Following that, there is a <form> element used for submitting slot booking related data. Inside the form, there is a conditional block using Django template syntax to check if there are any shops available.

If shops are available, a dropdown menu is displayed for selecting a shop. The options are dynamically generated using a loop over the shops variable. Below the dropdown menu, a <div> element with the ID "map" is present, which will be used to display a map later in the code.

Next, there is another <div> element with the ID "shopAddress" which will be used to display shop addresses. After that, a label and an input field for selecting a date are provided.

At the end of the form, there is a submit button labeled "View the available slots". The form is closed after this button.

After the form, there is another conditional block that checks if there are any available time slots. If there are, an <h2> heading is displayed with the text "Available slots:". A <ul> (unordered list) is used to display each available time slot using a loop over the timeslots variable.

If there are no available time slots, an <h1> heading is displayed with the text "No slots available".

Another conditional block follows, which checks if there are available time slots. If there are, an <h1> heading is displayed with the text "Select a slot". Another <form> element is present inside this block for submitting the selected slot. Similar to the previous form, it includes a CSRF token and uses a loop over the timeslots variable to generate options for selecting a slot.

Towards the end of the template, there is a conditional block that checks if there are any shops registered yet. If there are no registered shops, an <h1> heading is displayed with the text "No shops registered yet".

Finally, there are two buttons displayed at the bottom of the page. The first button is labeled "Going back to Customer view" and has a CSS class of "GoingBackButton". Clicking on this button triggers the Redirect() function, redirecting the user to the appropriate URL. The second button is labeled "LogOut" and has a CSS class of "anybutton". Clicking on this button also triggers the Redirect() function to log the user out.

In the <script> tag, there is JavaScript code responsible for handling various functionalities.

First, it sets up the map by creating a Leaflet map object and configuring its initial view. The map is displayed on the webpage using the specified HTML element with the ID "map". The code also adds a tile layer from OpenStreetMap to provide the underlying map data.

Next, the code checks if the user's browser supports geolocation. If geolocation is supported, it retrieves the user's current position using the **getCurrentPosition()** method. The latitude and longitude coordinates are used to center the map view, and a marker with a custom red icon is created and added to the map at the user's location.

The JavaScript code then deals with displaying shop markers on the map. It uses a loop to iterate over the shop data, which includes information such as the shop's address, name, queue status, and review rating. For each shop, the code makes use of the Leaflet Control.Geocoder plugin to perform geocoding, which converts the shop's address into latitude and longitude coordinates.

If the geocoding process successfully retrieves results for a shop's address, a marker is created at the corresponding coordinates and added to the map. The marker is customized to display a popup containing information about the shop, including its name, queue status, and review rating.

However, if geocoding fails to retrieve results for a shop's address, an error message is logged, and the shop's address is recorded in a separate HTML element with the ID "shopAddress". This element is used to display the addresses of shops that couldn't be shown on the map due to geocoding issues.

The JavaScript code concludes by updating the "shopAddress" element to display the list of shop addresses that couldn't be shown on the map, along with a message indicating the absence of addresses if all addresses were successfully displayed.

2.5.3 Scan QR

The provided HTML code represents a template for a web page designed to facilitate the scanning of QR codes. Let's discuss its structure and functionality in more detail:

The HTML document starts with a doctype declaration and sets the language of the page to English. It then includes the necessary dependencies by loading CSS stylesheets from W3Schools and Google Fonts. Additionally, a custom CSS file is referenced using the **{% static %}** template tag, indicating that the path to the file will be determined dynamically.
Moving on to the body of the document, it begins with a **<div>** element with the ID "scanner-container". This container is intended to hold the video feed from the device's camera, which will be used for QR code scanning.
A **<script>** tag follows, referencing an external JavaScript file called "instascan.min.js". This file provides the required functionality for scanning QR codes using the device's camera.
Inside the **<script>** tag, the JavaScript code starts by declaring a variable named **scanner** and setting it to **null**. This variable will later store a reference to the QR code scanner object.
The **startScanning()** function is defined, which will be called when the page finishes loading. This function dynamically creates a **<video>** element and assigns it the ID "scanner-video". The video element is then appended to the "scanner-container" **<div>**.
The function continues by creating an instance of the **Instascan.Scanner** class, passing the video element as the configuration option to use the video feed as the source for scanning QR codes.
The scanner object is assigned an event listener for the 'scan' event, which triggers when a QR code is successfully scanned. Inside the listener, the content of the scanned QR code is passed to the **sendQRCodeValue()** function.
To retrieve the available cameras on the device, the code calls the **getCameras()** method from the **Instascan.Camera** object. If cameras are found, the scanner is started using the first available camera. Otherwise, an error message is logged to the console.
The **sendQRCodeValue()** function is responsible for dynamically creating a form element. This form is configured with the necessary attributes, including the HTTP method set to "post" and the action URL set to '/ShopOwner/Scan/'. The function then creates two hidden input fields within the form: one for the CSRF token, and the other for the scanned QR code value. After populating the form, it is appended to the document's **<body>**. Finally, the form is submitted programmatically.
At the end of the HTML document's body, two **<input>** elements of type "button" are included. These buttons enable the user to navigate back to the ShopOwner view or log out. They have specific CSS classes assigned to them for positioning and visual styling.

3. Requirements Implemented in the application

In this section, we will examine the requirements catalogue and determine which requirements have been successfully implemented.

Almost all the methods have been implemented in the view.py file in Django.

Firstly, the list of the requirements is here reported, identifying which of them has been addressed in the implementation. Next, we will depict how and why certain requirements has been selected to be satisfied by our application, starting from the structure of the UML architecture.

3.1 List of implemented requirements

Table 1: List of implemented requirements

Number	Description
R1	Iqueue must allow the customers to see the number of people in the queue at a specific shop
R2	Iqueue shall allow the shop owners to insert the data of their activity
R3	Iqueue shall allow the shop owners to insert special offers for their shop
R4	Iqueue must read the QR code of the clients with the app
R5	The app shall allow customers to track their rewards
R6	Iqueue must allow a customer to book a time slot on the app to visit the shop
R7	Iqueue must keep track of the customer position
R10	The app shall allow customers to create and save shopping lists or wish lists.
R11	The app shall show the customer's purchase history and receipts
R12	The app will be able to divide the registered facilities into categories
R13	The app will permit the shop owner to insert their products/services
R14	The app will enable shop owners to insert the price of their products/services
R15	The app will enable shop owners to modify the price of their products/services
R16	Iqueue must be able to generate the client QR code
R17	Iqueue must show the available time slots of a shop
R18	Iqueue must permit a customer to delete his booked time slot
R19	The app will show the registered shops on a map/list
R21	Iqueue shall permit the shop owners to delete the data of their activity
R22	Iqueue shall allow the shop owners to modify special offers for their shop
R23	Iqueue shall allow the shop owners to delete special offers for their shop
R24	The app must be able to track customer rewards and loyalty points
R25	Iqueue must allow user registration
R26	Iqueue must allow the user to operate as customer or as shop owner
R27	Iqueue must allow log-in as customer
R28	Iqueue must allow log-in as shop owner
R29	Iqueue must allow log-out
R30	The app will enable shop owners to remove their products/services
R31	The app will enable shop owners to count their available products/services
R32	Iqueue will be able to realize to queue counting by means of the QR codes
R33	Iqueue will allow the shop owner to manually increment or decrement the queue counter
R34	Iqueue shall allow the customer to select the category of the shops he is interested to visit
R35	Iqueue shall allow the customer to select a specific shop
R36	Iqueue shall allow the customer to select a specific shop product/service

R38	The app shall allow the shop owner to advertise their shop
R39	The app shows the user of the interested queue variation
R40	Iqueue shall give the shop address to a third-party GPS system to guide the client to the shop
R41	Iqueue will allow the client to provide feedback and rating to the shops

3.2 Requirements in the implementation

Now, we will discuss where starting from the UML architecture, the root for the implementation development, the requirements have been satisfied.

To have a better understanding of the UML architecture, please read the related document.

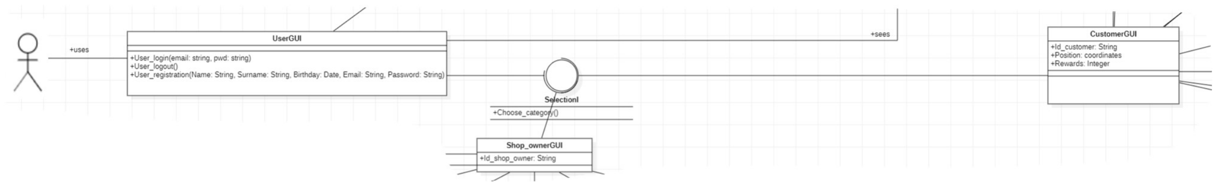


Figure 3: Relationships among User, Shop_owner and Customer GUI.

- User_login(in email:string, in pwd:string)

This method has been implemented in Django in the function **login_view.**

***Requirements*: R27, R28**

- User_registration(in Name:String, in Surname:String, in Birthday:Date, in Email:String, in Password:String)

This method has been implemented in Django in the function **registration_view**.

***Requirements*: R25**

- User_logout()

This method has been implemented in Django html templates through a button which allows the user to go back to the log in page.

***Requirements*: R29**

- Choose_category()

Choose_category is a requirement which has been implemented with the function “SelectRole”, which allows the user to select desired role.

***Requirements*: R27, R28, R26**

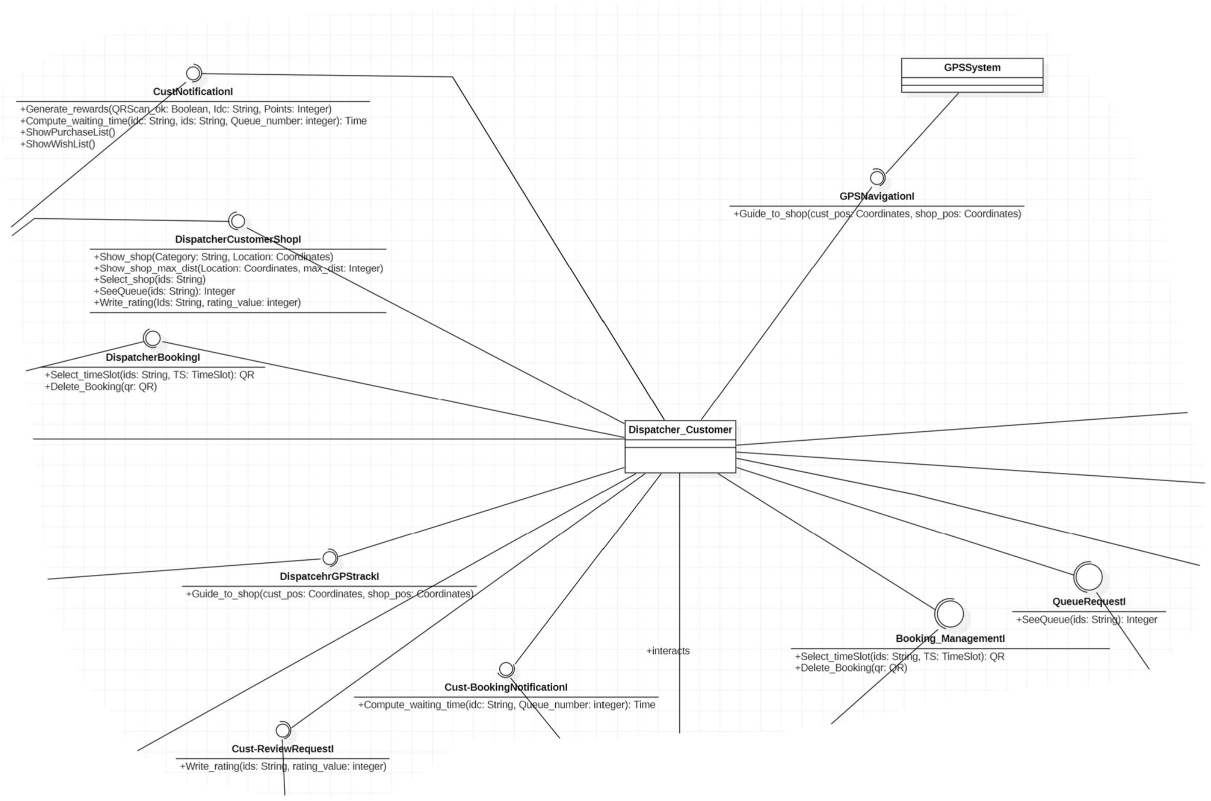

CustNotificationI
+Generate_rewards(QRScan_ok: Boolean, Idc: String, Points: Integer)
+Compute_waiting_time(idc: String, ids: String, Queue_number: integer): Time
+ShowPurchaseList()
+ShowWishList()
GPSSystem
GPSNavigationI
+Guide_to_shop(cust_pos: Coordinates, shop_pos: Coordinates)
DispatcherCustomerShopI
+Show_shop(Category: String, Location: Coordinates)
+Show_shop_max_dist(Location: Coordinates, max_dist: Integer)
+Select_shop(ids: String)
+SeeQueue(ids: String): Integer
+Write_rating(Ids: String, rating_value: integer)
DispatcherBookingI
+Select_timeSlot(ids: String, TS: TimeSlot): QR
+Delete_Booking(qr: QR)
Dispatcher_Customer
DispatcehrGPStrackI
+Guide_to_shop(cust_pos: Coordinates, shop_pos: Coordinates)
QueueRequestI
+SeeQueue(ids: String): Integer
Booking_ManagementI
+Select_timeSlot(ids: String, TS: TimeSlot): QR
+Delete_Booking(qr: QR)
+interacts
Cust-BookingNotificationI
+Compute_waiting_time(idc: String, Queue_number: integer): Time
Cust-ReviewRequestI
+Write_rating(ids: String, rating_value: integer)

Figure 4: Dispatcher customer relationships

- Show_shop(Category: string, Location: coordinates)

This function has been implemented within the function **booking** and the html file **bookig.html.** In booking, we filter the shops according to the shop category selected by the customer and, then, they are shown on the map with their queue and rating. The map is implementented in javascript, using OpenStreetMaps API.

Requirements: R12, R19, R34, R39

- Show_shop_max_dist(in Location:Coordinates, in ids:String)

This method in the current version of the app has not been implemented, since the implemented map already permits to identify the distance between customer and shops. Anyway, it will be released in future updates.

- Select_shop(ids: String)

This function is implemented in the **booking** function, when the user is able to selcet the shop form the menu. The graphic interface is implemented in **booking.html**

Requirements: R35

- See_Queue(ids: String)

This function is implemented thanks to the method **checkQueue** of the class **Shop**, used in the booking.html to see the actual queue.

***Requirements*: R1, R32**

- Write_ratings(ids: String, rating_value: integer)

This function has been implemented in **write_review** function. This function allows the customer to write a review of the shop in which they have bought a product.

Requirements: R41

- Select_timeSlot(ids: String, TS: TimeSlot)

This function has been implemented in the function **booking** with the corresponding html file **booking.html**

Requirements: R6, R17

- Delete_Booking(qr: QR)

This function has been implemented in the function DeleteQR with its corresponding html file **DeleteQR.html**

Requirements: R18,

- Guide_to_shop(cus_pos: Coordinates, shop_pos: Coordinates)

This function has been implemented in the html file **CustomerReservations.html** where Google Maps API allows the customer to navigate towards the shop of the reservation.

Requirements: R40

- Generate_rewards(QRScan_ok: Boolean, idc: String, Points: Integer)

This function has been implemented in **Scan_product**. Once the product is scanned, a rewards equals to the price of the item is added to the virtual wallet of the the client.

Requirements: R5, R24

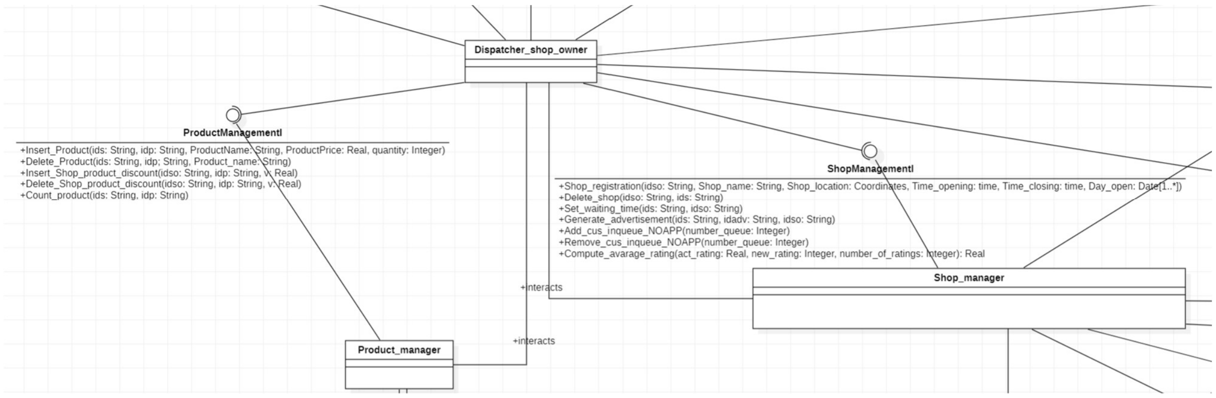

Dispatcher_shop_owner
ProductManagementI
+Insert_Product(ids: String, idp: String, ProductName: String, ProductPrice: Real, quantity: Integer)
+Delete_Product(ids: String, idp; String, Product_name: String)
+Insert_Shop_product_discount(idso: String, idp: String, v: Real)
+Delete_Shop_product_discount(idso: String, idp: String, v: Real)
+Count_product(ids: String, idp: String)
ShopManagementI
+Shop_registration(idso: String, Shop_name: String, Shop_location: Coordinates, Time_opening: time, Time_closing: time, Day_open: Date[1..*])
+Delete_shop(idso: String, ids: String)
+Set_waiting_time(ids: String, idso: String)
+Generate_advertisement(ids: String, idadv: String, idso: String)
+Add_cus_inqueue_NOAPP(number_queue: Integer)
+Remove_cus_inqueue_NOAPP(number_queue: Integer)
+Compute_avarage_rating(act_rating: Real, new_rating: Integer, number_of_ratings: Integer): Real
+interacts
Shop_manager
+interacts
Product_manager

Figure 5: Relationships between ShopManagementI and ProductManagementI.

- Shop_registration(in idso:String, in Shop_name:String, in Shop_location:Coordinates, in Time_opening:time, in Time_closing:time, in Day_open:Date[1..*])

This method has been implemented in the function **registration_view**, which creates the object shop with the data received from the html template **ShopRegistration.html**.

Requirements: R2

- Delete_shop(in Idso:String)

This method has been implemented in the function **DeleteShop.**

Requirements: R21

- Generate_advertisement(ids: String, idadv: String, idso: String)

This function has been implemented in the fucntion **Advertisement_shop**, with its corresponding html file **Advertisement.html**

Requirements: R38

- Add_cus_inqueue_NOAPP(in number_queue:Integer)

This function has been implemented in the **MyShops_view** and the relative html page.

Requirements: R33

- Remove_cus_inqueue_NOAPP(in number_queue:Integer)

This function has been implemented in the **MyShops_view** and the relative html page.

Requirements: R33

- Compute_average_rating(act_rating: Real, new_rating: Integer, number_of_ratings: Integer)

This function has been implemented in **write_review** each time a customer makes a review of the shop.

Requirements: R41

- Insert_Product(ids: String, idp: String, ProductName: String, ProductPrice: Real, quantity: Integer)

This function has been implemented in the function **Product_view** with its template ProductRegistration.html

Requirements: R13, R14, R31

- Delete_Product(ids: String, idp; String, Product_name: String)

This function has been implemented within the function **MyShops_view** with its corresponding file **MyShops.hmtl.**

Requirements: R21,30

- Insert_Shop_ product_discount(idso: String, idp: String,v: Real)

This function has been implemented in both **Product_view** and **Edit_product** where the shop owner can insert the discount at the registration of the product and modifying it corrispectively.

Requirements: R3, R22

- Delete_Shop_product_discount(idso: String, idp: String,v: Real)

This function has been implemented in the function **Edit_product**, where the shop_owner can set back the discount to zero.

Requirements: R21, R23

- Count_product(ids: String, idp: String)

This function has been implemented in **Product_view**

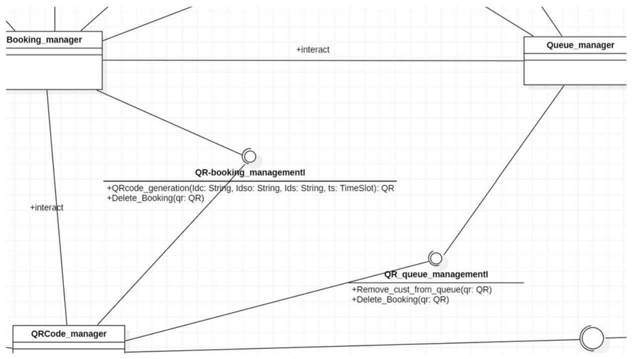


Figure 6: Relationships among QR-booking_managementI and QR_queue_managementI

- QRcode_generation(in idc:String, in idso:String, in ids:String, in ts:TimeSlot): QR

This method has been implemented in the function **QRcode_generation**.

Requirements: R7, R16

- Add_cust_in_queue(in qr:QR)

This method has been implemented through the method **checkQueue** in the class **Shop.** In the app, this function takes the actual time and date, retrieve the corresponding timeslot, and compute the queue based on the bookings registered in the database.

Requirements: R32

- Remove_cust_from_queue(in QRScan_ok:Boolean)

This function has been implemented by the function **scan_qr.** This function takes the data stored in the QR code of the booking and is responsible to make the customer exit the queue.

Requirements: R32

- Sent_notification(in idso:String, in idqr:String)

This method has not been implemented because we thought that it could have created confusion to the shop owner, especially during booking peak time when booking registrations are numerous.

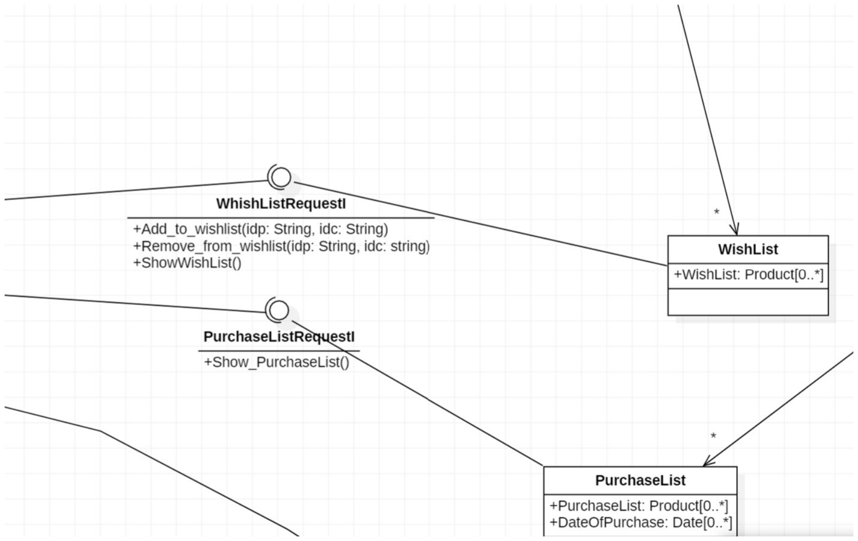

WhishListRequestI
+Add_to_wishlist(idp: String, idc: String)
+Remove_from_wishlist(idp: String, idc: string)
+ShowWishList()
WishList
+WishList: Product[0..*]
*
PurchaseListRequestI
+Show_PurchaseList()
PurchaseList
+PurchaseList: Product[0..*]
+DateOfPurchase: Date[0..*]
*

Figure 7: Relationships among WhishList RequestI and PurchaseListRequestI

- Add_to_wish(idp: String, idc: String)

This function has been implemented in the function **Wish_list** with its corresponding html file **Wishlist.html**

Requirements: R10, R36

- Remove_from_wishlist(idp: String, idc: String)

This function has been implemented in the function **Wish_list** with its corresponding html file **Wishlist.html**

Requirements: R10

- Show_WishList()

This function has been implemented in the function **Wish_list** with its corresponding html file **Wishlist.html**

Requirements: R11

- Show_PurchaseList()

This function has been implemented in the function **Purchase_list** with its corresponding html file **Purchase_list.html**

Requirements: R11

- AddPurchase(in idp:String, in idc:String)

This function has been implemented in the **Scan product view**.

Requirements: R4

- QRCodeScan(in qr:QR): Boolean

This function has been implemented by means of the **Scan QR view** and the relative html page.

Requirements: R4

- Generate_Review(in ids:String, in idc:String, in date:Date): Review

This function has been implemented in the view Scan QR view.

Requirements: R41

By means of the view Edit Product and the relative html page, we are able to satisfy the **Requirement R15** (the edit product method was not reported in the UML diagrams to lighten a bit the number of methods reported).

In addition to the previoulsy mentioned requirements estiblished in the RASD document, the team consider worthy to report the new Iqueue functionality to scan products and automatically diminish the product quantity in the store and added the scanned product to the customer purchaselist.

4. User Manual

The user manual aims to assist users of the Iqueue application, further clarifying the actions that can be undertaken and how with the Iqueue application. These manuals act as a one-stop portal where end-users of a website can get prompt help, dissimilar to emails and conventional calls that take hours or days to reply to.

This user manual works as a reference that can be used also to guide the product support team, permits lower calls and enriches user experience.

4.1 Login page

On the login page, a user enters his e-mail address (field [1]) and the password (field [2]) that he used to register in Iqueue. Then, he presses sign-in [3] to access the application.

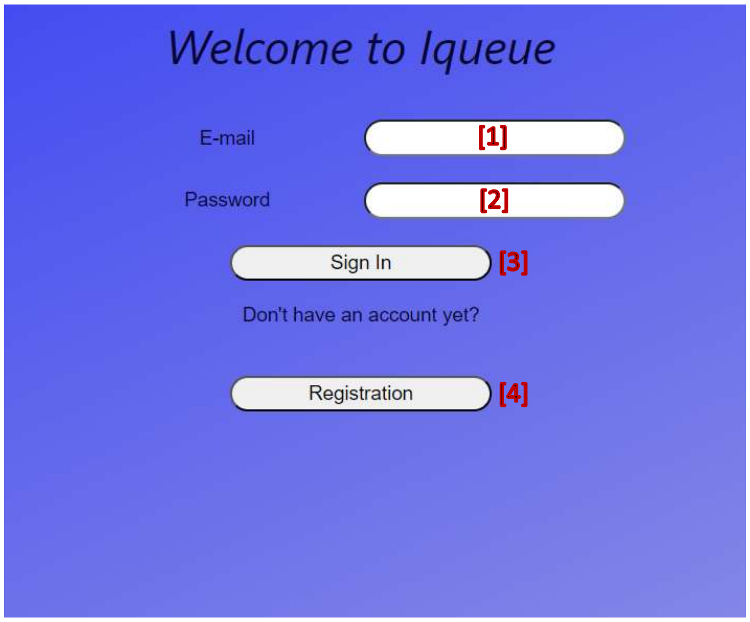


Figure 8: User guide – Login page.

If a user does not have an account, he must register by pressing the registration button [4].

4.2 Registration page

On the registration page, a new user can insert his information as reported in the next figure. It is important to say that a user cannot register again with the same e-mail address.

After the registration, the user is redirected to the *login page*.

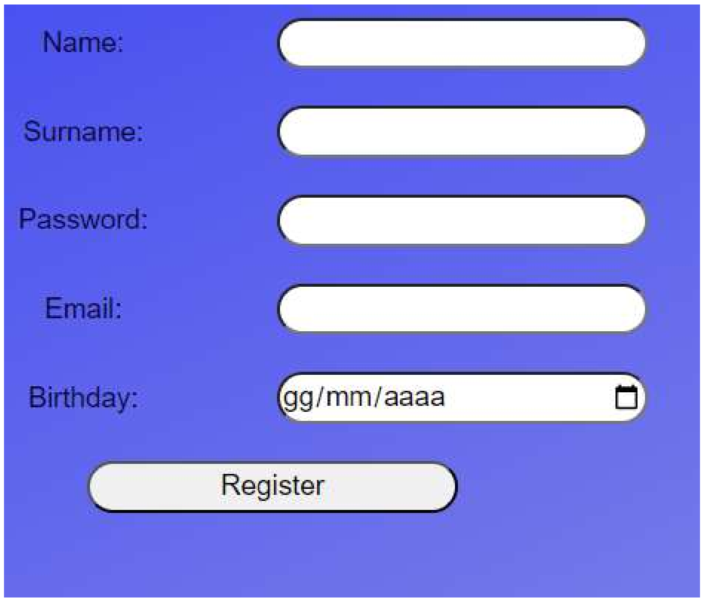


Figure 9: User guide – Registration page.

4.3 Select role page

From this page, the user can select to act as a shop owner or customer by pressing the respective button, respectively [1] and [2].

In the field [3] are reported the information associated with the actual user customer ID and shop-owner ID. The customer ID is relevant for a customer because after he has made a reservation, the sop-owner will call the people in the queue based on their names and customer IDs.

On this page is also reported the information of the advertised shops: in particular their name, image and address. More information on advertisements will be provided in the *Advertisement page* Section.

From this view, the user can log out and be redirected to the initial *login page* by pressing the button [5].

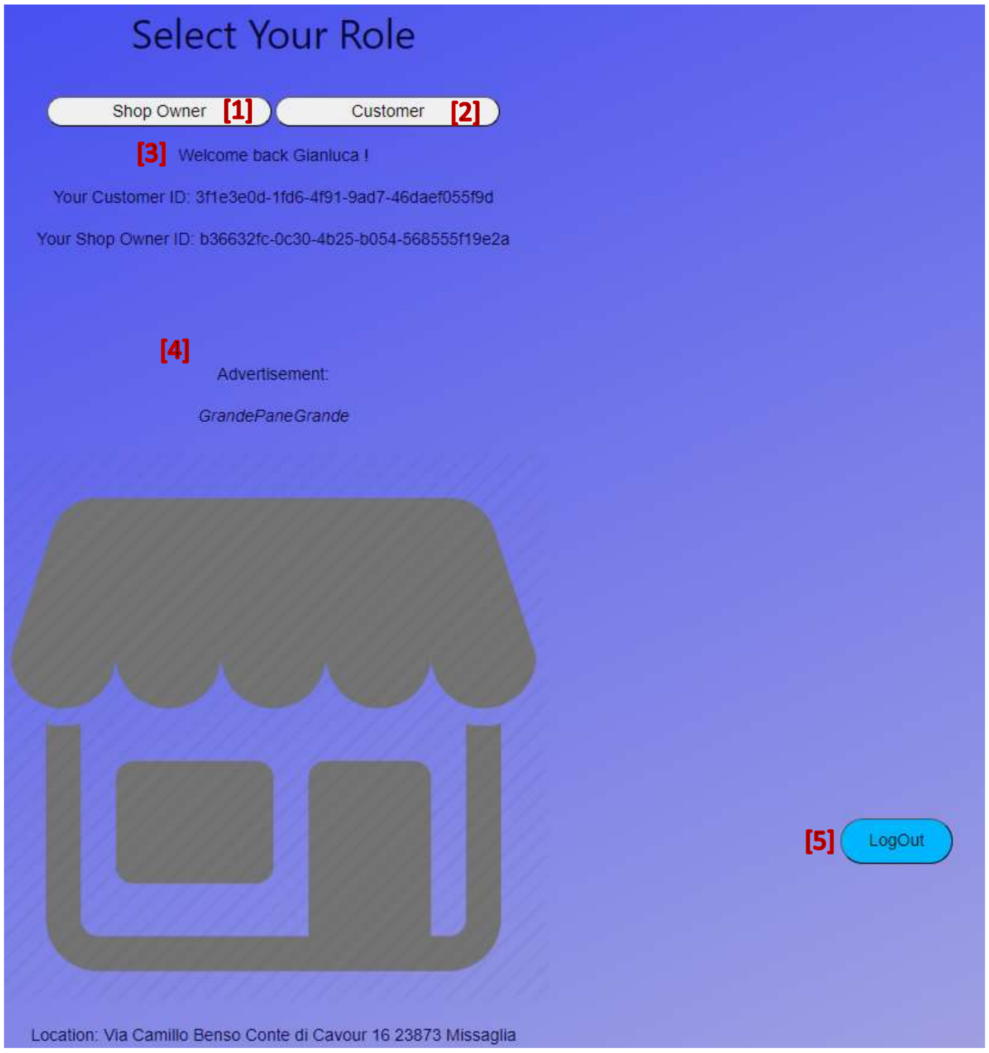

Select Your Role
Shop Owner [1]
Customer [2]
[3] Welcome back Gianluca !
Your Customer ID: 3f1e3e0d-1fd6-4f91-9ad7-46daef055f9d
Your Shop Owner ID: b36632fc-0c30-4b25-b054-568555f19e2a
[4]
Advertisement:
GrandePaneGrande
[5] LogOut
Location: Via Camillo Benso Conte di Cavour 16 23873 Missaglia

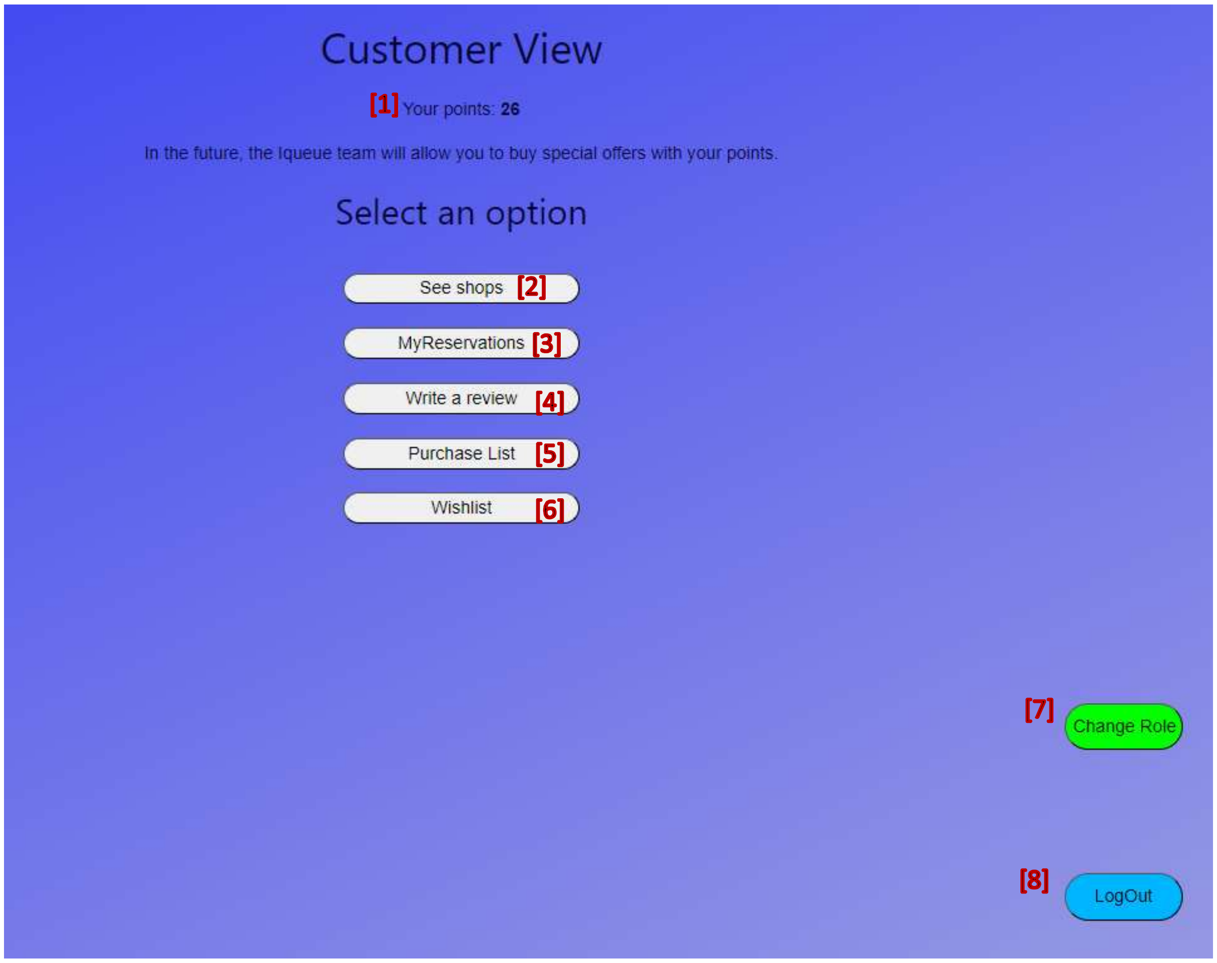
Figure 10: User guide – Select Role page.

4.4 Customer main page

On this page, a shop customer can take his actions that are the following:

- See shops [2]: by pressing this button, the customer can start the procedure that will allow him to see the shops registered in Iqueue and eventually make a reservation. After pressing the button, the customer is redirected to the select *shop category page*.
- MyReservations [3]: by pressing this button, the customer is redirected to *Myreservations page*, where he can manage his reservations.
- Write a Review [4]: by pressing this button, the customer is redirected to the *Reviews page*, where he can manage his reviews.
- Purchase List [5]: by pressing this button, the customer is redirected to the *Purchase List page*, where he can see the products and services he bought in the shops.
- Wishlist [6]: by pressing this button, the customer is redirected to the *Wishlist page*, where he can add/remove products and services to his wishlist.

From this view, the customer can also see the reward points [1] he gained through his Iqueue experience: when the customer goes to a shop, his points increase by the prices of the products he buys, hence scanned by the shop owner.


Customer View

[1] Your points: **26**

In the future, the Iqueue team will allow you to buy special offers with your points.

Select an option

Figure 11: User guide – Customer Main page.

From this page, it is possible to go back back to the *Select Role Page* by pressing the button [7] or to the *login page* by pressing the button [8].

4.5 Shop category page

The customer from this page can select the category of the shops that he wants to inspect. The categories are the ones below reported. In the category others, the customer can find categories of shops apart from the ones above (e.g. clubs).

After pressing the button [1], the customer is redirected to the map where he can proceed with the booking or simply inspect the shops.

From this page, the customer can go back to the *Customer main page* or the *login page* by pressing their respective buttons.

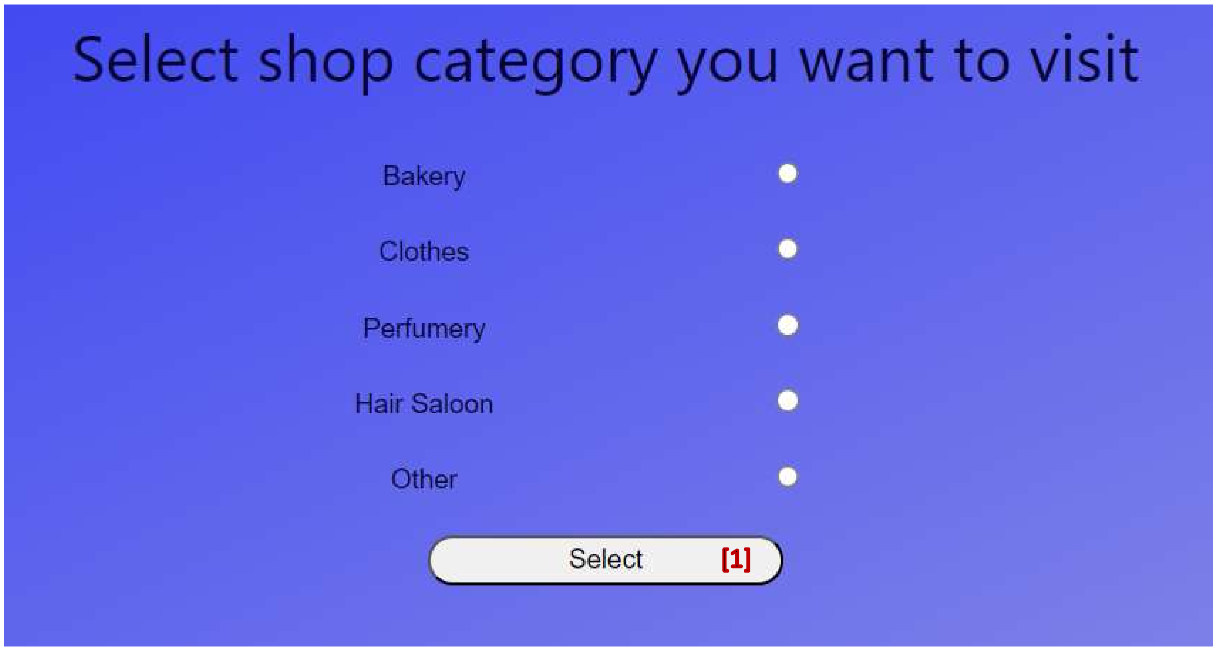


Figure 12: User guide – Shop category page.

4.6 Booking page

From this page, the customer can see the registered shops of the selected category on the map [8]. Directly on the map, the customer can see not only the shop position but also their name, actual queue and rating [3]. The information about the shops unable to be reported on the map by the geolocalization is furnished below the map in the section [4]. The customer position is the one reported by its icon [2].

A customer can select the shop for which he wants to make a reservation by selecting it in the selection bar [1]. Then, he can see the available slots for a certain date by inserting this date in the bar [5] and pressing the button [6]. The user can now see the available slots of the shops that he has selected in the section [7].

If slots are available, the section [9] appears and the customer can select a slot from the menu bar [10]. The customer can then confirm his reservation by pressing the button [11]: after doing so, it is redirected to the *QR confirmation page*.

Also from this page, the customer can go back to the *Customer main page* or the *login page* by pressing their respective buttons.

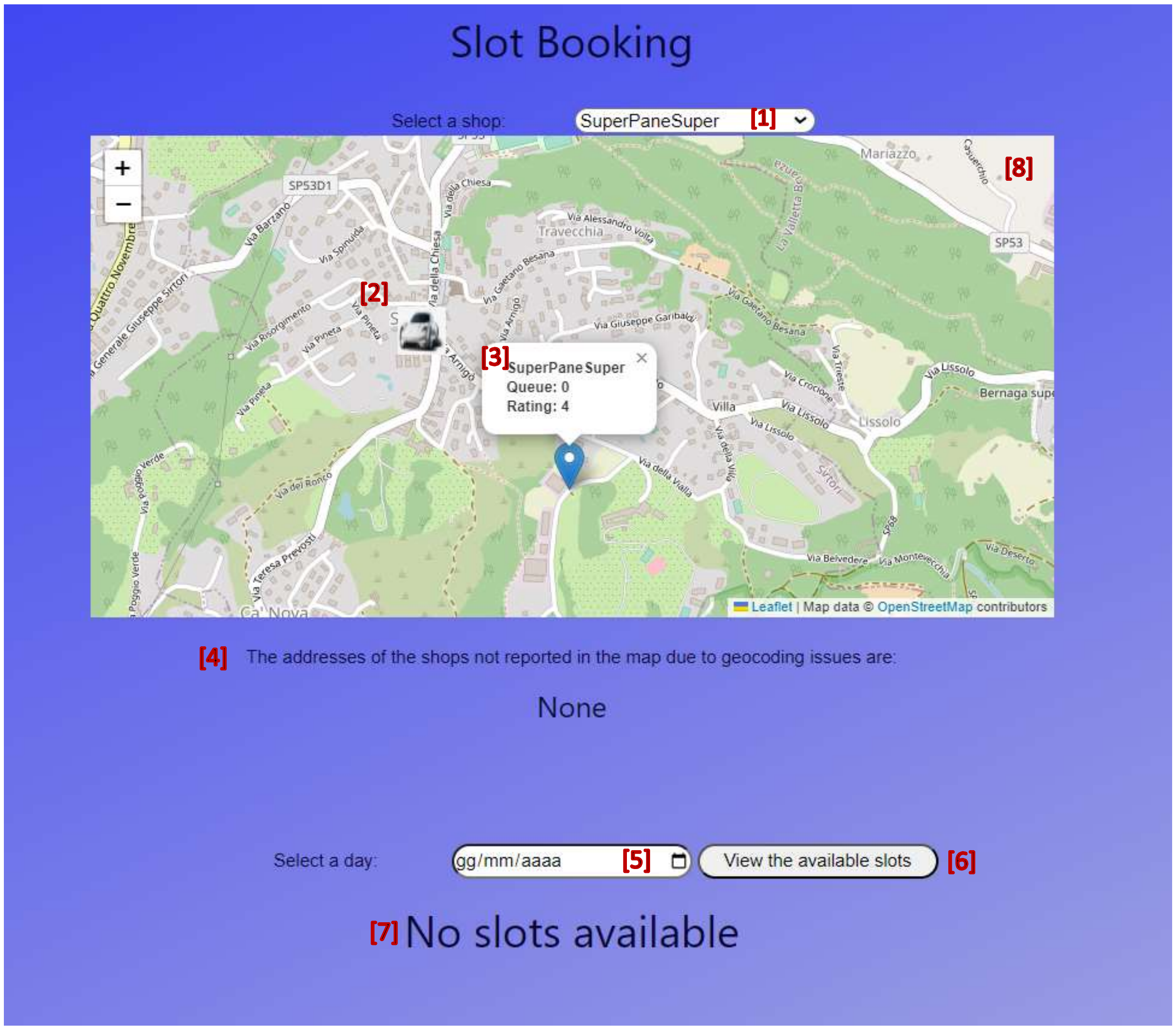


Figure 13: User guide – Booking page 1.

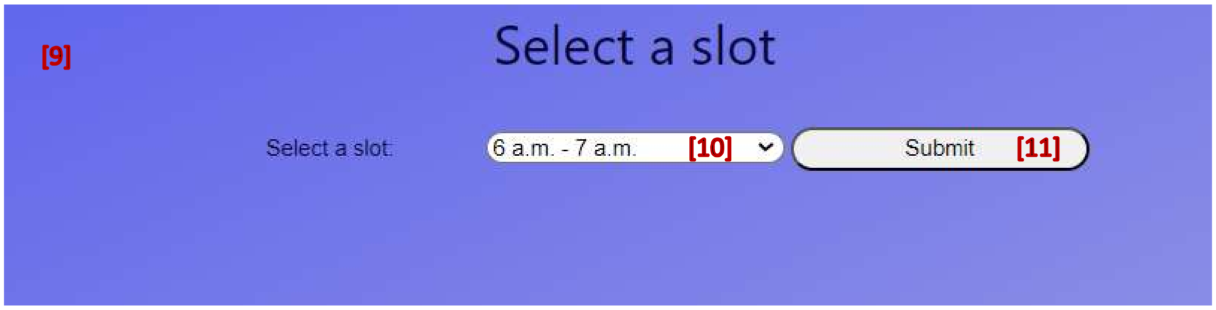
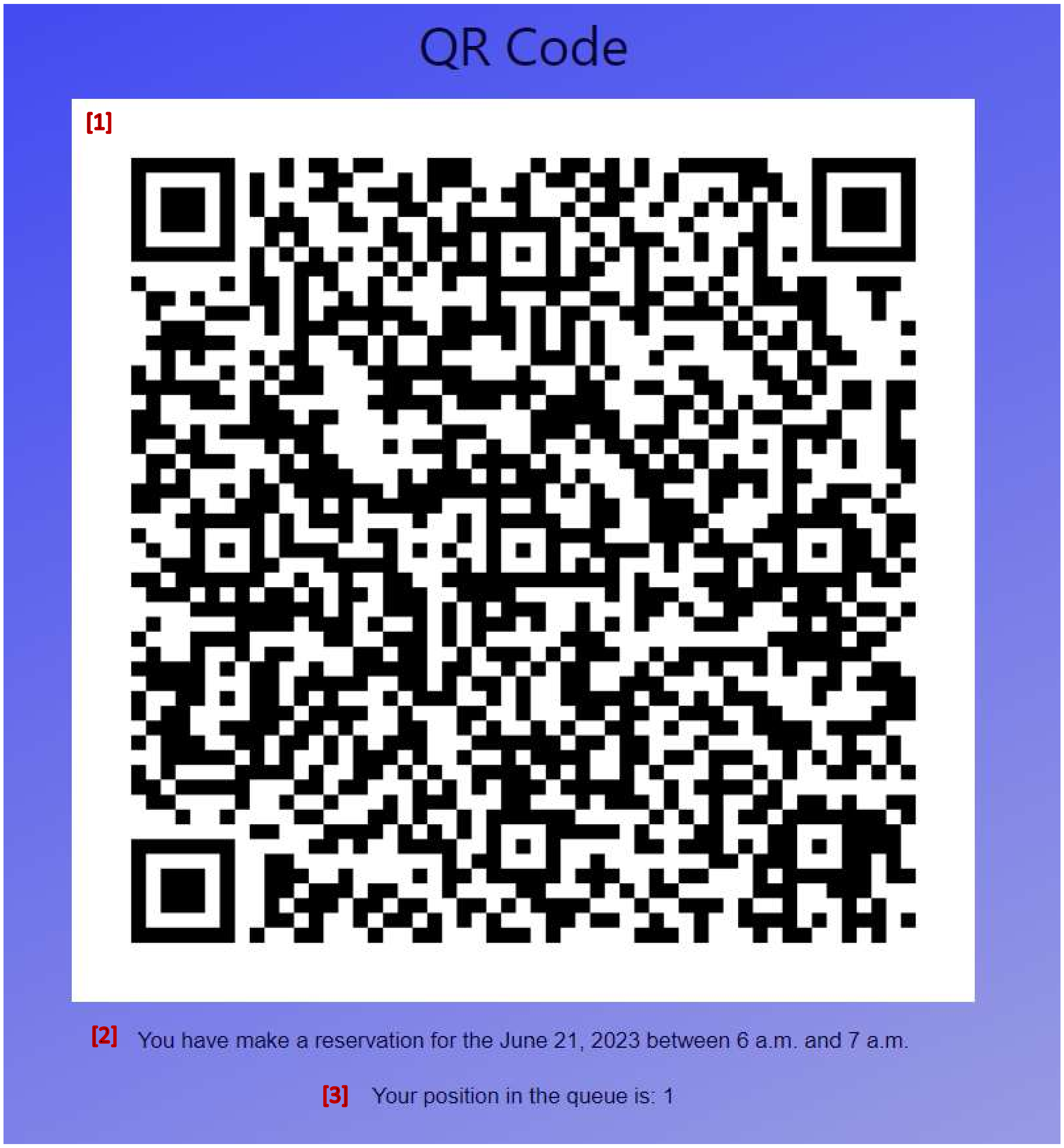


Figure 14: User guide – Booking page 2.

4.7 QR confirmation page

After booking a slot, the QR associated with his reservation is provided to the customer [1]. On this page, the customer can also see a brief recap of his reservation [2] and his position in the queue [3]. The QR is now added to the ones on the *Myreservations page*.

The customer now that the reservation is done can go back to the *Customer main page* or the *login page* by pressing their respective buttons.



You have make a reservation for the June 21, 2023 between 6 a.m. and 7 a.m.

Your position in the queue is: 1

Figure 15: User guide – Reservation QR page.

4.8 Myreservations page

On this page, a customer can control the reservations he has made. In particular, it is reported the QR code [1], the shop name [2] its address [3], when the reservation is [4] and the position in the queue [5].

From this page, the customer can choose to be guided to the shop by pressing the button [6]: doing this, the customer is redirected to Maps where the shop addresses it is already inserted automatically. Of course, the customer needs a device with a GPS to enjoy this feature.

The customer can also delete his reservation by pressing the button [7]. No fees are applied to the customer if this operation is done.

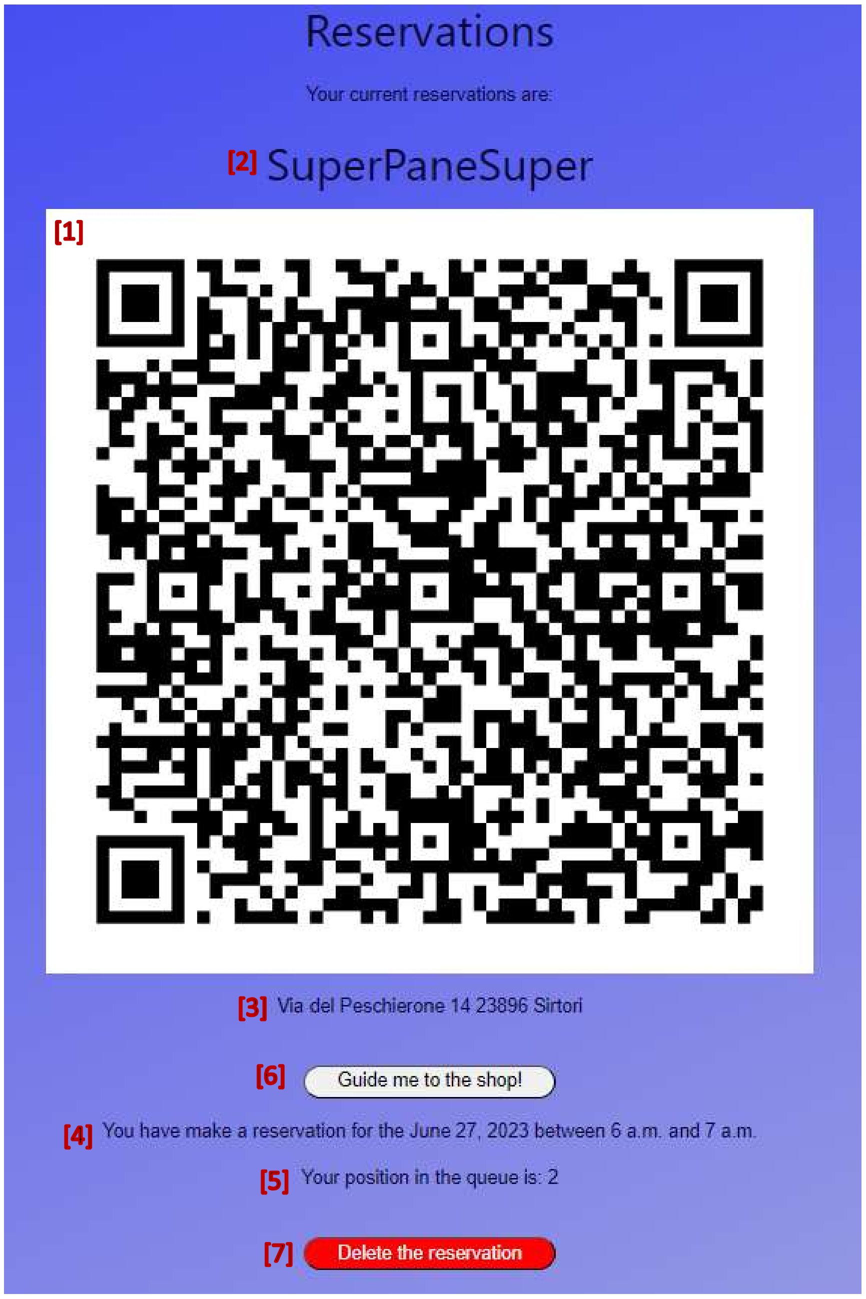

Reservations
Your current reservations are:
[2] SuperPaneSuper
[1]
[3] Via del Peschierone 14 23896 Sirtori
[6] Guide me to the shop!
[4] You have make a reservation for the June 27, 2023 between 6 a.m. and 7 a.m.
[5] Your position in the queue is: 2
[7] Delete the reservation

Figure 16: User guide – Myreservations page.

Also from this page, the customer can go back to the *Customer main page* or the *login page* by pressing their respective buttons.

4.9 Reviews page

On this page, a customer can review a shop that he visited, hence where his reservation QR was scanned. A list of reviews that the customer could compile is presented: in it, for each shop visited is reported his name [1], a grade that is selected by a selection bar [2] by the customer and a button send [3] to dispatch the review.

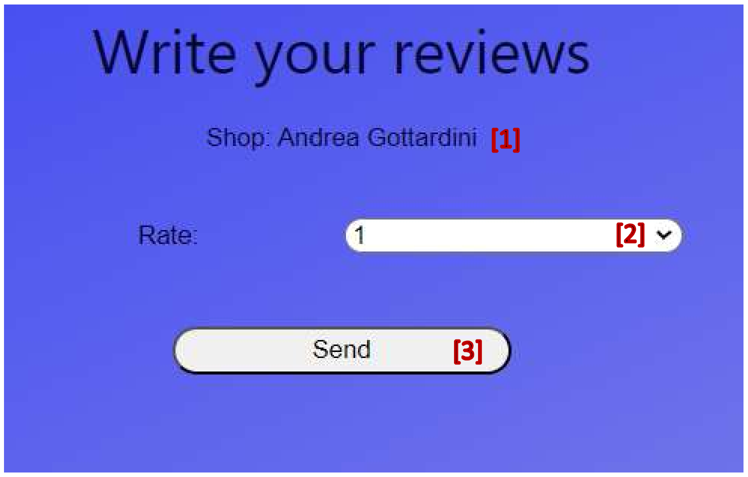
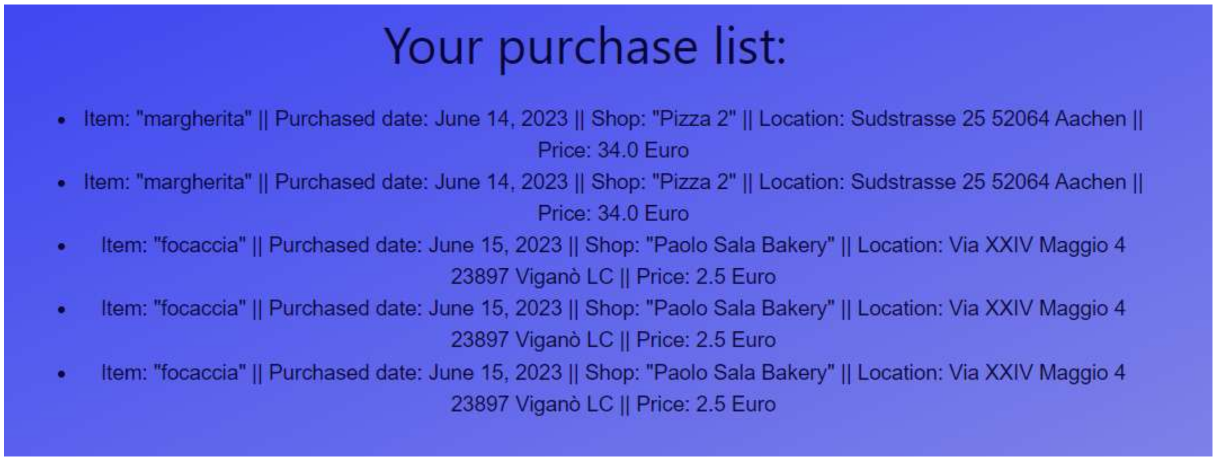


Figure 17: User guide – Reviews page.

4.10 Purchase list page

The customer can oversee and control the products/service that he bought in the Iqueue registered shop. The information of the date of purchase, their shop origin and price is reported.

Also from this page, the customer can go back to the *Customer main page* or the *login page* by pressing their respective buttons.



Your purchase list:

- Item: "margherita" || Purchased date: June 14, 2023 || Shop: "Pizza 2" || Location: Sudstrasse 25 52064 Aachen || Price: 34.0 Euro
- Item: "margherita" || Purchased date: June 14, 2023 || Shop: "Pizza 2" || Location: Sudstrasse 25 52064 Aachen || Price: 34.0 Euro
- Item: "focaccia" || Purchased date: June 15, 2023 || Shop: "Paolo Sala Bakery" || Location: Via XXIV Maggio 4 23897 Viganò LC || Price: 2.5 Euro
- Item: "focaccia" || Purchased date: June 15, 2023 || Shop: "Paolo Sala Bakery" || Location: Via XXIV Maggio 4 23897 Viganò LC || Price: 2.5 Euro
- Item: "focaccia" || Purchased date: June 15, 2023 || Shop: "Paolo Sala Bakery" || Location: Via XXIV Maggio 4 23897 Viganò LC || Price: 2.5 Euro

Figure 18: User guide – Purchase list page.

An item remains memorized in the purchase even after its origin shop is canceled from the Iqueue application.

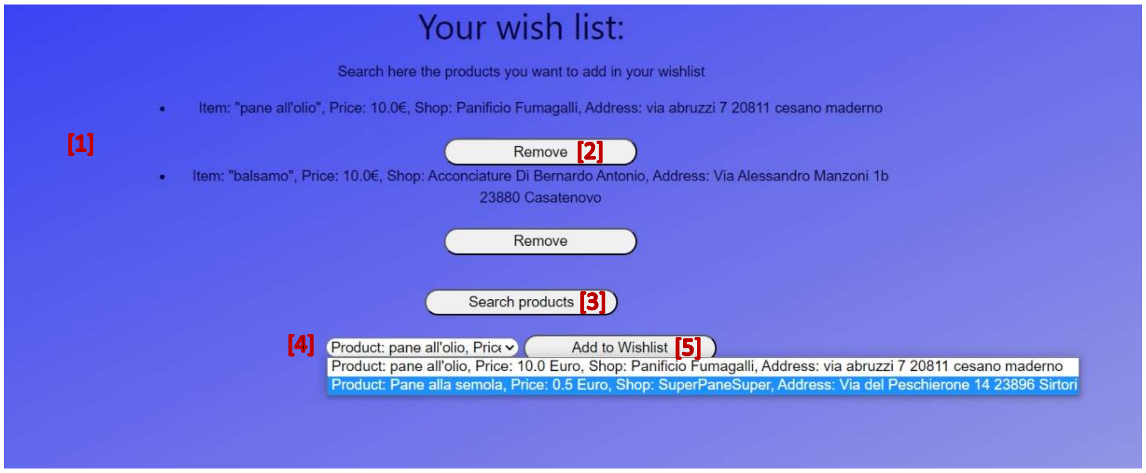
4.11 Wish list page

From this page the a customer can add products to his wishlist. The selected products are added in the section [1]. The customer can remove items from the wish list by pressing the button [2].

A product can be added to the wish list by pressing the button search products [3]: doing this, a search bar appears. Writing here the product name that we want to research and pressing the button search that is in the same of the button [5] Iqueue provides a list of products [4] belonging to the registered shops that matches its name (the research is not sensitive on the uppercase or lowercase). The customer can now add the prduct to the wishlist by pressing the add to wishlist button [5].

If a shop is canceled, also its products are removed from the wishlist.

Also from this page, the customer can go back to the *Customer main page* or the *login page* by pressing their respective buttons.



- Item: "pane all'olio", Price: 10.0€, Shop: Panificio Fumagalli, Address: via abruzzi 7 20811 cesano maderno
- Item: "balsamo", Price: 10.0€, Shop: Acconciature Di Bernardo Antonio, Address: Via Alessandro Manzoni 1b 23880 Casatenovo

Product: pane all'olio, Price: 10.0 Euro, Shop: Panificio Fumagalli, Address: via abruzzi 7 20811 cesano maderno
Product: Pane alla semola, Price: 0.5 Euro, Shop: SuperPaneSuper, Address: Via del Peschierone 14 23896 Sirtori

Figure 19: User guide – Wish list page.

4.12 Shop owner main page

On this page, a shop owner can take his actions that are the following:

- Myshops [1]: by pressing this button, the shop owner is redirected to the *Myshops page*. On this page, the shop owner can control and manage his shops.
- Product [2]: by pressing this button, the shop owner is redirected to the *Product page*. On this page, the shop owner can control and manage his products.
- Advertisement [3]: by pressing this button, the shop owner is redirected to the *Advertisement page*, where he can advertise his shop.
- Scan QR [4]: by pressing this button, the customer is redirected to the *Scan QR Reservation page*, where he can see the products and services he bought in the shops.

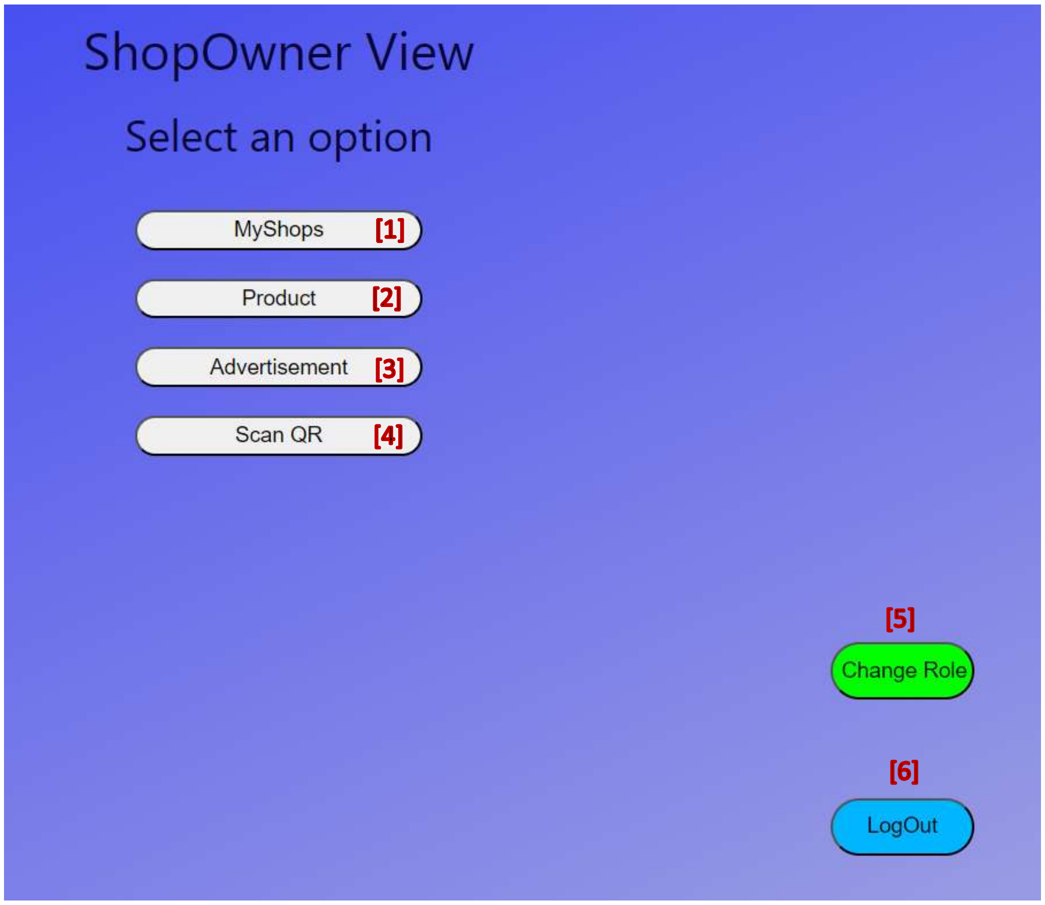


Figure 20: User guide – Shop Owner Main page.

From this page, it is possible to go back back to the *Select Role Page* by pressing the button [5] or to the *login page* by pressing the button [6].

4.13 Myshops page

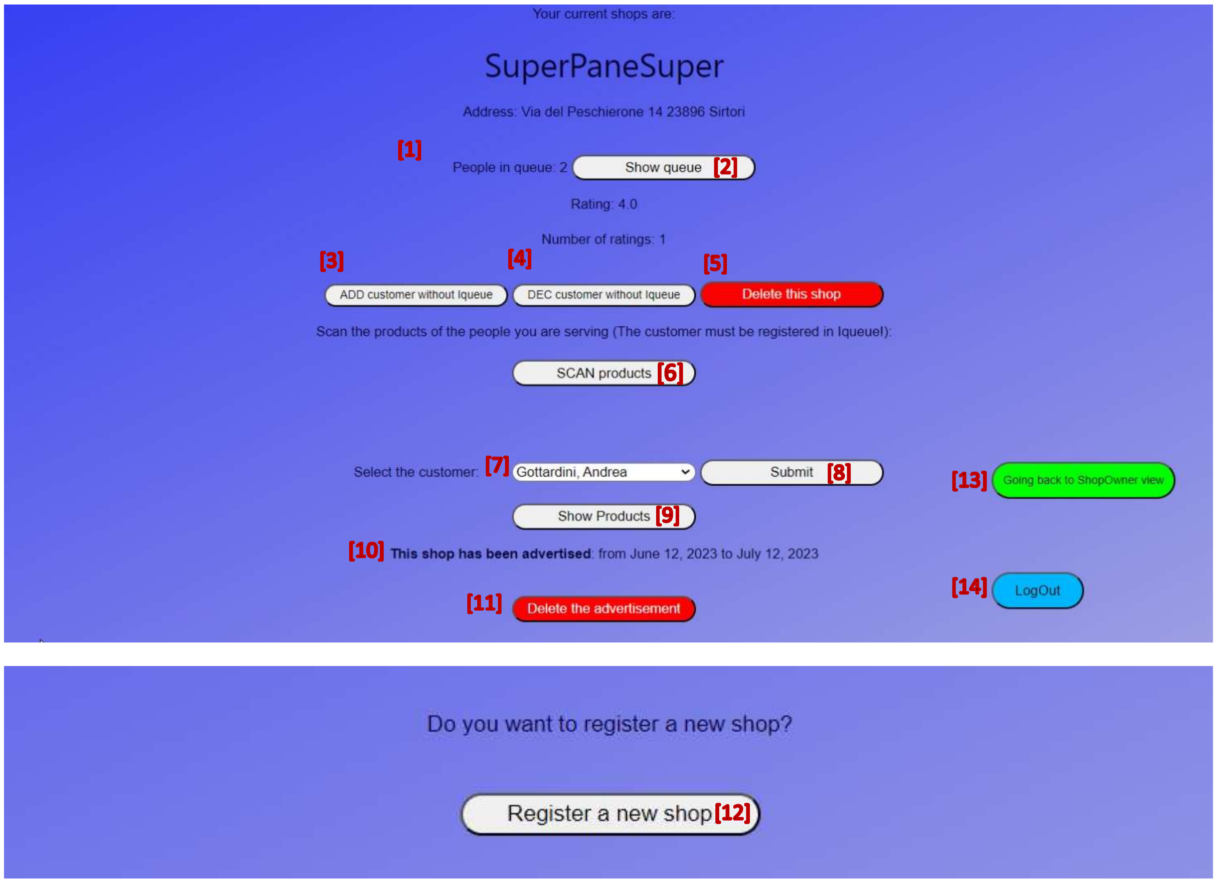
On this page, the shop owner can oversee his shop's situations and manage them. In particular, the page reports the information associated with the shop as its name, address, actual people in the queue, rating e number of ratings that it has received in the section [1].

In addition, if the shop has been advertised, in the section[10] it appears the date when the advertisement period of the shop will end. The purpose of the ADD customer without Iqueue and DEC customer without Iqueue is to permit the shop owner to include in the queue and so keep track of the customer without the application. It is important to denote that customers without Iqueue application will be placed at the bottom of the queue list in their arriving order.

The shop owner can take the following actions on this page by pressing the following buttons:

- Show queue [2]: redirect to a page where the people in the queue with their name and customer ID. For further details, please read the next Section *Shop queue page*.
- ADD customer without Iqueue [3]: increment by 1 the number of people in the queue. It must be used when a customer without the application arrives at the shop.
- DEC customer without Iqueue [4]: decrement by 1 the number of people in the queue. It must be used when a customer without the application leaves the shop.

- Delete this shop [5]: after asking the confirmation from the shop owner, the shop will be deleted. When a shop is deleted from Iqueue, are also deleted the reservations to it, its products and its possible advertisement. The shop products in the wishlists of the customers are removed, whereas they remain in the customers' purchase lists.
- SCAN products [6]: this button appears on the page whenever is present in the queue an Iqueue customer. If pressed, the selection bar [8] pops up: from this bar, the shop owner can select a customer in the queue for which the product will be scanned by pressing the submit button [8].
- Submit [8]: it redirects to the *Scan products page*. For further details, refer to the relative section.
- Show products [9]: redirect to the *product list page*, where the shop owner can see the products that he has registered to a certain shop.
- Delete the advertisement [11]: deletes the advertisement associated with the shop.


Your current shops are:

SuperPaneSuper

Address: Via del Peschierone 14 23896 Sirtori

[1] People in queue: 2 Show queue [2]

Rating: 4.0

Number of ratings: 1

[3] ADD customer without Iqueue [4] DEC customer without Iqueue [5] Delete this shop

Scan the products of the people you are serving (The customer must be registered in Iqueue!):

SCAN products [6]

Select the customer: [7] Gottardini, Andrea Submit [8] [13] Going back to ShopOwner view

Show Products [9]

[10] **This shop has been advertised**: from June 12, 2023 to July 12, 2023

[11] Delete the advertisement [14] LogOut

Do you want to register a new shop?

Register a new shop [12]

Figure 21: User guide – Myshops page.

The shop owner can also register a new shop by pressing the button [12]. In this case, the shop owner is redirected to the *shop registration page*.

From this page, it is possible to go back back to the *Shop Owner main Page* by pressing the button [13] or to the *login page* by pressing the button [14]. For simplicity, this buttons will not be reported in the next screenshots even if they are present.

4.14 Shop queue page

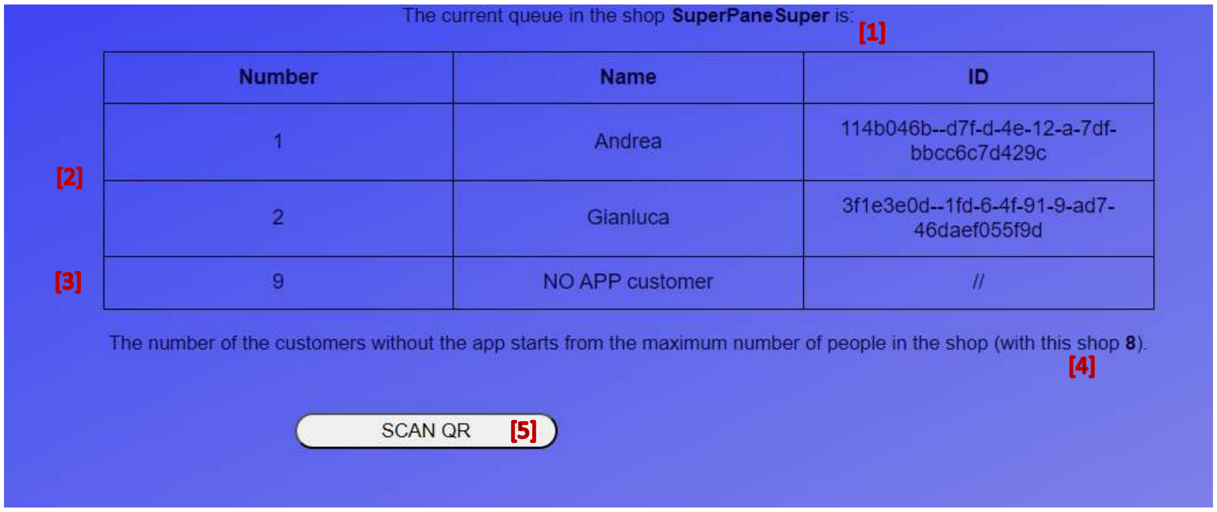
On this page, it can be seen the name of the shop [1] for which we are listing the queue. The queue list is organized into two sections: the one where are reported the Iqueue customer [2] and the one associated with normal customers [3]. The number in the queue associated with the customers without Iqueue starts from the next number of the maximum number of people for the shop [4].

It is suggested that the shop owner calls the customer based on the order reported by calling their name. In the case of two people with the same name, the shop owner shall ask for their ID to identify their position in the queue. Only after the shop owner has called all the Iqueue customers in the queue, it shall call the ones without the app.

It is suggested to the shop owner to install a digital monitor where the queue list is shown to faster operations and propose an improved experience to the customers.

A customer to asses his reservation shall show his QR reservation of the shop to the shop owner.

By pressing the button SCAN QR [5], the shop owner is redirected to the *Scan QR Reservation page*.


The current queue in the shop **SuperPaneSuper** is: [1]

Number	Name	ID
1	Andrea	114b046b--d7f-d-4e-12-a-7df-bbcc6c7d429c
2	Gianluca	3f1e3e0d--1fd-6-4f-91-9-ad7-46daef055f9d
9	NO APP customer	//

The number of the customers without the app starts from the maximum number of people in the shop (with this shop **8**). [4]

Figure 22: User guide – Shop queue page.

Also from this page, it is possible to go back back to the *Shop Owner main Page* or the *login page* by pressing the respective buttons.

4.15 Scan products page

On this page, the shop owner can scan the product QRs for the customer that he has selected previously on the *Myshops page*. After a product is scanned, the shop owner is redirected to the *Myshops page*.

It is important to denote that from this scanner, only product QRs can be analyzed not the one of the reservations. After a product QR is scanned, the quantity of that product is diminished by one and that product is added to the customer purchase list.

Attention! To use your camera, you need to allow Iqueue to do so when your browser window pop-ups.

Attention! This operation must be done before scanning the reservation QR.

A shop owner can print its product QRs by going to the *print product QRs page* which will be explained next.

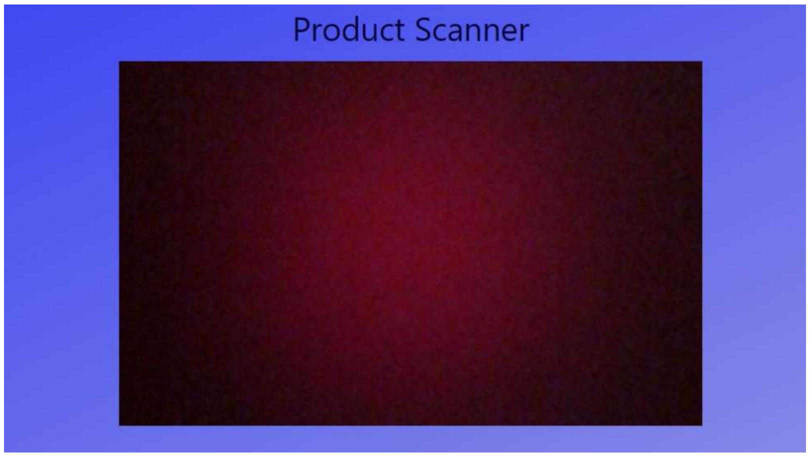


Figure 23: User guide – SCAN products page.

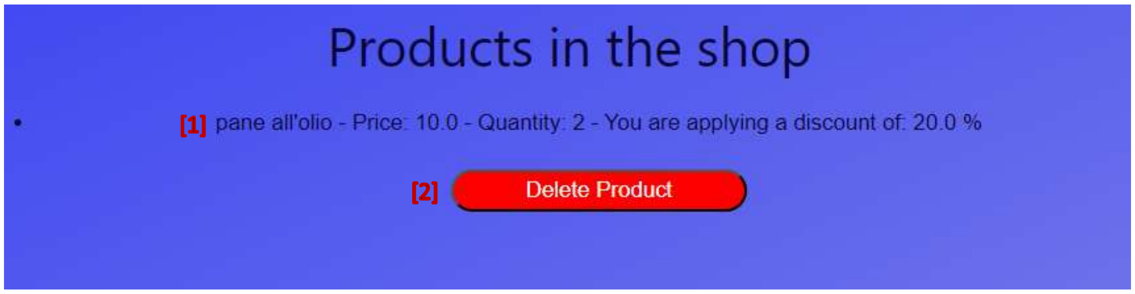
4.16 Product list page

On this page, the shop owner can see the list of products that he has registered with the shop. Additionally, it is reported their actual price, their quantity and the actual discount in the section [1]. As said previously, when a product is scanned in the *SCAN products page*, its quantity will diminish.

A shop owner can also delete a product in his shop by pressing the delete product button [2].

Attention! No confirmation is required in this case!

When a product is deleted, it is also its QR and it is removed from the customer's wishlist.



Products in the shop

- [1] pane all'olio - Price: 10.0 - Quantity: 2 - You are applying a discount of: 20.0 %

Figure 24: User guide – product list page.

Also from this page, it is possible to go back back to the *Shop Owner main Page* or the *login page* by pressing the respective buttons.

4.17 Shop registration page

Here the shop owner can insert the information associated with the shop that he wants to register. If the shop does not belong to one of the reported categories, the shop owner shall select the "other" option.

After defining the opening and closing time, the shop owner has to define the slot duration, thus the amount of time where he can manage a number of people equal to the one that he has specified in the maximum number of clients field. As reported, the slot duration has to be given in minutes.

The shop owner could also choose to insert an image in his shop [1]. If he does not do so, a default image is associated with the shop.

After pressing the record button [2], the form is submitted and the shop owner has to wait until Iqueue finishes the registration of the shop in its servers.

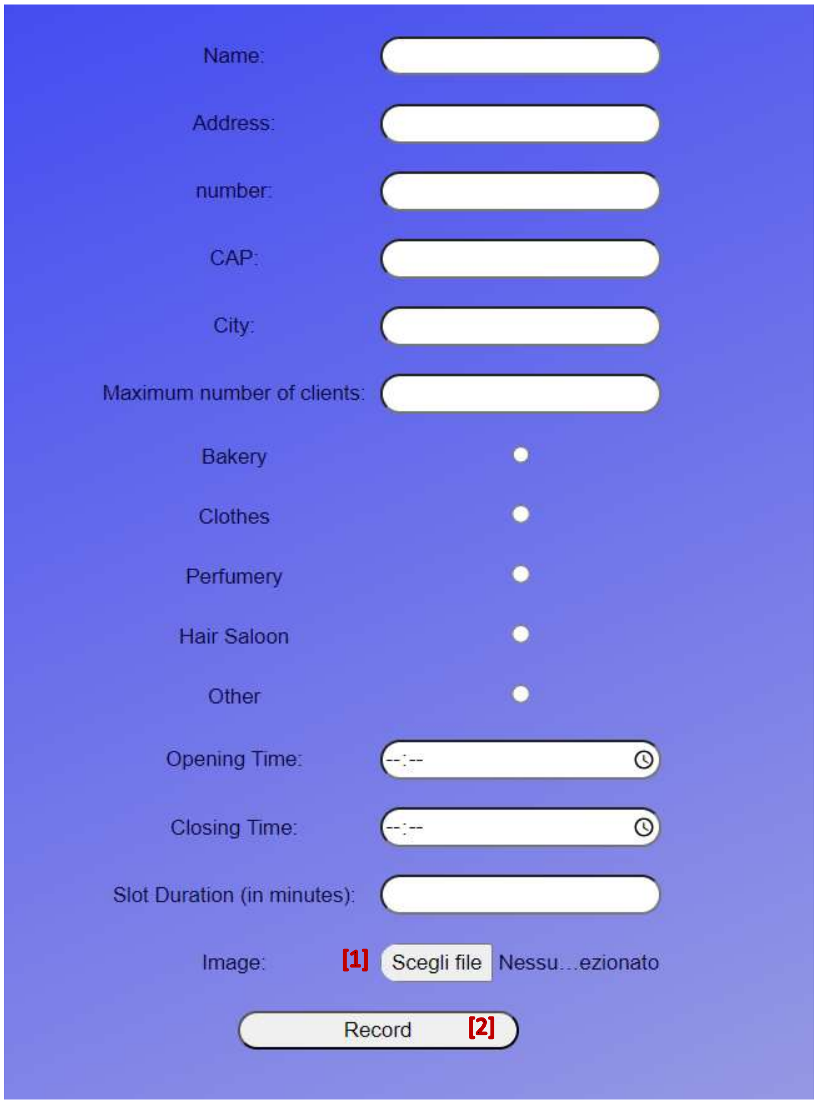


Figure 25: User guide – Shop registration page.

From this page, it is possible only to go to the *login page* by pressing its button, since the operation of the shop registration is considered important.

4.18 Product page

On this page the shop owner can choose the shop to which add products in the selection bar [1]. Then, he need to compile the below form [2] and then register the product pressing the confirm button [3].

This page also allow the possiblity to print the QRs associated with the products by pressing the button Print_QR [4]. Doing so, the shop owner is redirected to the *product QR page* explained next.

If the shop owner desires to modify the information of his products, then he needs to press the Modify product button [5], that will redirect the shop owner to the *Edit product page*.

We remember that a product quantity is diminished every time its QR is scanned by the shop owner.

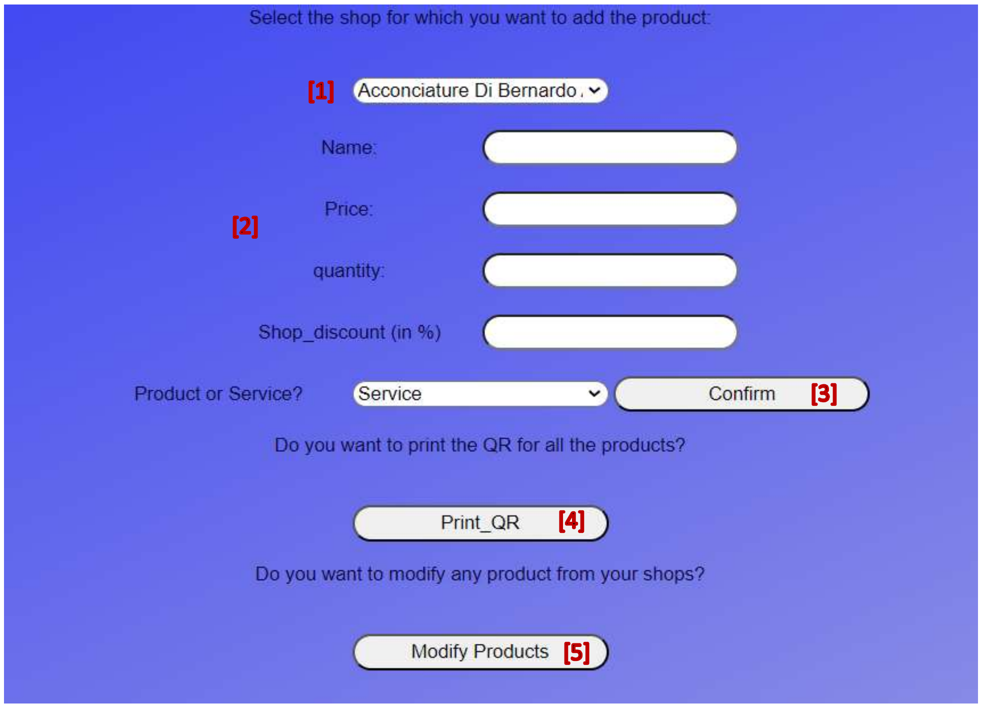

Select the shop for which you want to add the product:
[1]
Acconciature Di Bernardo
Name:
Price:
[2]
quantity:
Shop_discount (in %)
Product or Service?
Service
Confirm
[3]
Do you want to print the QR for all the products?
Print_QR
[4]
Do you want to modify any product from your shops?
Modify Products
[5]

Figure 26: User guide – Product page.

From this page, it is possible to go back back to the *Shop Owner main page* or the *login page* by pressing the respective buttons.

4.19 Product QR page

On this page, the shop owner can find out the QRs associated with his products. At the top of each QR [2] it is reported the product name [1] and at the bottom the belonging shop [3].

After printing the product QRs, the shop owner shall go back to the *shop owner main page*.

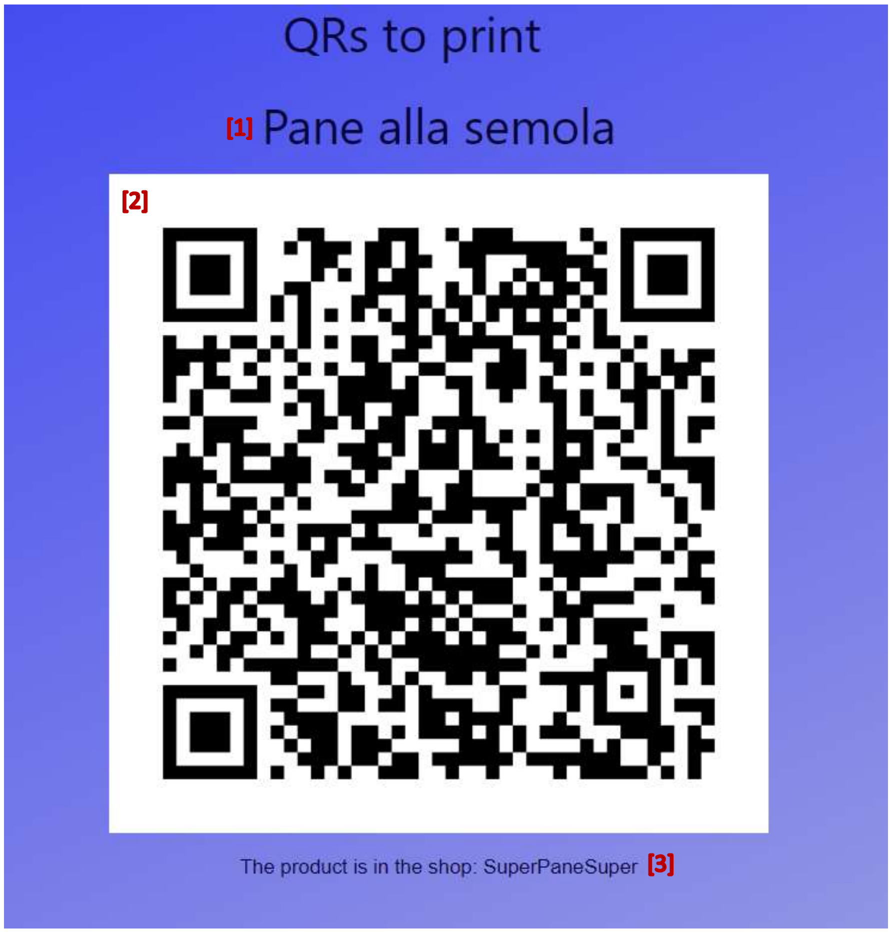


Figure 27: User guide – Product QR page.

4.20 Edit product page

The shop owner has to select from the selection bar [1] the shop in which he wants to modify product features. After doing so, he needs to press the submit button [2]. Then, he needs to select the product to edit among the ones of the selected shop from the selection bar [3] and press submit [4].

In the end, the shop owner has to insert the new product information in the form [5] and confirm his choices by pressing the button [6].

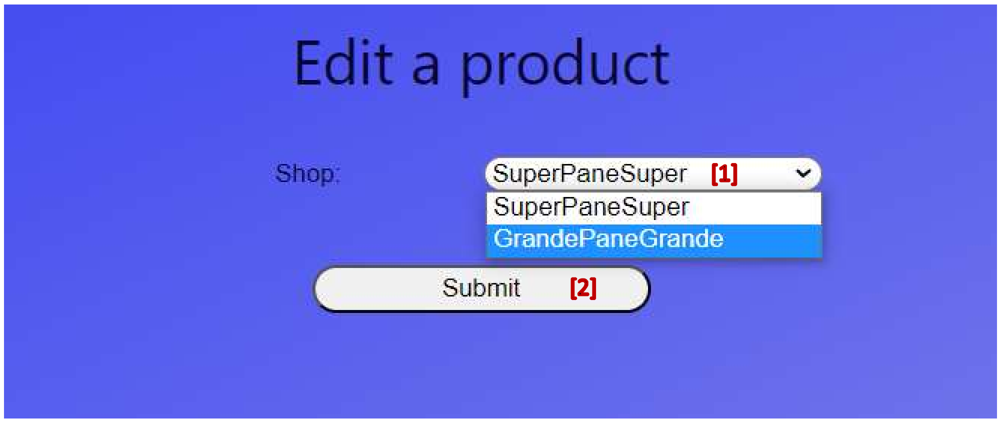


Figure 28: User guide – Edit product page - 1.

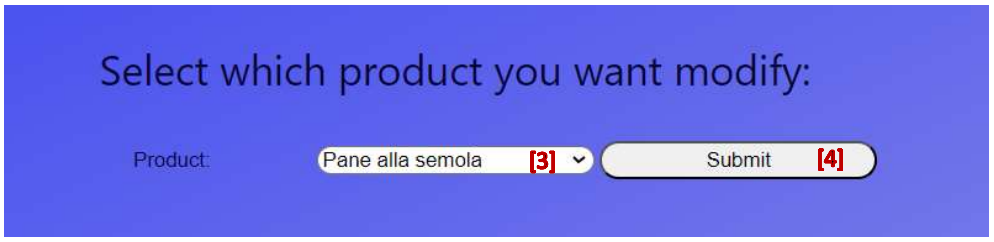


Figure 29: User guide – Edit product page - 2.

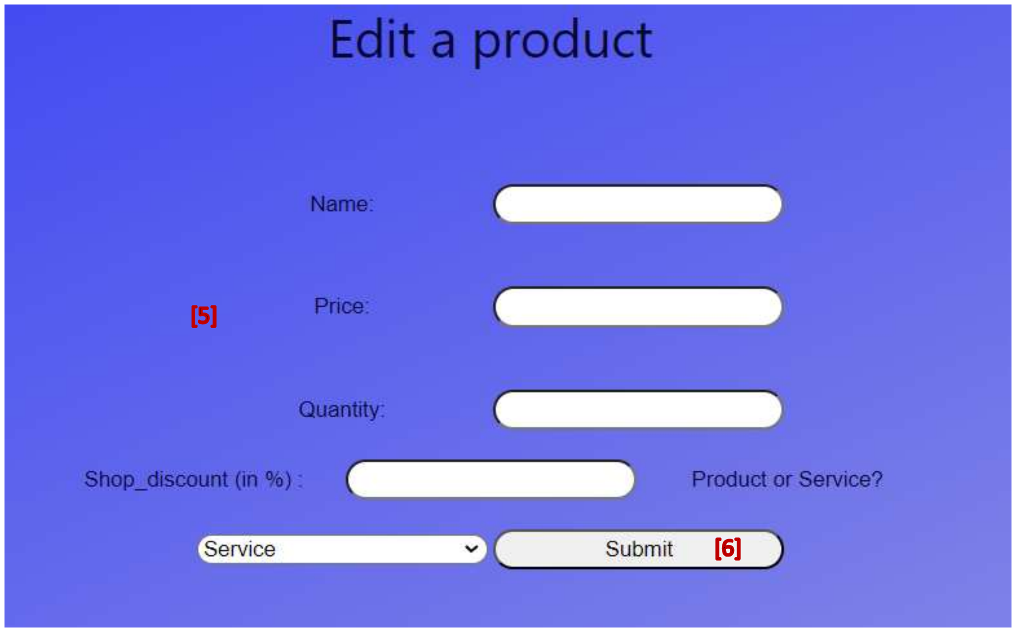


Figure 30: User guide – Edit product page - 3.

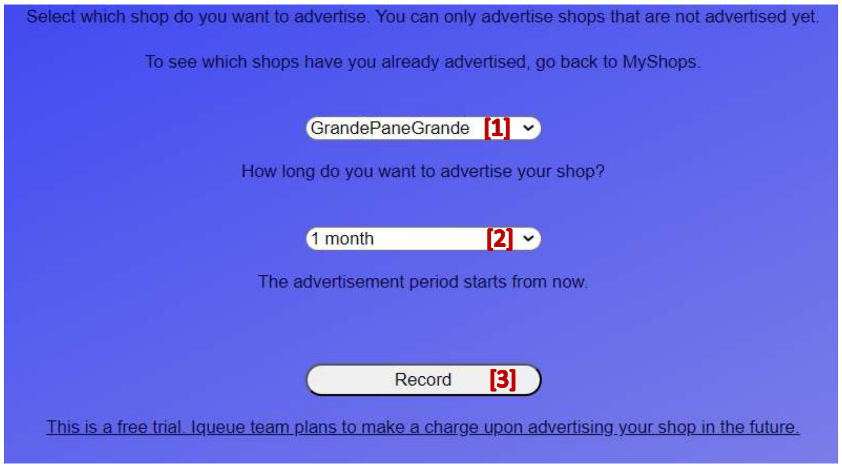
From this page, it is possible to go back back to the *Shop Owner main page* or the *login page* by pressing the respective buttons.

4.21 Advertisement page

Thanks to this page, the shop owner can advertise his shops. First, the shop owner shall select the shop that he wants to advertise from the selection bar [1]. Then, he shall select the advertisement period (1, 3, 6 months). In the end, he needs to press the record button [3] to confirm his choices. The shop image and information will now be reported to the Iqueue home page (*Select role page*).

Note that the selectable shops are the ones that have not been advertised yet.

Attention! Free advertising is a characteristic that is used to promote this Iqueue possibility. In the future, advertising a shop will require a fee.


Select which shop do you want to advertise. You can only advertise shops that are not advertised yet.

To see which shops have you already advertised, go back to MyShops.

GrandePaneGrande [1]

How long do you want to advertise your shop?

1 month [2]

The advertisement period starts from now.

Record [3]

This is a free trial. Iqueue team plans to make a charge upon advertising your shop in the future.

Figure 31: User guide – Advertisement page.

4.22 Scan QR Reservation page

From this page the shop owner can scan the QR of the customers. When a QR is scanned, the reservation of the customer is cancelled from his Myreservations page and the customer will recive a review form of the shop in his *reviews page*. After the customer has sent this form, the average review mark and number of reviews of a shop is updated.

Attention! To use your camera, you need to allow Iqueue to do so when your browser window pop-ups.

Attention! This operation is the last one that must be executed: the product scan cannot be done after it.

Figure 32: User guide – SCAN reservation page.