Immagine che contiene logo

Descrizione generata automaticamente

Iqueue Project

Implementation Document

Software Engineering for Automation (2022-2023)

Immagine che contiene testo

Descrizione generata automaticamente

Summary

[1. Frameworks 2](#_Toc137638476)

[2. Code structure 3](#_Toc137638477)

[2.1 Settings.py 3](#_Toc137638478)

[2.2 Views.py 4](#_Toc137638479)

[2.2.1 Select Role 4](#_Toc137638480)

[2.2.2 Registration View 5](#_Toc137638481)

[2.2.3 Login view 6](#_Toc137638482)

[2.2.4 Booking view 6](#_Toc137638483)

[2.2.5 Reservation view 8](#_Toc137638484)

[2.2.6 Delete QR 8](#_Toc137638485)

[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: 9](#_Toc137638486)

[2.3 models.py 19](#_Toc137638487)

[1. Requirements Implemented in the application 21](#_Toc137638488)

[2. User Manual 21](#_Toc137638489)

PER MAGGIORE CHIAREZZA, RIASCOLTARE PARTE REGISTRAZIONE ROSSI DOVE SPIEGA QUESTO DOCUMENTO

# 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.

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.

Forms.py

Manca



Non so se vuoi tenere l’immagine, secondo me è carina

# Code structure

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

## Settings.py

The settings.py file in Django contains various configurations and settings for your Django project. Let's go through the detailed description of each section in the given 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.

1. 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.

1. 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.

1. 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.

1. 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.

1. 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**.

1. Password Validation:

* **“AUTH\_PASSWORD\_VALIDATORS”:** This setting lists the password validation backends to be used for validating user passwords.

1. 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.

1. Static Files Directories:

* **“STATICFILES\_DIRS”:** This setting lists the directories from which Django collects static files.

## 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.

1. Additionally, a **PurchaseList** and a **WishList** associated with the new account are created and saved to the database.
2. Finally, the user is redirected to the 'success' page.
3. 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.
4. 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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 zer0 and renders appropriate templates based on the scanning and purchase result.

### 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.

### 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.

### 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.

## 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.

## 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.

## 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.

## 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).

# Requirements Implemented in the application

TESTO

Spiegare dove i requisiti sono stati implementati. Giustificare come mai sono stati implementati certi requisiti e altri no. Mostrare come codice metcha architectural diagram.

Vedere se scrivere for the sake of simplicity, we considered that a shop is opened from Monday to Friday with the same hours and no break.

# User Manual

TESTO