ROOMIT

Maintenance manual

Eylon Sade
Ofir Povimonsky
Noa Magrisso

Content

Introduction

ROOMIT!

A server-Client website written in Python, HTML, and CSS.

Let's get to know the structure of the server and client sides, what we have done so far to make it easier to expand - and example step by step to add new features.

The Server is written in Python using the Django web framework.
The main responsibilities of the server are to manage all the data of the users, issue the matches, and make it accessible to the Client side.

Installation

Local

To set up the ROOMIT project, follow these steps:

1. Clone the repository to your local machine.

   `git clone https://github.com/ofirpovi/RoomitFinalProject.git`

2. Navigate to the project directory.

   `cd RoomitFinalProject`

3. (Optional) Set up a virtual environment for the project.

   `python -m venv env`

4. (Optional) Activate the virtual environment.

   `source env/bin/activate`

5. Install the required Python packages using pip.

   `pip install -r requirements.txt`

   OR

   Alternatively, you can use the provided install_packages.bat file to automatically install the required packages.

   - Double-click the install_packages.bat file and it will download and install all the necessary packages for the ROOMIT project.

Remote

Our project was deployed on Pythonanywhere. For the installation part and to create your own Pythonanywhere account follow this tutorial.

After making all the configuration changes, scroll to the top of the web app configuration page.

- Click on the "Reload" button to restart the web app with the updated settings.
- Access Your ROOMIT Project
- Once the web app is restarted, you will see the URL for accessing your ROOMIT project.
- Click on the URL to open your project in a new tab or window.
- You should now be able to interact with ROOMIT on the PythonAnywhere server.

Server

Server Structure

Main App

The main app of the server is the **roomit_project** directory. There are 3 parts to this app:

- settings.py

managing all server settings:

- ALLOWED_HOSTS
- INSTALLED_APPS
- WSGI_APPLICATION
- DATABASES
- LOGGING
- AUTH_PASSWORD_VALIDATORS

- urls.py

redirect the base URL to the url.py file of each sub-app.

- wsgi.py

defining the Web Server Gateway Interface for the server.

Sub Apps

The sub-apps of the server are the **users, roomit_app** directories.
There are 9 parts to each app:

- models.py

Defining all the models for the app. A model is basically a database table.

- views.py

Defining all the views for the app. A view interacts with a model and template to complete a response. The view receives HTTP requests and sends HTTP responses.

- urls.py

redirect the URL path to the relevant view function.

- admin.py

define if the admin can manage the app from the admin site.

- migrations

Include all the migrations that were applied.
* After any changes on DB, you should run the makemigrtions and migrate commands (You can find the syntax in the commands section).

- tests

This directory contains all the tests for that specific app.

- forms.py

The forms.py file is used to define forms for your application. Forms in Django are used to handle user input and validate data.

roomit_app

- Filter.py

The filter.py file is used to define custom filters for your templates. Filters allow you to modify or format data within your templates.

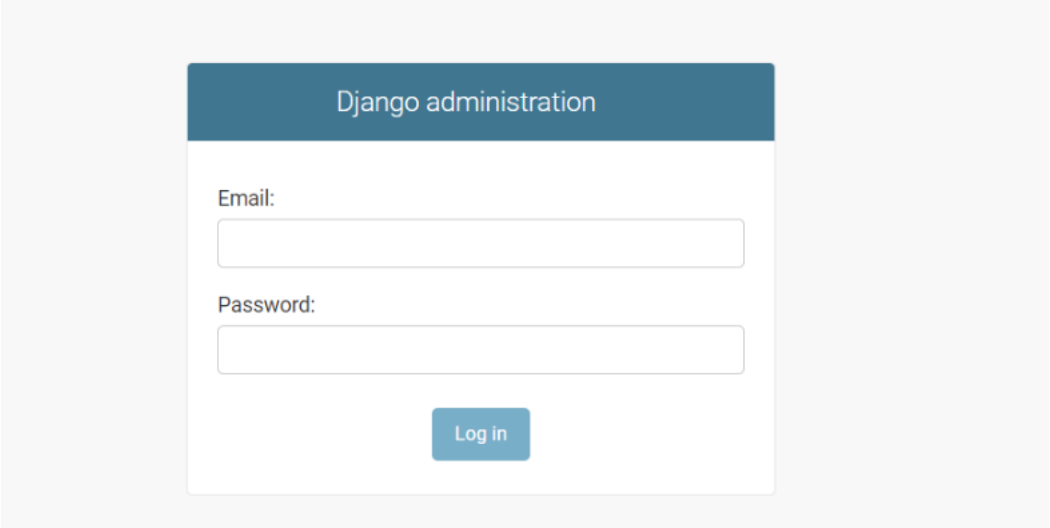
users

- signals.py

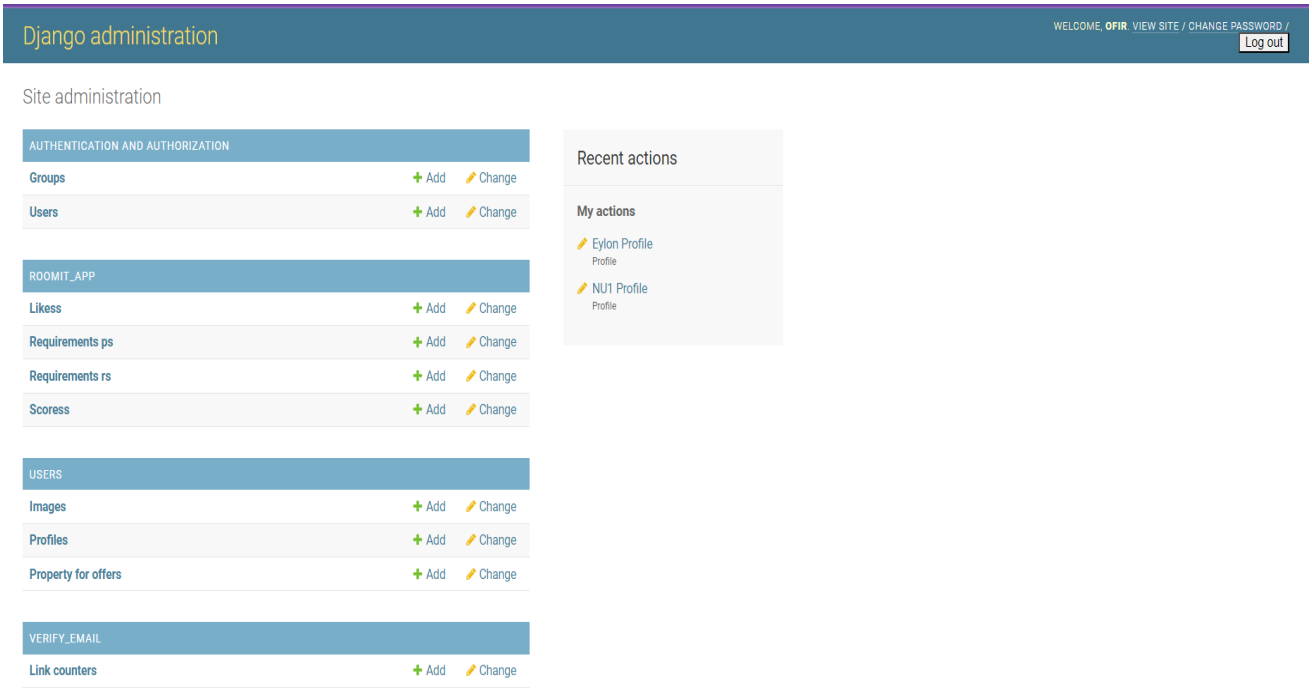
The signals.py file is used to define signals and their associated handlers. Signals provide a way to decouple various parts of your application by allowing certain senders to notify a set of receivers when an action or event occurs. By defining signals and their handlers in signals.py, you can define custom actions or logic to be executed when specific events occur, such as user registration, data updates, or other system events.

Admin user

- add admin: run in the terminal - python manage.py createsuperuser
- enter the email and password in the terminal for user admin.
- enter to the admin login page (url - $SERVER_URL/admin), and login as admin.



- from the admin page you can manage all the objects in the server.



Django Commends

- create a new app:

`python manage.py startapp <app_name>`

- creating new migrations based on the changes you have made to your models:

`python manage.py makemigrations <app_name>`

- applying and unapplying migrations update the:

`python manage.py migrate`

- run the server:

`python manage.py runserver (0.0.0.0:8000 optional)`

- creating a super user:

`python manage.py createsuperuser`

- run all the app tests:

```
python manage.py test <app_name>
```

New features:

- Adding New Sub App - Example

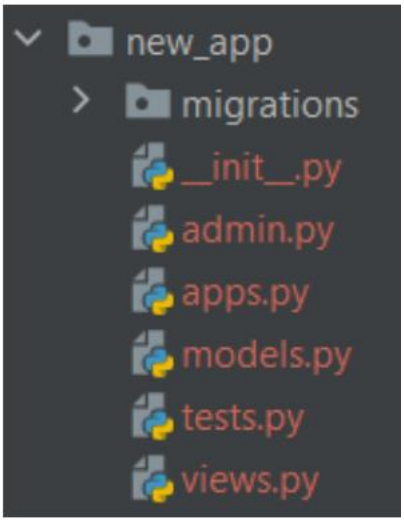
- enter the main server directory in the terminal:

```
cd $BASE_FOLDER\Final-Project\Server
```

- create sub new app:

```
python manage.py startapp new_app
```

- You should see the new app directory:



- add app name to server_django/settings.py under list INSTALLED_APPS.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'roomit_app.apps.RoomitAppConfig',
    'users.apps.UsersConfig',
    'rest_framework',
    'crispy_forms',
    'crispy_bootstrap5',
    'verify_email.apps.VerifyEmailConfig',
    'phonenumber_field',
    'infscroll',
    'djmoney',
    'django.contrib.postgres',
    'django_filters',
    'django.test',
    # 'djongo',
    'django_extensions',
]
```

- Adding a new model or modifying the your_app/models.py file:

- Create your model/ modify an existing model.
- Run makemigrations for the update file:

`python manage.py makemigrations <app_name>`

- update the DB:

`python manage.py migrate`

- Adding a new view to views.py:

1. Locate the view.py file.
2. Understand the Existing Code
   - Take a moment to familiarize yourself with the existing code in view.py.
   - Understand how the views are defined and how they interact with the models and templates.
3. Define the New View Function
   - In the view.py file, define a new function for your view.
   - Start by adding the function definition using the 'Python def' keyword.
   - Give the function a meaningful name that describes the purpose of the view.

- The function should have two parameters: request and any additional parameters required for your view's functionality.

4. Define the New View Function
   - Within the newly defined function, implement the logic for your view.
   - This may include retrieving data from the database, performing calculations, or any other necessary operations.
   - You can also import any required models, forms, or other dependencies at the top of the file.
5. Return a Response
   - After executing the necessary logic, determine what response should be returned to the user.
   - This can be an HTML template rendered with the data, a JSON response, or any other appropriate response based on your view's requirements.
   - Use Django's HttpResponse or render functions to construct and return the response.

6. Return a Response
   - To make the new view accessible via a URL, you need to map it to a URL pattern.
   - Open the urls.py file in your Django app's directory.
   - Add an entry to the urlpatterns list that maps the desired URL pattern to your new view function.
   - Choose a URL pattern that reflects the purpose of your view and follow Django's URL pattern syntax.

```
urlpatterns = [
    path('register/', users_views.register, name='register'),
    path('login/', auth_views.LoginView.as_view(template_name= 'users/login.html'), name='login'),
    path('logout/', auth_views.LogoutView.as_view(template_name= 'users/logout.html'), name='logout'),
    path('fill_info/<str:username>/', users_views.info, name='fill_info'),
    path('profile/<str:username>/', users_views.profile, name='profile'),
    path('password-reset/', auth_views.PasswordResetView.as_view(template_name= 'users/password_reset.html'), name='password_reset'),
    path('password-reset/done/', auth_views.PasswordResetDoneView.as_view(template_name= 'users/password_reset_done.html'), name='password_reset_done'),
    path('password-reset-confirm/<uidb64>/<token>/', auth_views.PasswordResetConfirmView.as_view(template_name= 'users/password_reset_confirm.html'), name='pa
    path('password-reset-complete/', auth_views.PasswordResetCompleteView.as_view(template_name= 'users/password_reset_complete.html'), name='password_reset_c
    path('set-status/', users_views.set_status, name='set-status'),
    path('change-status/', users_views.change_status, name='change-status'),
    path('property-offer-create/<str:username>/', users_views.create_property_offer_view, name='property-offer-create'),
    path('property-offer-display/<str:username>/', users_views.display_property_offer, name='property-offer-display'),
    path('property-reqs-display/<str:username>/', users_views.display_property_reqs, name='property-reqs-display'),
    path('roomi-reqs-display/<str:username>/', users_views.display_roomi_reqs, name='roomi-reqs-display'),
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

7. Test the New View
   - Save the changes to view.py and urls.py.
   - Start your Django development server.
   - Open a web browser and navigate to the URL associated with your new view.
   - Verify that the view functions as expected and returns the desired response.

Client Structure

- templates directory

Located in each app. This directory contains the HTML templates that are used to render the user interface of your ROOMIT application. Templates present the data to the users and provide a structured layout. Each sub-app within the server may have its own templates directory to organize the templates specific to that app.