



## SOCIAL NETWORK APP

### Meeting the requirements

The application complies with the specified requirements:

- A. **Users can create accounts:** The views.py file contains a function called **register** which is called when the user makes a request to the endpoint <http://127.0.0.1:8000/register/>.

Figure 1 shows the Registration Page:

SA SuperAwesomeSocialNetwork Register Log In

### Registration Page

First name:

Last name:

Username:  Required. 150 characters or fewer. Letters, digits and @/./+/\_ only.

Email address:

Password:

Do you already have an account? [Log in](#)

© 2022 by Student Number: 190126209 😊

Figure 1. Registration Page

For the registration process the username, email and password fields are required. Once the account is created the user is automatically logged in and redirected to their profile page where they can include more information (profile picture, bio, etc.). A default image is added to their profile:

SA

SuperAwesomeSocialNetwork

Search


UserA

## Profile page for UserA

First name:

Last name:

Current profile picture:



UserImg:

Currently: [default\\_profile\\_img.png](#)

Change:

No file chosen

Bio:

Cancel

Save

© 2022 by Student Number: 190126209 😊

Figure 2. Profile Page

SA

SuperAwesomeSocialNetwork

Search

UserA


## Profile page for UserA

Your profile was updated

First name:

Last name:

Current profile picture:



UserImg:

No file chosen

Bio:

Cancel

Save

© 2022 by Student Number: 190126209 😊

Figure 3. Updated Profile Page

- B. **Users can log in and log out:** The views.py file contains a function called **login** which is called when the user makes a request to the endpoint <http://127.0.0.1:8000/login/>. Once the user is logged in is redirected to their Home Page. Figure 2 shows the Login Page:

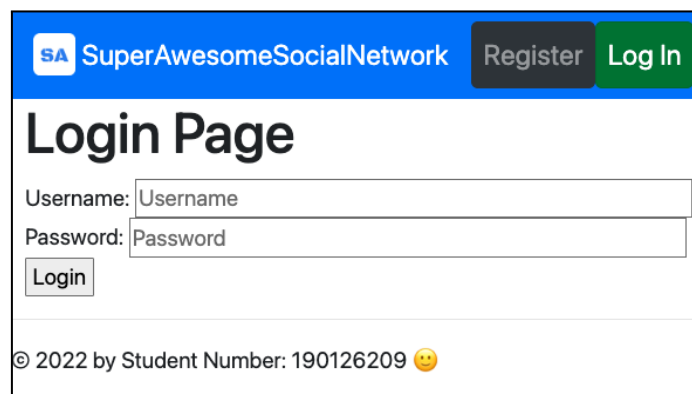


Figure 4. Login Page

Once a user is logged in, they can logout by clicking their profile picture in the top right corner, a dropdown with the Log out option will show:

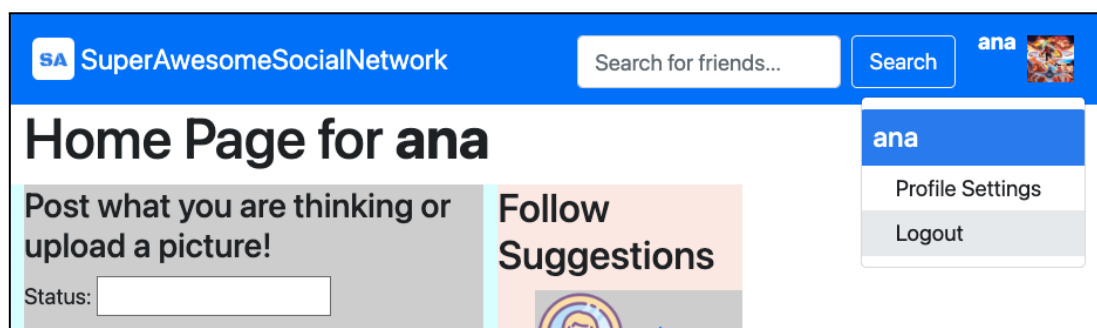


Figure 5. Logout option

There is also a button for Logout in the chat page. The user can also access their Profile Page ([Figure 2](#)) through the link shown in [Figure 5](#) just above the Logout link.

- C. **Users can search for other users:** As can be seen in [Figure 5](#), the option for searching for other users is located in the top right corner next to their profile picture. Once the user types any string and clicks the *Search* button or *Enter* a new page is shown with the results. This is a simple search where I look for the usernames

containing the string sent in the request. I had created some dummy accounts to test my application:

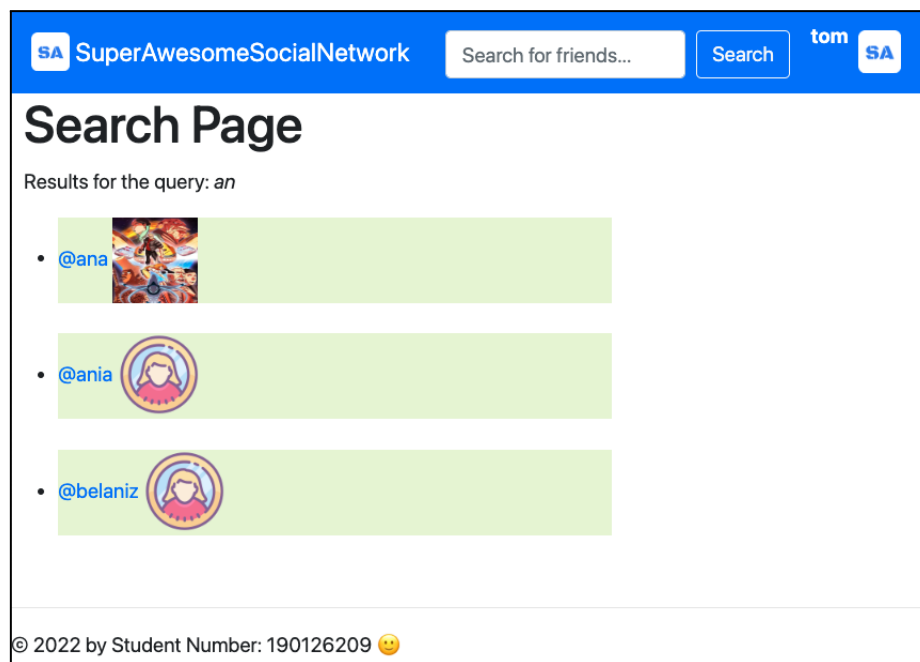


Figure 6. Search for other users (matches found)

Once the results are shown, if the user clicks the *@username* shown, they are redirected to the profile page of that user, where they can choose to follow, unfollow or message them.

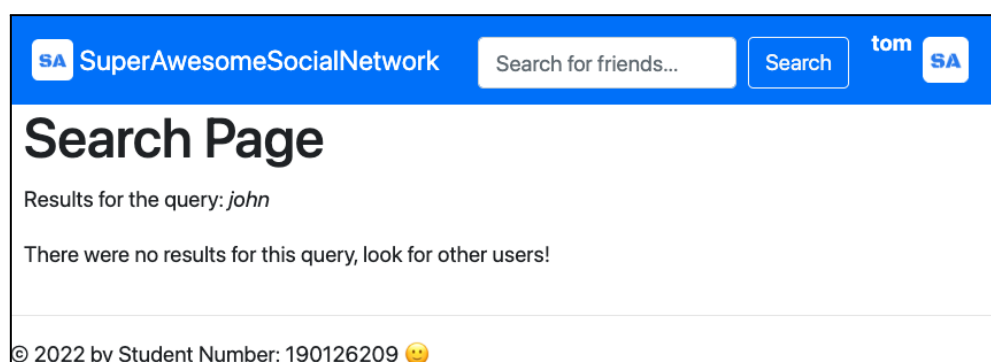


Figure 7. Search for other users (matches not found)

**D. Users can add other users as friends:** Once logged in, a user is redirected to their Homepage where they can see on the right side a list of six people that are registered in the system but the user is not yet following. User A can follow user B without user B

following user A, there is no process to accept an invitation to become friends in this application. Once user A adds user B as a friend, then user A can see the posts made by user B shown in user A's homepage, without user B knowing anything about user A.

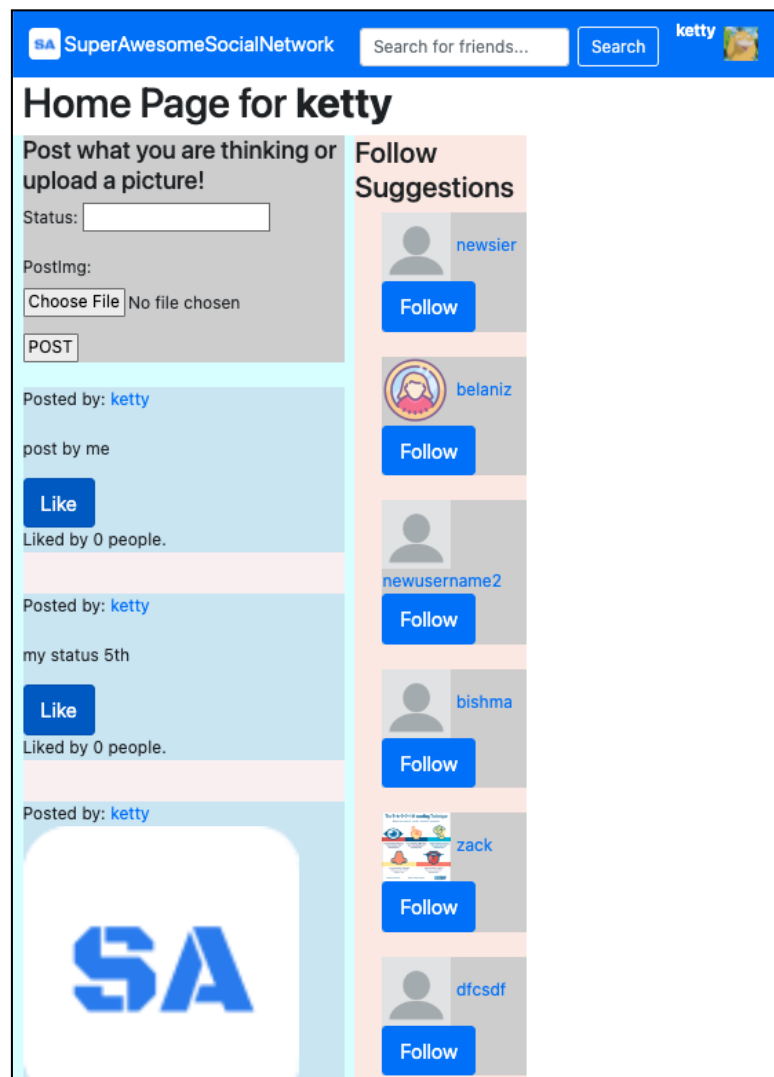


Figure 8. Follow Suggestion List

Once user A follows use B, user B will no longer appear in the list of suggested friends for user A.

- E. **Users can chat in real time with friends:** This functionality is provided by the *message* function in *views.py*, *consumers.py*, *asgi.py* and the use of a Redis server, which must be running along with the application server. Once user A befriends user B, user A can see user B's Profile Page, there is a button on the left side (next to the Follow/Unfollow button) that allows user A to start a messaging conversation with user B (this button is only shown if user A is seeing a profile for a user that it is currently following). If user B also follows user A then they can maintain a real time communication though the page:

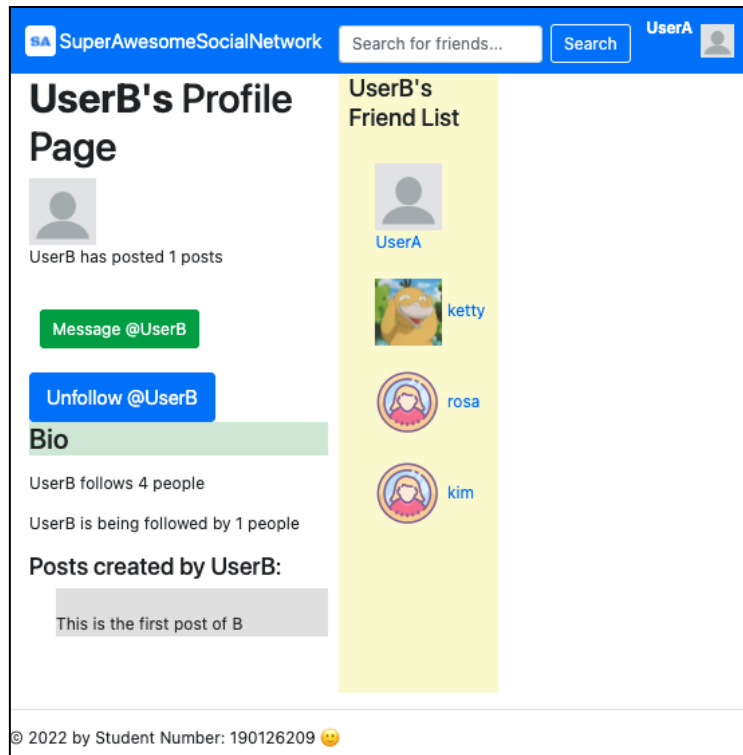


Figure 9. Message button shown in UserB's Profile Page (green button)

When the user clicks on this Message button they make a request to [http://127.0.0.1:8000/message/<str:room\\_name>/](http://127.0.0.1:8000/message/<str:room_name>/) , the room\_name parameter is made up of the usernames of both users (which are unique) and the “bigger” name comes first, making it a unique identifier. This is a private chatroom between these users, and even if userA knows the room\_name for other accounts that they do not own, they can not access these chats (see [Figure 11](#)).



Figure 10. Message Page (no messages between the users yet)

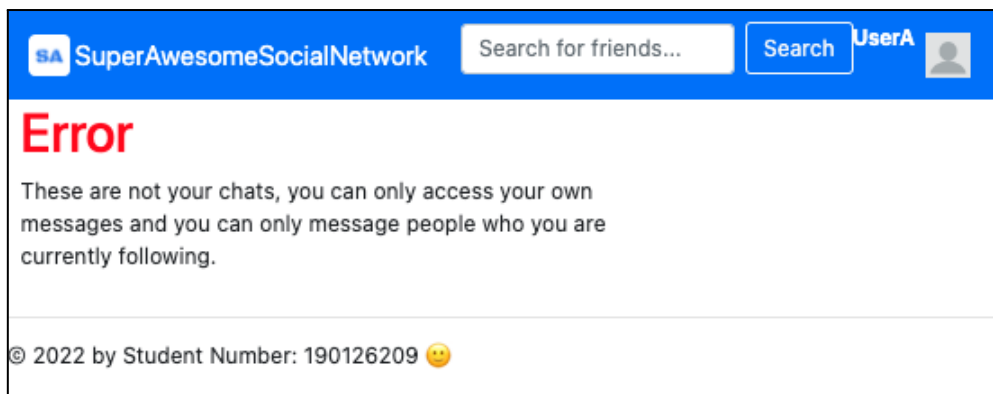


Figure 11. Message shown if the user tries to access chat rooms that does not own.

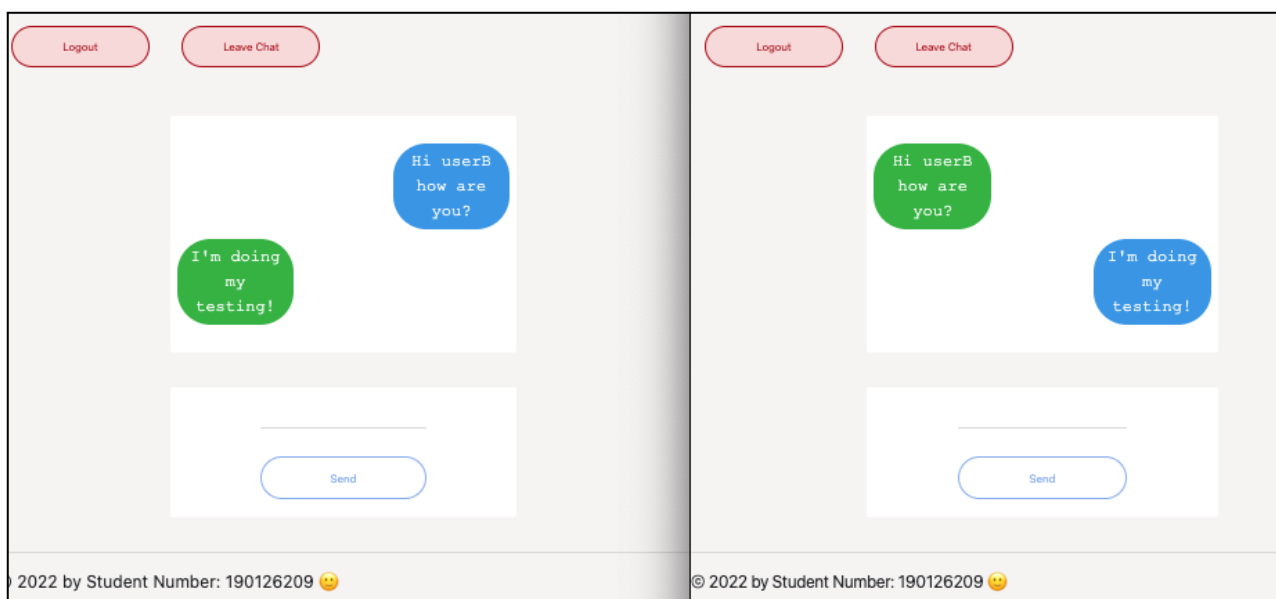


Figure 12. Real-time chat between UserA and UserB.

As can be seen in [Figure 12](#), messages written by *yourself* will show in blue on the right side of the chat, while messages written by the other user are shown in green to the left.

This functionality made use of WebSockets, as established in the requirements. See the `asgi.py` and `consumers.py` files to see the implementation.

- F. **Users can add status updates to their home page:** As shown in [Figure 8](#), the user's homepage offers in the top left side (below the navigation bar) an option to create new posts. These posts can contain text or an image (or both). Once a user creates a post, this post will be shown in their Homepage along with the posts of all the other people that the user is following. Ordering by creation date is not provided in this list. Each user can go back to their Homepage by clicking the Page Logo in the top left corner, and go to their Profile Page by clicking on their name in the top left corner next to their profile image.

- G. **Users can add media (such as images to their account and these are accessible via their home page):** Explained in the previous step.
- H. **Correct use of models and migrations:** All the models were created inside the `models.py` file and I migrated them into the default Django SQLite database (since no information was given in the requirements and we had to do this for the previous midterm).
- I. **Correct use of form, validators and serialization:** The serializers for all the models described in the `models.py` file can be found in the `serializers.py` file. All except one of the serializers inherit from `serializers.ModelSerializer` and their validations are automatically included by the Django REST framework. The `UserSerializer` does not inherit from `serializers.ModelSerializer` because I needed to remove the default validations on it. More information on why I did this can be found in the comments for this serializer.

I also had to override the update method on the `UserProfileSerializer` to allow writable nested representations in the API. I created forms inside the `forms.py` file, all of them inherit from `django.ModelForm` (since it includes validations by default). Since the app allows for a user to register, log in, create posts and update profiles I had to create a form for each of these functionalities.

- J. **Correct use of django-rest-framework:** I used this framework following the instructions given in their official [documentation](#). I created endpoints for the RESTful API in the `urls.py` file of the app. These endpoints offer the user the ability to perform CRUD operations (not all CRUD operations are available for all endpoints). All the API endpoints require the user to be authenticated. The functionality of the API is provided by the functions located in the `api.py` file.

- `api/users`: allows users to see a list of all registered users and their basic information ([Figure 13](#)). Only allows GET requests.
- `api/user/<int:pk>` : allows users to see the specific details of a user specified by the users primary key, which is their id ([Figure 14](#)). Only allows GET requests.
- `api/edit_user/<int:pk>`: allows users to see and update the details of the user (GET and PUT requests). The `pk` parameter is the id of the `UserProfile` object, not the `User` object. This page can only be accessed by the owner of the account, otherwise they get a 403 response ([Figure 16](#)). The users can update their name, last name, user profile image or their bio information (any combination of them or just one field). The only required parameter for PUT



requests is the *username*, since it must be the same as the logged in user (to prevent users from updating other people's profiles) see [Figure 15](#).

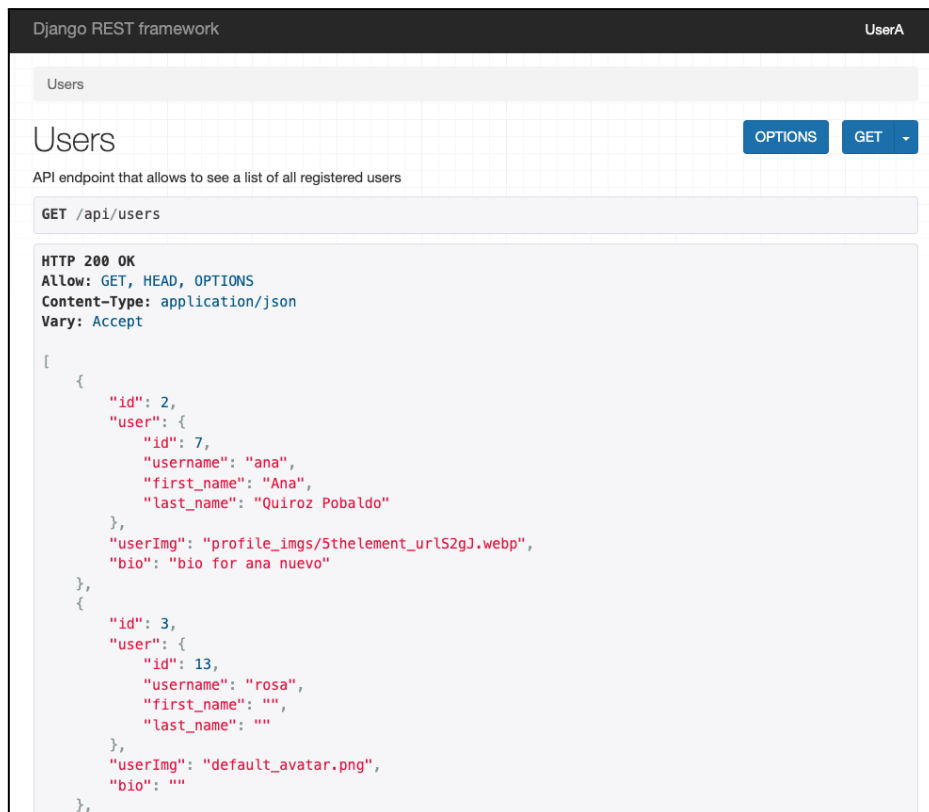


Figure 13. api/users endpoint

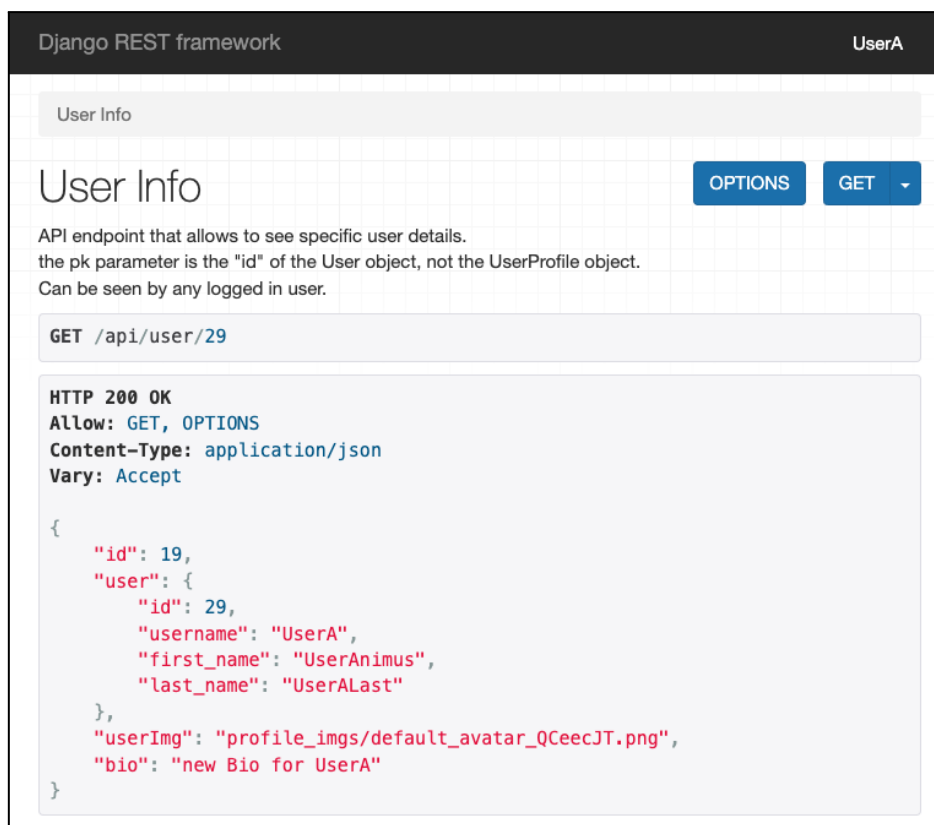


Figure 14. api/user/<int:pk> endpoint

The screenshot shows the Django REST framework interface for the 'User Detail' endpoint. The top bar indicates the user is 'UserA'. The endpoint is 'PUT /api/edit\_user/19'. The response is 'HTTP 200 OK' with 'Allow: GET, PUT, PATCH, HEAD, OPTIONS', 'Content-Type: application/json', and 'Vary: Accept'. The JSON response is:

```
{
  "id": 19,
  "user": {
    "id": 29,
    "username": "UserA",
    "first_name": "UserAnimus",
    "last_name": "UserALastname"
  },
  "userImg": "profile_imgs/Screen_Shot_2022-09-02_at_8.45.56_PM.png",
  "bio": "bio for A"
}
```

Below the response, there are tabs for 'Raw data' and 'HTML form'. The 'HTML form' tab is active, showing a form with fields for 'Username' (UserA), 'First name' (UserAnimus), 'Last name' (UserALastname), 'Userimg' (Choose File), and 'Bio' (bio for A). A 'PUT' button is at the bottom right of the form.

Figure 15. `api/edit_user/<int:pk>` endpoint (access granted)

The screenshot shows the Django REST framework interface for the 'User Detail' endpoint. The top bar indicates the user is 'UserA'. The endpoint is 'GET /api/edit\_user/5'. The response is 'HTTP 403 Forbidden' with 'Allow: GET, PUT, PATCH, HEAD, OPTIONS', 'Content-Type: application/json', and 'Vary: Accept'. The JSON response is:

```
{
  "detail": "You do not have permission to perform this action because you are not the owner of this account"
}
```

Figure 16. `api/edit_user/<int:pk>` endpoint (access denied)

- `api/posts` : allows users to see a list of all their posts and their details. Also allows them to create new posts (see [Figure 17](#)).

Django REST framework
tom

Post List

Post List

API endpoint that allows to see a list of all the posts created by the logged in user.

GET /api/posts

```

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
  {
    "id": "1ff4fb40-d28a-400c-b6d8-93d5eb82bd02",
    "postImg": null,
    "creationDate": "2022-09-03T21:26:11.379708Z",
    "status": "This is Tom's first post",
    "likeCount": 0
  },
  {
    "id": "16ce25a0-3017-4d6d-b174-ecbe0a6945a0",
    "postImg": null,
    "creationDate": "2022-09-03T21:26:17.913517Z",
    "status": "Tom's second post",
    "likeCount": 0
  },
  {
    "id": "5c1099b8-0033-4325-8503-27644fcab40b",
    "postImg": "post_imgs/example_0fsDh0n.jpg",
    "creationDate": "2022-09-03T21:26:31.121410Z",
    "status": "Tom's third post",
    "likeCount": 0
  }
]

```

Raw data
HTML form

Posting
Choose File no file selected

Creationdate
09/03/2022, 12:30 PM

Status

POST

Figure 17. api/posts endpoint

Django REST framework
ana

Post Detail

Post Detail

API endpoint that allows to see the details of a post, update those details, and delete the post completely. The logged in user should be the owner of the post to be able to do these actions.

GET /api/post/33e2ed5d-0023-43c0-b198-14fb2c39a6ba

```

HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
  "id": "33e2ed5d-0023-43c0-b198-14fb2c39a6ba",
  "postImg": "post_imgs/red.png",
  "creationDate": "2022-09-03T21:28:36.072070Z",
  "status": "this is Ana's second Post",
  "likeCount": 2
}

```

Raw data
HTML form

Posting
Choose File no file selected

Creationdate
09/03/2022, 12:30 PM

Status
this is Ana's second Post

PUT

Figure 18. api/post/<int:pk> endpoint. (access granted)

- [api/post/<int:pk>](#) : allows users to see the details of a unique post identified by *pk*. It also allows for a user to delete or update said post ([Figure 18](#)). The user is not allowed to modify the likeCount of the post or its owner. This page is only accessible to the user owner of the post, otherwise you get a 403 response ([Figure 19](#)).
- [api/friends](#) : allows users to see a list of the people they befriended and their profile information (See [Figure 20](#)).



Figure 19. `api/post/<int:pk>` endpoint. (access denied)

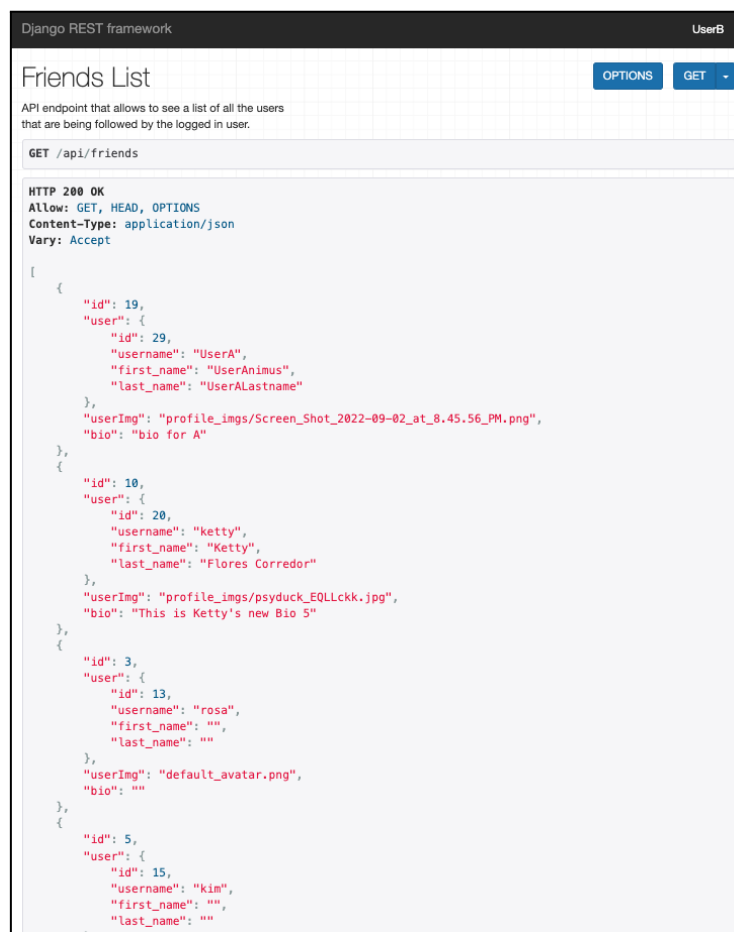


Figure 20. `api/friends` endpoint

K. **Correct use of URL routing:** All the endpoints for this API are defined in the *urls.py* file inside the *awesome\_social\_network\_app* folder, each one points to a different function defined in the *views.py* or *api.py* files.

L. **Appropriate use of unit testing:** All the unit tests are defined inside a *tests.py* file. Each test name is descriptive and each test runs in isolation from the others. I made use of the *factory\_boy* framework, which also includes *Faker*. With these frameworks I was able to create instances of every model in *models.py*. These instances are used in my unit tests so that I do not have to create each instance by hand for every test I make.

I created factories for five classes in the *models.py* file: *User*, *UserProfile*, *PostUpdate*, *PostLiked* and *Following* using *factory.django.DjangoModelFactory*. In the setup for the tests I defined a function to generate image files, to be able to test the functionalities of the API for uploading images.

The tests tried to cover many situations:

1. Users are authenticated (or not)
2. Users get correct responses from the endpoints
3. Users get errors if trying to see the profile of a non-existing user
4. Users get an error if trying to edit the profile of another user
5. Users can only access a list of their own posts
6. Users can only edit/see/delete posts that they own.
7. Users can actually update the contents of a post they own.
8. Users can POST a new post (with or without images or status)
9. Users get an error if trying to access posts that do not exist.
10. Users can see, edit, delete their posts
11. Users can only see/edit their own profile
12. Users can see a list of the people they follow.
13. ETC.

I had to update the serializers to add read only fields to prevent users from updating fields that should not be updated (e.g. edit the owner of a post, edit the number of likes in a post, etc.). This I only realized when creating the tests. I did not include tests for the chat or the “like post” functionalities. It should be noted that when the tests are run, the images that are created for testing are also saved in the app’s database and are not deleted along with the test database (The images are just blue or red squares).

**M. An appropriate method for storing and displaying media files is given:** In the settings.py file I defined two variables:

```
MEDIA_URL = '/users_media/'  
  
MEDIA_ROOT = os.path.join(BASE_DIR, 'users_media')
```

These define where the application will store the images that are uploaded by the users. I created a folder to store the post images and one to store the profile images of the users. In the *models.py* file I define the *upload\_to* attribute for the Image Fields to be one of those folders.

**Implements an appropriate database model to model accounts, the stored data and the relationships between accounts.**

There are six models in the project: User, UserProfile, PostUpdate, PostLiked, Following, Chat and ChatRoom. The User table is provided by default by Django. I included code to make the email field on this table required and unique when a user registers. The fields for each one are:

User:

- id (pk, auto)
- username (required, unique)
- first\_name
- last\_name
- email (required, unique)
- password (required)

UserProfile:

- id (pk, auto)
- user (One to One field to User model)
- userImg
- bio

PostUpdate:

- id (pk, uuid)
- user (Foreign Key to User model)
- postImg
- creationDate (defaults to current time at the moment of creation)

- status
- likeCount

PostLiked:

- id (pk, auto)
- post (Foreign Key to PostUpdate model)
- user (Foreign Key to User model)

Following:

- id (pk, auto)
- followerUser (Foreign Key to User model)
- followedUser (Foreign Key to User model)

Chat:

- id (pk, auto)
- content
- timestamp
- user (Foreign Key to User model)
- room (Foreign Key to ChatRoom model)

ChatRoom:

- id (pk, auto)
- name

The relationships between these models can be better appreciated in the Entity-Relationship diagrams shown in [Figures 21](#) and [22](#). A UserProfile belongs to just one User, this is because I just created UserProfile to be able to add extra information about a user which is not provided by the default Django User Model.

A User can create many PostUpdate objects (these are user's posts). A PostUpdate object belongs to just one User object, by a Database constraint of Foreign Key, which means that if the User is deleted its posts will also be deleted from the database since I set the parameter `on_delete` to `CASCADE`, since there should be no posts belonging to a nonexistent User.

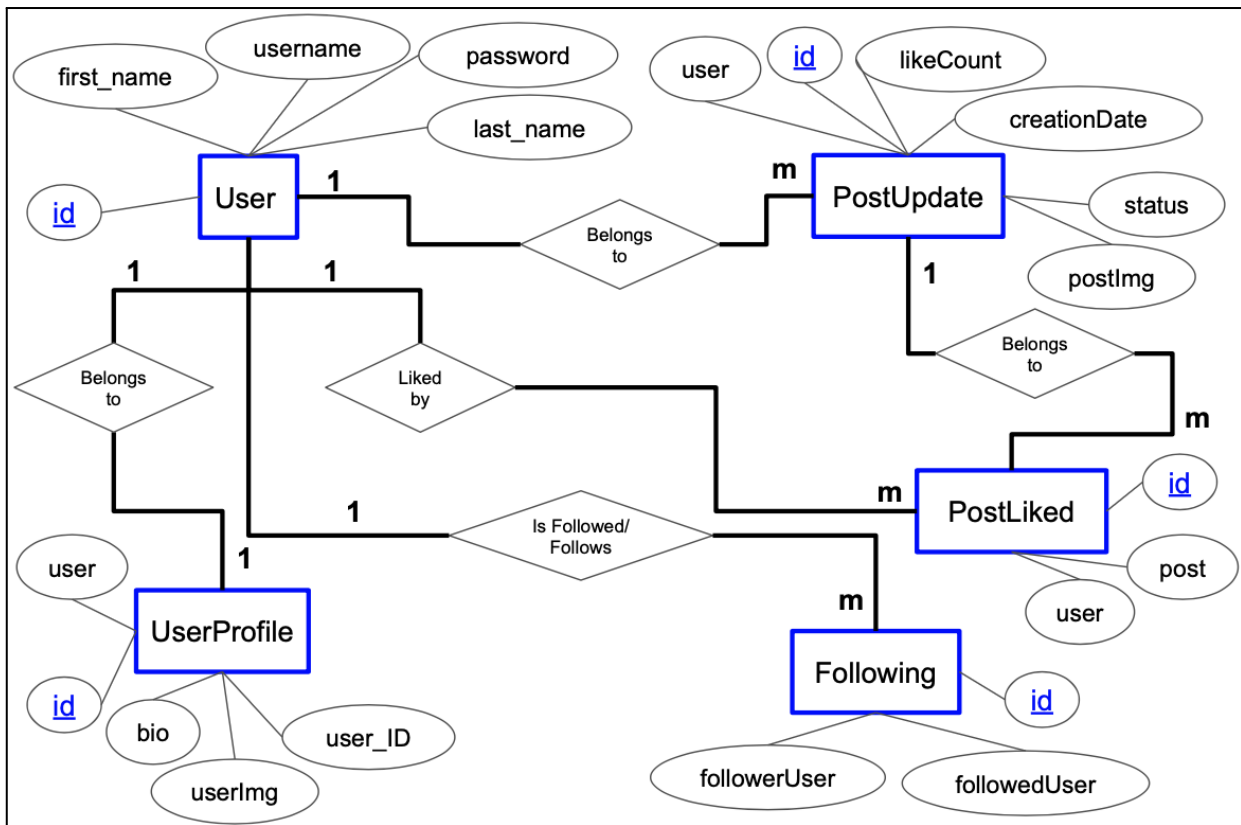


Figure 21. Entity-Relationship Diagram (Part 1)

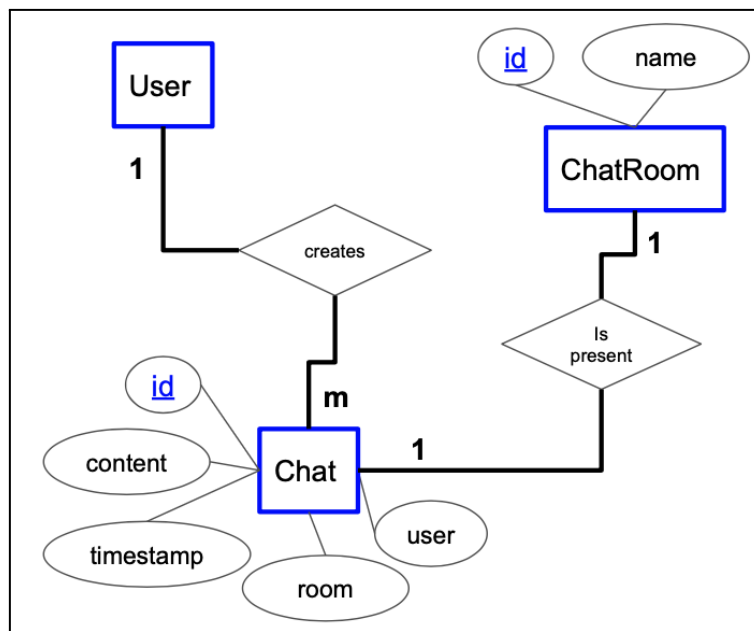


Figure 22. Entity-Relationship Diagram (Part 2)

A PostUpdate object can be associated with zero or more PostLike objects by a Foreign Key constraint. If the post is deleted, so are the many PostLike objects that were created for it. A User can create many PostLike objects, each for a different PostUpdate object (since a User can only like one post once, or zero).



A User can decide to follow another user, in which case a Following object is created. A Following object associates two User objects by using Foreign Key constraints. The same user can create many Following objects (which means to follow many people). When a user “unfollows” somebody, the associated Following object is deleted from the database.

A User can create many Chat objects (these are the messages written by a user in a chatroom). The Chat belongs to a User and to a ChatRoom by Foreign Key constraints. Chat objects belong to a certain ChatRoom object, and in order to make the ChatRoom a private business between two users I use a name field in the ChatRoom model, which will be unique for any combination of UserA and UserB. That way a user can only access ChatRooms that include their usernames in the name field.

The ChatRoom name is created by merging the usernames of the users in the chat, and placing the “bigger” one first. Example: if one username is “tom” and other “username” is “ana” then the ChatRoom name for these two users will always be “tom\_\_ana”. Taking into account that each username is unique, the name of this private ChatRoom between these two users is also unique. This is established in the custom\_functions.py file in the getChatRoomName function. I did it this way because I could not devise another way of making the chats “private”.

The decisions I took to create the models were based on what I knew I would need to implement the requirements of the project. I knew I had to create an extension of the User model (UserProfile) because I needed to be able to store more information about my users. I created a PostUpdate model because I needed a way to store the posts that my users created and I knew it should be linked with the User model because a post should not exist without the user that created it. The PostUpdate model includes an ImageField because one of the requirements was that users could upload pictures. It also includes a “status” field because the requirements established that a user should be able to post status updates. With this model I solve both issues, since a user can create a post with just text, with just an image, or both.

The PostLiked model was an extra feature, to allow me to track how many likes a PostUpdate object received. I created the Following model that way because I think it was an easy workaround for Users following Users, just to include a Table that says “userA” follows “userB”. That way if I wanted to get the list of all the people that are being followed by “userA” I just had to query this table where the followerUser was “userA”.

The Chat model was created because the requirements established that we needed to use at least one WebSocket, and for this I needed chat objects and room objects. The chats would be

all the messages that a user wrote, and that is why the Chat model has a Foreign Key constraint to reference the User model. Also a chat (or message) is written in some place, that place would be the “room” field, which is a Foreign Key reference to the ChatRoom model. That way if “userA” writes a chat to “userB”, that chat will be associated with the ChatRoom with name “userB\_\_userA”, and if I want to retrieve all the messages that these users had written to each other I just have to query the Chat table to get all the Chat objects where the room has a name of “userB\_\_userA”.

The code is properly arranged and I used object oriented design. I created two new files *custom\_functions.py* and *custom\_permissions.py* to maintain cleaner code and separate concerns for commonly used functionalities (e.g. get a list of friends or posts).

I found some bugs when creating the tests. For example, I realized that users could assign their posts to other users, create posts for other users, or change the profiles of other users while using the API to make POST and PUT requests. So it was necessary for me to include `read_only_fields` for the Serializers or override their “put” method in order to solve those bugs.

As is recommended, the static files (e.g. css and JavaScript files) are included in a static folder in the application. All my html templates are included in a template folder and they make extensive use of Django template language to avoid repeating code.

### **Implementation of appropriate code for a REST interface that allows users to access their data.**

Since there were no specific requirements on what the API should allow users to access, I decided that the API should allow each user to:

- See, edit and delete their own posts.
- See a list with all the posts they own.
- See a list of all their friends
- See a list of all the registered users
- See and edit their profile (first name, last name, profile image, bio, etc.)
- See their profile information

Please refer to section [J](#) for detailed explanation of each endpoint.

## Setting up the application

In order to run the application, please unzip the file FinalProjectKF.zip. Create a new virtual environment and use the requirements.txt provided to install all the required libraries for the project. You must run the Redis server for the application to work properly. In my machine I had to run this command once inside my virtual environment:

```
% /usr/local/opt/redis/bin/redis-server
```

But the command you need to run the Redis server might differ from this one. Next, open a terminal and *cd* into the FinalProjectKF/SA\_social\_network folder, there you should run the following command:

```
% python manage.py runserver
```

In my machine I had to write *python3* instead of *python* in the previous command, maybe it will be the same for you depending on the Python version on your computer. There are several Users already included in the SQLite database (tania, daniel, george, ana, tom, etc.), you can see them all by login into the admin page once the servers are running. Go to <http://127.0.0.1:8000/admin> and use “admin” as username and 1234 as the password (all other users have the same password).

Since the project is using SQLite as a database, in order to test the real live chat it is necessary that you log in with one user in one browser page and log in into another account on an incognito (private) window. Remember that in order to message another user, you must follow that user first (e.g. ana follows tom and vice versa, then they can message each other).

## How to run the tests

Once you have unzip the FinalProjectKF.zip file, open a terminal and *cd* into the FinalProjectKF/SA\_social\_network folder, there you should run the following command:

```
% python manage.py test
```

It should run and pass 26 tests. Some warnings might be raised, these are due to the fact that the testing uses a format for the date objects that is not exactly the same that the PostUpdate model uses ( time-zone awareness, naive datetimes).

It should be noted that all the functionalities provided by the application (and the API) require authentication. In the tests.py file I had to force\_authenticate a default user, because for some reason the *self.client.login* function did not work, I had to use *self.client.force\_authenticate*.

## Further improvements

- The application does make use of some CSS and Bootstrap classes but not to the point of having a good user interface. More work should be done in this area to allow for a responsive and appealing website.
- Tests should be added for the real time chat and the like post functionalities. For this the API should allow creating new PostLiked objects (but I did not have enough time for this).
- Add ordering by date on the list of posts shown to the user in their Home Page
- Implement better search functionality.
- Implement an "Request" process to establish a Friendship between users
- Implement notifications for messages since at this point unless a user has a chat room with a friend opened, they would not know if some new chat has come up or not.
- Add more information in the chat (e.g. time of creation of message) and ways to scroll through the messages since at this point the page will show all the messages created, and not just the last few which can be annoying if there are many chats.
- Add functionality to add comments to a post.