**MID-TERM ASSIGNMENT REPORT**

**ADVANCED WEB DEVELOPMENT FINAL COURSEWORK**

**SOCIAL NET APP**

**Contents**

[**File Structure:** 3](#_Toc97058586)

[**How to Run:** 4](#_Toc97058587)

[**Database Model:** 5](#_Toc97058588)

[**Functionalities:** 6](#_Toc97058589)

[a) Users can create accounts 6](#_Toc97058590)

[b) Users can log in and log out 10](#_Toc97058591)

[c) Users can search for other users 14](#_Toc97058592)

[d) Users can add other users as friends 16](#_Toc97058593)

[e) Users can chat in realtime with friends 23](#_Toc97058594)

[f) Users can add status updates to their home page 26](#_Toc97058595)

[g) Users can add media (such as images to their account and these are accessible via their home page 26](#_Toc97058596)

[h) correct use of models and migrations 35](#_Toc97058597)

[i) correct use of form, validators and serialisation 36](#_Toc97058598)

[j) correct use of django-rest-framework 37](#_Toc97058599)

[k) correct use of URL routing 39](#_Toc97058600)

[l) appropriate use of unit testing 40](#_Toc97058601)

[m) An appropriate method for storing and displaying media files is given 40](#_Toc97058602)

[**Summary:** 41](#_Toc97058603)

[**Appendix:** 41](#_Toc97058604)

All necessary packages are listed in the requirements.txt but the main packages to be installed which will install other related packages are listed below.

Run pip install for following libraries:

Django==3.0.3

Pillow==9.0.1

channels==3.0.4

channels-redis== 3.3.1

djangorestframework==3.13.1

\*note: Test accounts that can be used are shown in the appendix.

# **File Structure:**

Text

Description automatically generated

Figure .1 – File structure of SocialNetApp project

appenv – python virtual environment

src/friendchat – friendchat app for friend to friend chat feature

src/friends – friend app for sending, cancelling, accepting and declining friend request, view friend list, add and remove friend features

src/media and src/media\_cdn – folder used to store all user uploaded images (profile image or post image)

src/SocialNetApp – main project folder, settings.py, routing.py etc

src/static and src/static\_cdn – folder used to store bootstrap folder, project\_files, admin static files

src/templates – contains base, header, footer html files along with css and js html files. Each app may consist of their own templates folder which will contain .html files for the respective app pages.

src/userposts – userposts app for users to post status updates or images on their homepage

src/users – users app for user registration, login, logout, view profile, edit profile and search users features

src/db.sqlite3 – database for this application

src/requirements.txt – all required packages in virtual environment

# **How to Run:**

Extract the zip file and change directory to that folder in the terminal of your editor or console command. Run the following commands to activate the virtual environment, change directory to src folder and run the server where the manage.py file is.

A screenshot of a computer

Description automatically generated

Figure .1 – Commands to run virtual environment and server

The list of accounts available for testing is written under the Appendix section.

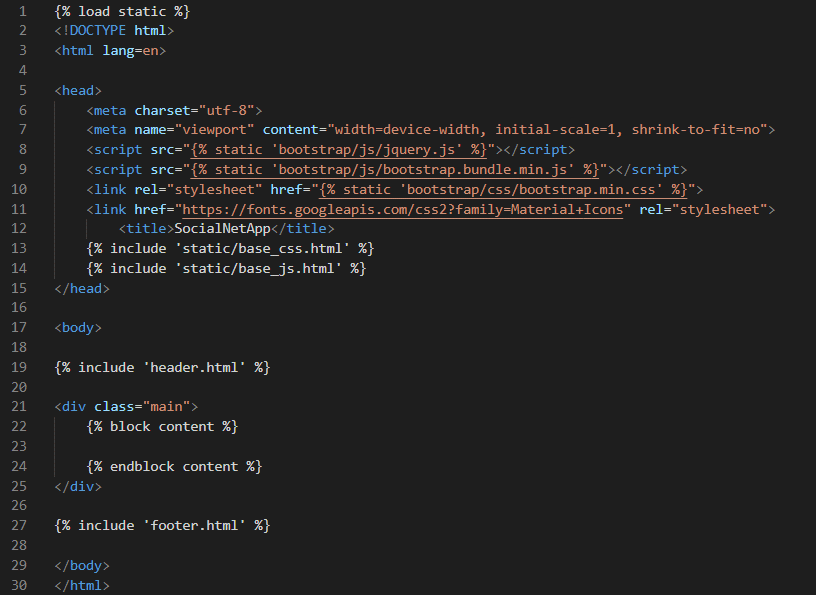


Figure .2 – base.html (src/templates/base.html)

The layout of the application will be rather standardized where each page will extend the base.html file which includes the header and footer html file while the contents are blocked in the main div. Google fonts are included to create a ‘logo’ for the SocialNetApp in the header using cursive text fonts. base\_css contains some styling for the application but each individual html file may still have their own styling included in the file itself.

# **Database Model:**

Diagram, table

Description automatically generated

Figure 1.3 – Database model diagram

The database is modelled as shown in the diagram above.

When a User first creates their account, an empty friend list will be created automatically for that user. Each User should only have their own unique friend list and each friend list should belong to only 1 user and thus, it is a one-to-one relationship.

Users may send multiple friend requests to multiple users and each user may receive multiple friend requests. At any point of time, a particular friend request will belong to two users, the sender and receiver, and thus, it is a many-to-many relationship.

Users can make many posts on their home page, but each post can only have 1 poster/author and thus, it is a one-to-many relationship.

Users can open multiple channels to start a private chat with multiple people and each channel will also contain 2 users, thus, it is a many-to-many relationship.

User can send messages to a channel where each message will belong to only one channel, but a channel may have multiple messages. Each of this message can only have 1 sender and thus, they are both a one-to-many relationship.

# **Functionalities:**

### Users can create accounts

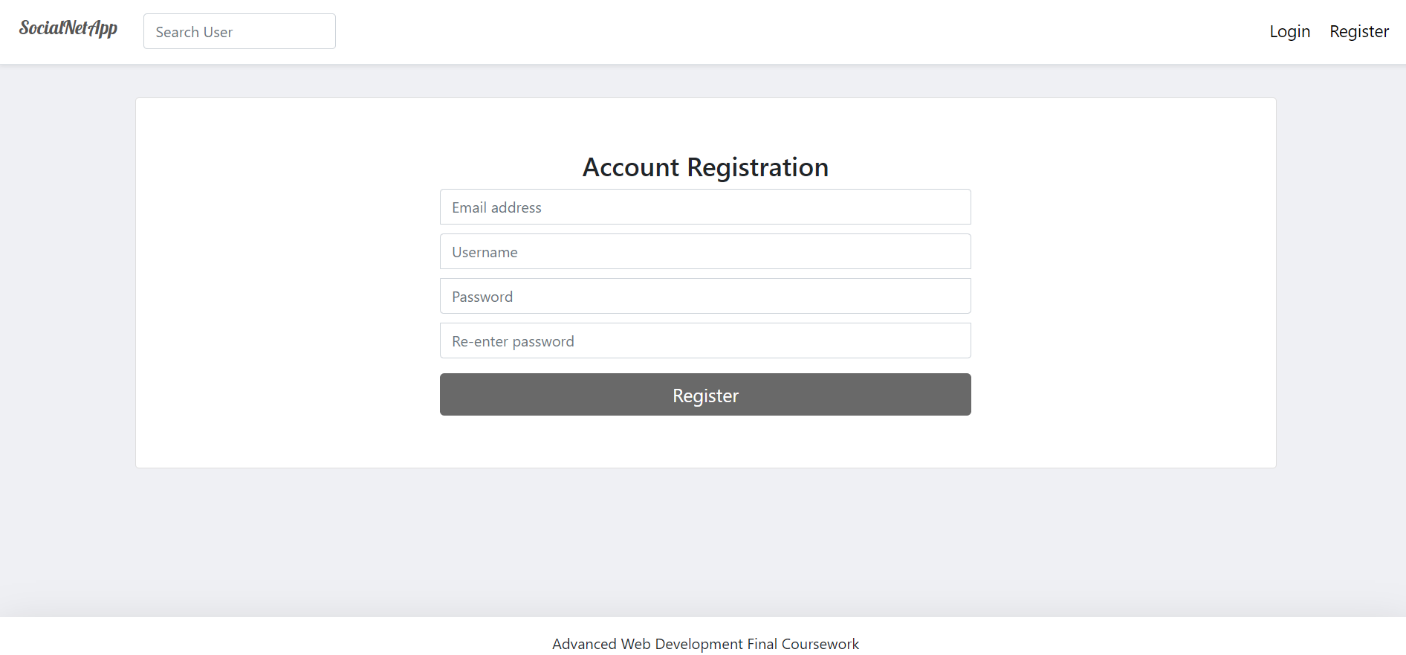


Figure 2a.1 – Account Registration Page

The registration page can be visited by clicking the register button on the header bar.



Figure 2a.2 – UserManager (users/models.py) line 1 – 30

Text

Description automatically generated

Figure 2a.3 - User Model (users/models.py) line 32 - 67

We will be creating a custom User model which inherits from AbstractBaseUser along with the use of UserManager which has functions such as create\_user() and create\_superuser() which we could customise and utilise to create our users.

Some default fields such as is\_active, is\_admin, is\_staff and is\_superuser will be overridden and we will also set our USERNAME\_FIELD (‘email’) and REQUIRED\_FIELDS (‘username’) accordingly. I have selected email to be the username field instead of the username as the username can be changed and updated while the email may not be changed. In an actual development application, it would be ideal to allow users to change the email but for the purpose of this assignment, users will not be able to do this. Password field need not be specified as the default may be used.

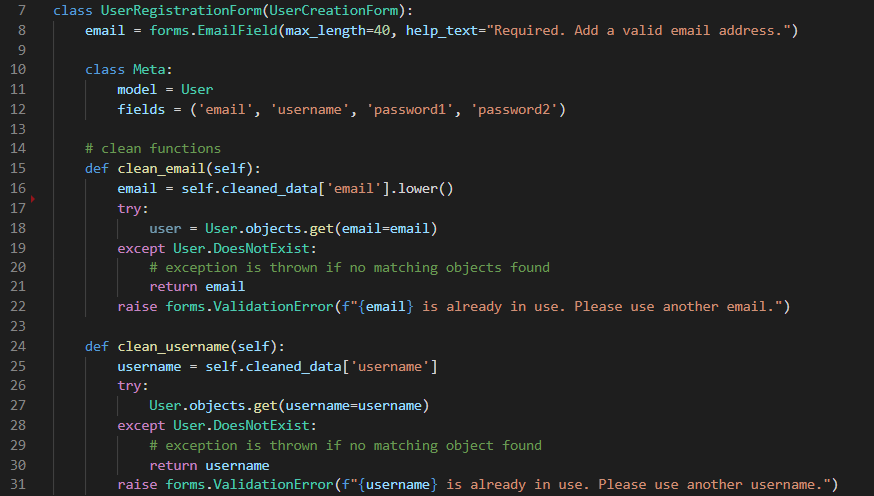


Figure 2a.4 - User Registration Form (users/forms.py) line 7 - 31

For the UserRegistrationForm, we will specify the email field using the forms.EmailField and I have chosen to set the max\_length of the email to 40 as it is uncommon for anyone to have an email longer than that.

We will be setting the model of this form based on the User model with an additional password1 and password2 fields to make users re-enter their password to confirm their password during registration.

2 additional clean functions are defined to process the email field to make it case insensitive and that the email is unique. I have made the username field case sensitive, but it must also be unique. When any form validation errors are raised, the errors are displayed.

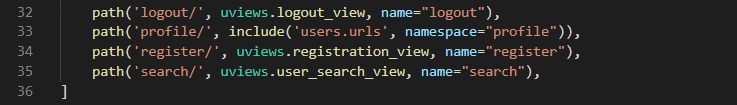


Figure 2.5 – register url path (SocialNetApp/urls.py) line 32 - 36

When the user is redirected or visits the registration url (‘register/’), the registration\_view in users/views.py will be called.

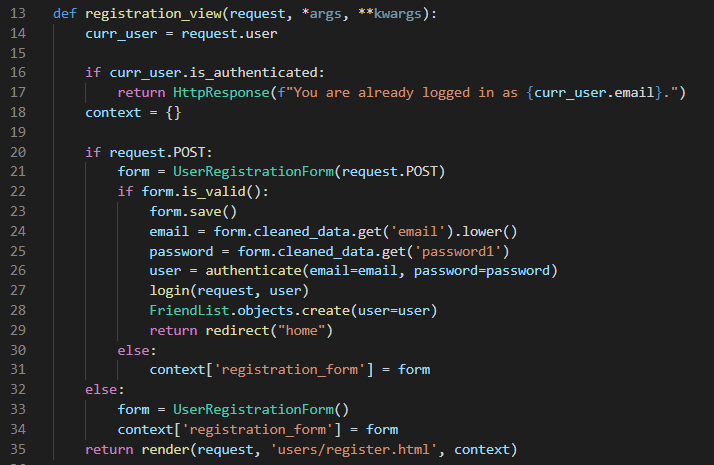


Figure 2a.6 – registration\_view (users/views.py) line 13 - 35

When the registration page is visited, we must first check if the user is already authenticated. Authenticated users will still be able to visit the registration page by entering the url into the browser despite the register button being not available to them.

If the user first visits the page, the request method will be “GET” and we will simply render the   
“users/register.html” page with an empty context. If the user clicks the submit button in the registration form in register.html page, the request method will be “POST” and we will proceed to validate the UserRegistrationForm.

If any form validation errors are raised, the validation errors will be passed to the context before re-rendering the register.html page with the updated context. If the form is valid, the form is saved, and user will be created and authenticated. The user will then be logged in and an empty friend list object will be created for that user, before redirecting the user to the home page.

#### **Validations:**

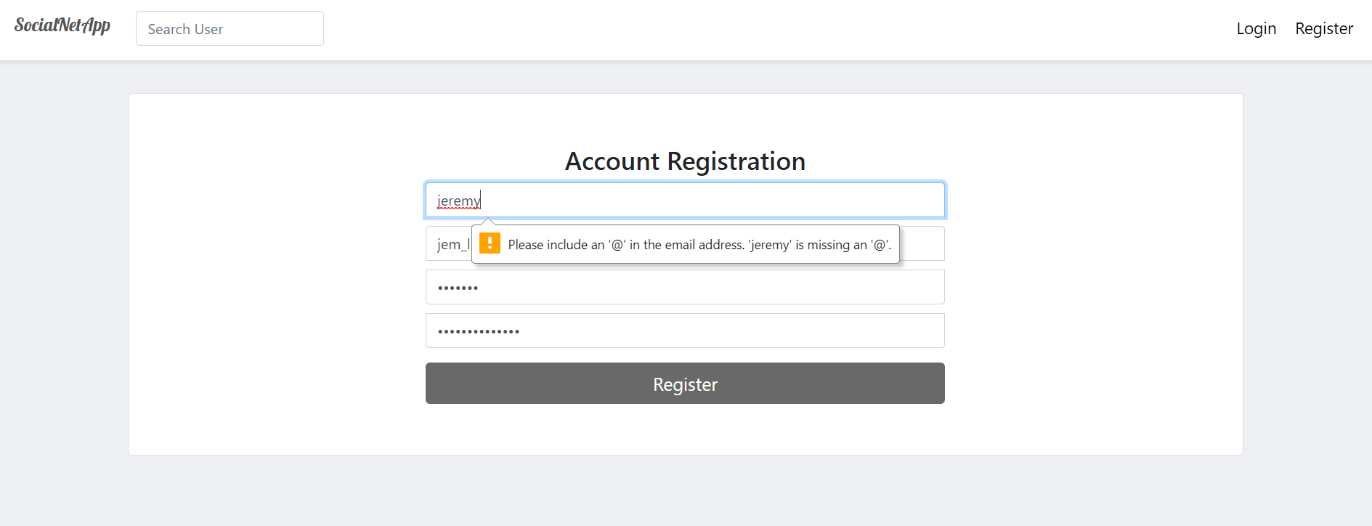


Figure 2a.7 – Account Registration Page, Form Validation – invalid email

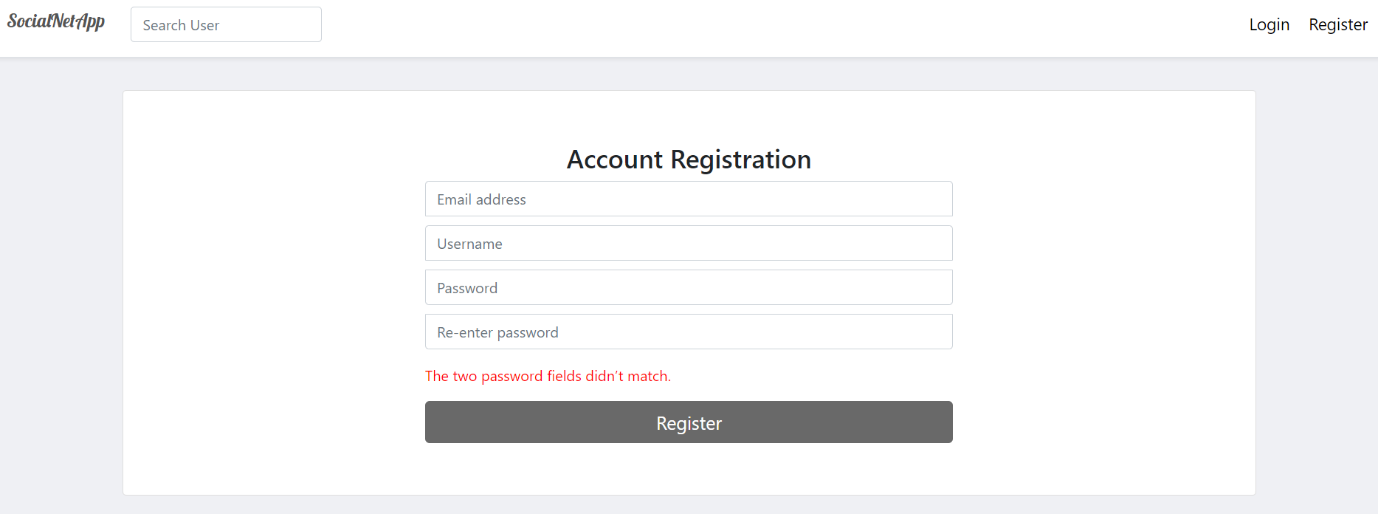


Figure 2a.8 – Account Registration Page, Form Validation – password does not match

User inputs will be validated by the form, ensuring that the email input is in the format of an email and that both password fields must be the same.

### Users can log in and log out

The login page can be visited by clicking the login button on the header bar.

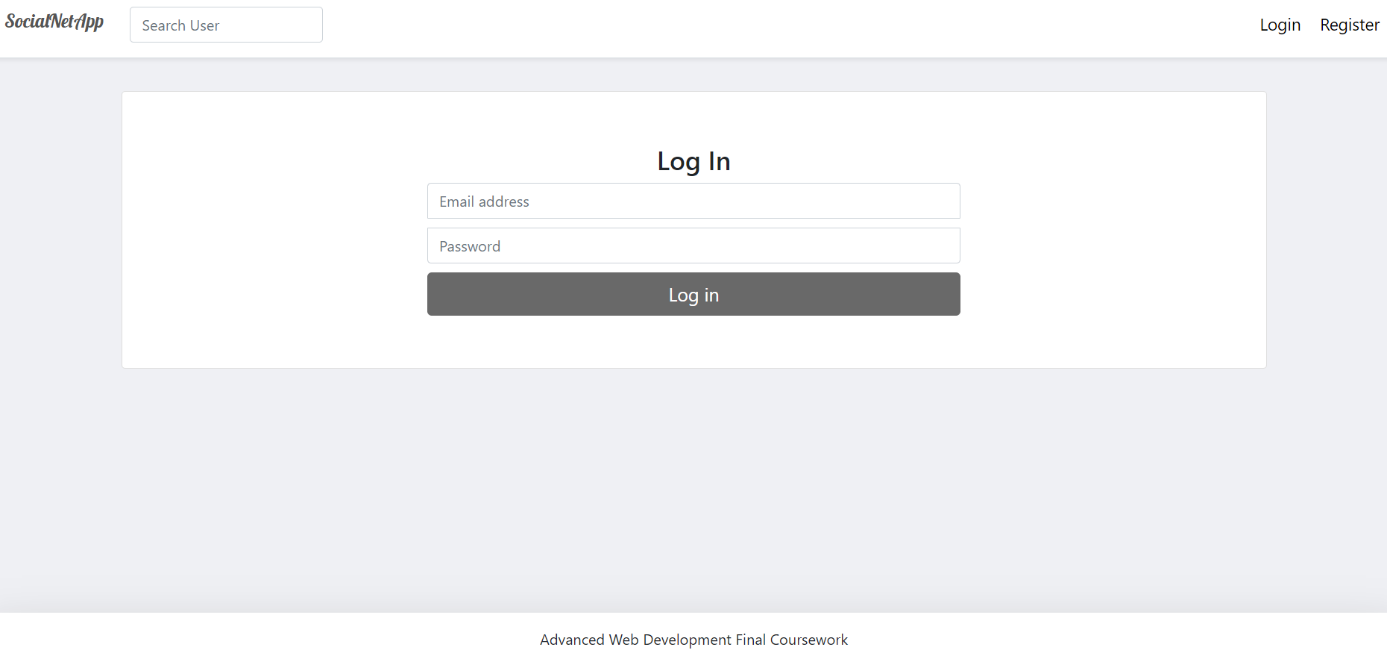


Figure 2b.1 – Login Page

The logout feature can be accessed while the user is logged in. When the user is authenticated, the login and register button on the header will be replaced with 3 icons. The right most icon indicates the profile and when clicked, a dropdown menu will be displayed which contains the following.

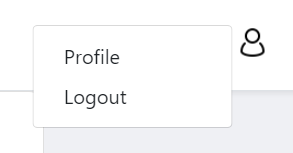


Figure 2b.2 – Logout Widget on Header

#### **Login Process:**

When the login button is clicked, the user is redirected to the url of name ‘login’

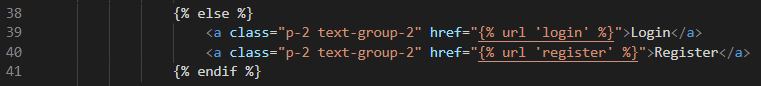


Figure 2b.3 – login and register button (templates/header.html) line 38 - 41



Figure 2b.4 – login path (SocialNetApp/urls.py) line 31

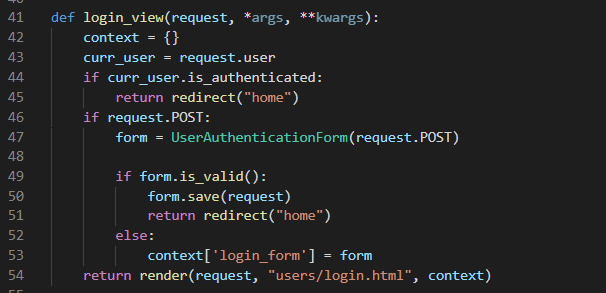


Figure 2b.5 – login\_view (users/views.py) line 41 – 54

The login\_view (users/views.py) view function will check if user is authenticated and if true, they will be redirected to the home page. If false, they will simply be directed to the login page for authentication. Next we will check if the current request is a post request and if true, we will validate the user’s login field inputs using UserAuthenticationForm. If the inputs for the form are valid, we will run the save() method of the UserAuthenticationForm which authenticates and log the user in, before redirecting the user to the ‘home’ path. If form is not valid, the form with validation error is assigned to the context key ‘login\_form’ which will be passed to the users/login.html along with the context.

A screenshot of a computer screen

Description automatically generated

Figure 2b.6 – UserAuthenticationForm (users/forms.py) line 33 – 53

Case insensitive model is included to allow forms to validate the username input from user regardless of the cases for the input. USERNAME\_FIELD is set to the email field in this application when writing the User model in users/models.py.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

#### **Logout Process:**

When the logout button is clicked, the user is redirected to url of name ‘logout’

A screenshot of a computer

Description automatically generated

Figure 2b.7 – profile icon drop down menu (templates/header.html) line 25 - 35

A screenshot of a computer

Description automatically generated

Figure 2b.8 – logout path (SocialNetApp/urls.py) line 32

Unlike login\_view, logout\_view (users/views.py) view function does not require us to check if the user is authenticated as regardless of their authentication status, they will be redirected to the home page after being logged out.

A screenshot of a computer

Description automatically generated

Figure 2b.9 – logout\_view (users/views.py) line 37 - 39

#### **Validations:**

Graphical user interface, text

Description automatically generated

Figure 2b.10 – Login Page, Form Validation – invalid email or password

User inputs will be validated by the form and users will be logged in only if the password field input matches the email input user in the database.

### c) Users can search for other users

Users, regardless of authentication status, may search for other users through the search bar on the header bar.

Graphical user interface, text, application

Description automatically generated

Figure 2c.1 – Search User Bar Widget on Header

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Figure 2c.2 – Search Query Result for ‘b’ query (users/user\_search.html)

When a user submits the search query by pressing the ‘enter’ key, a javascript function, executeQuery() will be called.

A screenshot of a computer

Description automatically generated

Figure 2c.3 – user search form (templates/header.html) line 7 - 10

A screenshot of a computer

Description automatically generated

Figure 2c.4 – executeQuery function for user search (templates/header.html) line 48 - 52

The executeQuery() function retrieves the user input by getting the value of the input element of the id “user\_q”. That query value is then user to redirect the user to the url of name ‘search’, with additional string value of “?q=” and the query value. The function must return false at the end to cancel the default form submit action.

A screenshot of a computer

Description automatically generated

Figure 2c.5 – search path (SocialNetApp/urls.py) line 35

The path with name ‘search’ will run the user\_search\_view in users/views.py.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 2c.6 – user\_search\_view view function (users/views.py) line 114 - 131

In the user\_search­\_view, we will check if the request method is GET even if there are no forms that will activate the post method in the search view to prevent any unintended paths which may lead to bugs.

We will get the search query using request.GET.get(“q”) and check if the len of that query is larger than 0. If the query is empty, we will render the user\_search.html page with an empty context.

We will use that search query and filter User objects based on any partial matches with usernames in the database. We may also include email but in this assignment, I have made it such that only the user itself should be able to see their own email in the profile and thus, no one else should be able to search for another user based on a users’ email.

From line 122 onwards, this is done to check if any of the searched users are friends with the current users which the page will display the result differently.

We will first check if the user is authenticated as unauthenticated users will not have any friends. If true, the user’s friend list will be retrieved, and we will iterate through the searched users and append the user object along with a Boolean returning function, FriendList.is\_mutual\_friend() which checks if the current iteration user object is present in the friend list.

We will then assign the processed searched users into the ‘profiles’ key of context and the user\_search.html page will be rendered.

### d) Users can add other users as friends

Users may add other users as friend when they visit their profile and click on the send friend request button. Once it is clicked, that user will receive a friend request and if that user accepts the request, they will both be friends. Users may accept friend requests either through the friend requests list page or by visiting the sender’s profile to accept friend request.

Graphical user interface, text

Description automatically generated

Figure 2d.1 – user jem\_lkw visits user dave’s profile and click on send friend request (users/profile.html)

A screenshot of a computer

Description automatically generated with medium confidence

Figure 2d.2 – user dave visits his own profile and click on friend requests (users/profile.html)

Graphical user interface, text, application, email

Description automatically generated

Figure 2d.3 – user dave accepts user jem\_lkw’s friend request by clicking tick icon (friends/friend\_requests.html)

A screenshot of a computer

Description automatically generated

Figure 2d.4 – user dave’s friend list page (friends/friend\_list.html)

A screenshot of a computer

Description automatically generated

Figure 2d.5 – FriendList Model (friends/models.py) line 4 - 28

The FriendList model consists of user and friends field where the user field has a one to one field relationship with the settings.AUTH\_USER\_MODEL which is the User model which we have defined in settings.py. Each user’s friend list can have none to many Users as their friend and each User can belong to none or many friend lists.

Add\_friend() function will add the specific user only if the user is not already in the user’s friend list while remove\_friend() function ensures that the specific user is in the user’s friend list before removing him/her.

The unfriend() function we call upon the remove\_friend() function to remove the specific user from the current user’s friend list and remove the current user from the specific user’s friend list.

The is\_mutual\_friend() function will be used when a user is visiting another user’s friend list. If another user’s friend list contains a mutual friend with the current user, the function returns true.

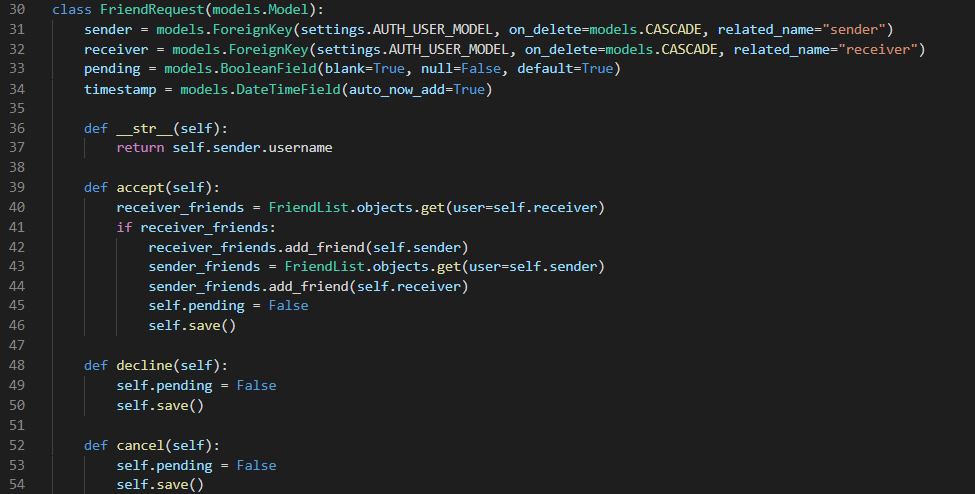


Figure 2d.6 – FriendRequest Model (friends/models.py) line 30 - 54

The FriendRequest model will require 4 fields, sender, receiver, pending and timestamp. Pending field will be used to determine if the friend request is still pending action from the receiver. Timestamp field has an auto\_now\_add parameter set to True so that whenever a new friend request is made, the timestamp is added automatically based on the time of request.

The accept() function will add the sender of the request into the receiver’s friend list and receiver of the request into the sender’s friend list. The pending status of the request is then set to False. The decline() or cancel() function simply sets the request status to False.

A screenshot of a computer

Description automatically generated

Figure 2d.7 – profile.html line 122 – 123 (users/templates/users/profile.html)

A screenshot of a computer

Description automatically generated with medium confidence

Figure 2d.8 – profile.html line 213 – 225 (users/templates/users/profile.html)

A screenshot of a computer

Description automatically generated with medium confidence

Figure 2d.9 – sendFriendRequest function ajax request (friends/templates/friends/send\_friend\_request.html) line 2 - 26

When the user clicks the send friend request button, the sendFriendRequest() function runs and launches a ajax request. This ajax request directs the user to the path of name “friend\_request” in the ‘friend\_app’ application with a “POST” method. This url path can be found in the friends/urls.py which calls the send\_friend\_request view function.

Text

Description automatically generated

Figure 2d.10 – send\_friend\_request() function (friends/views.py) line 22 – 56

We first check if the request method is “POST” and if the user is authenticated we will get the receiver user\_id and find the user object based on that id. Next, we will filter out for any FriendRequest objects with the sender as current user and the receiver as the respective id. If no FriendRequest objects are found, we will create a new FriendRequest and assign the ajax\_content of key ‘result’ to success.

If any requests are found with the pending field value as true, we will raise an exception stating that the requests have already been sent, this should not happen in the normal application run but we should still check for it. A new FriendRequest is then made. We will return a HttpResponse with the ajax\_content as json. This will update the current page that the user is in which should be the profile page of the receiver user.

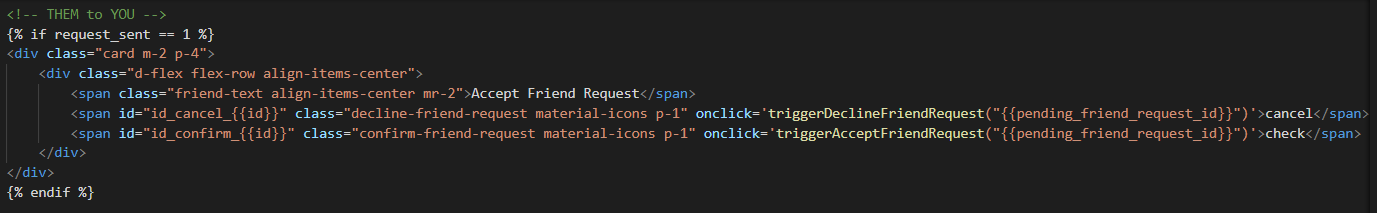


Figure 2d.11 – profile.html (users/templates/users/profile.html) line 108 – 117

The profile.html checks if the request\_sent status is equal to 1. The friend request statuses can be seen below in the enum class FriendRequestStatus.

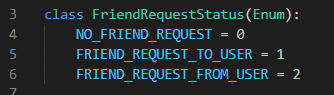


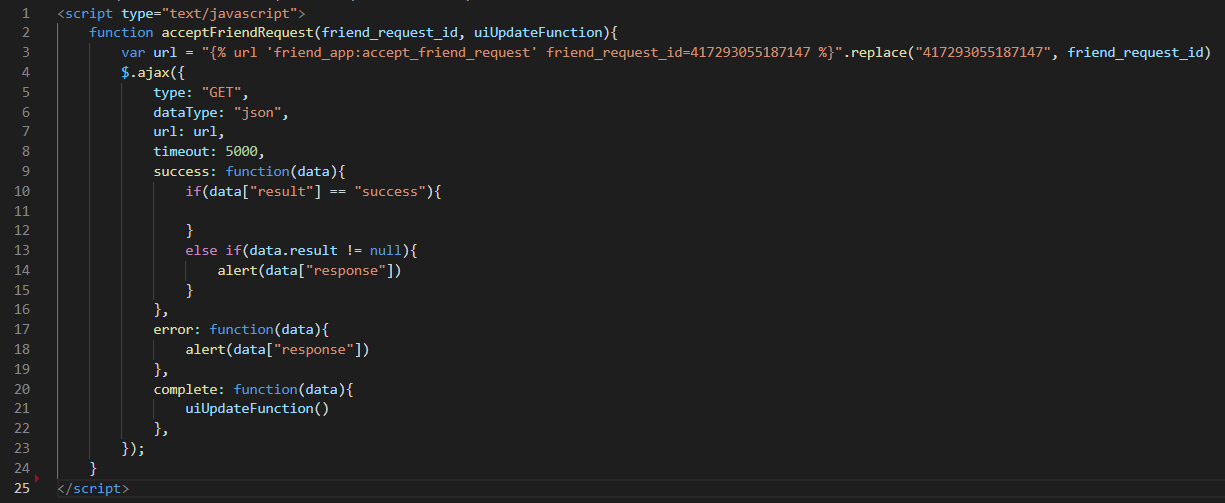
Figure 2d.12 – FriendRequestStatus enum class (friends/friend\_request\_status.py)

When a friend request is sent to a user, they may go to the sender’s profile to accept or decline the friend request by clicking the span “check” and “cancel” icon respectively. Clicking them triggers the following two function.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 2d.13 – triggerAcceptFriendRequest() and triggerDeclineFriendRequest (users/profile.html) line 235 - 240

Figure 2d.14 – acceptFriendRequest function ajax request (friends/accept\_friend\_request.html) line 1 - 25

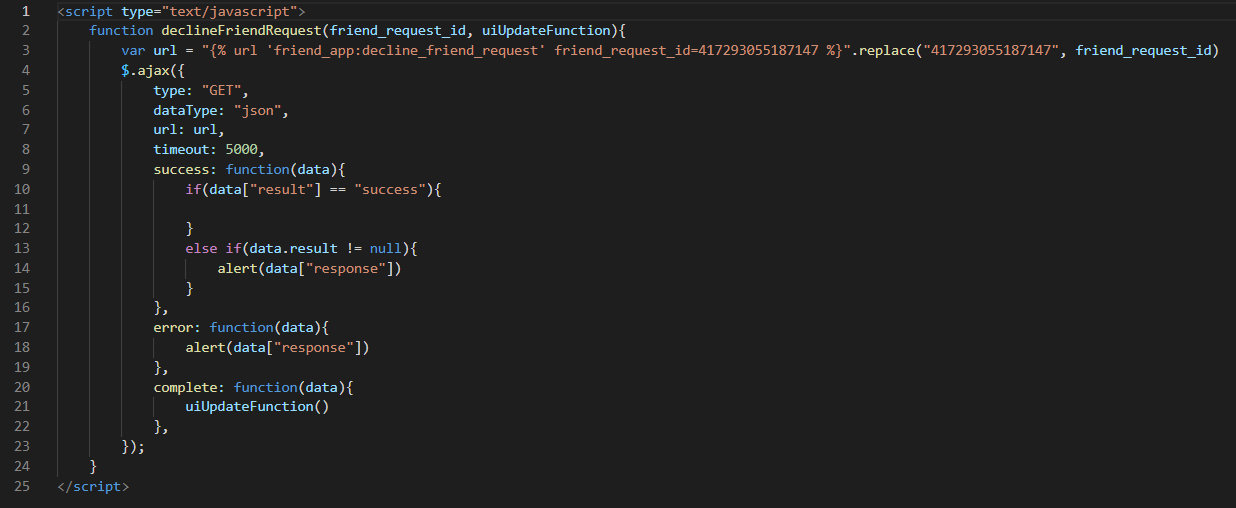


Figure 2d.15 – declineFriendRequest function ajax request (friends/decline\_friend\_request.html) line 1 - 25

These javascript functions uses ajax responses to redirect users to the accept and decline friend request urls. These urls do not render any html pages but will invoke the accept\_friend\_request and decline\_friend\_request functions in views.py. The initial url in the ajax request is given a random unguessable friend\_request\_id keyword before replacing it with the actual friend\_request\_id as it cannot be written in such a way that we can use the actual friend request id within the curly braces.

Text

Description automatically generated

Figure 2d.16 – accept\_friend\_request() function (friends/views.py) line 58 - 82

When the accept\_friend\_request function is called, we will first ensure that the request is a “GET” method and that the user is authenticated. We will retrieve the keyword argument “friend\_request\_id” passed from the ajax request. We will check if the friend request exists and if true, we will get that specific friend request object. Next, we will ensure that the current user is the receiver as only the receiver may accept friend requests, and run the accept() function for that friend request.

Text

Description automatically generated

Figure 2d.17 – decline\_friend\_request() function (friends/views.py) line 84 - 108

The decline\_friend\_request function works very similarly to accept\_friend\_request. The only difference is the method called when all conditions are met which are accept() and decline() respectively.



Figure 2d.18 – friend\_request\_view() function (friends/views.py) line 7 – 20

Friend requests may also be accepted from the friend requests page which lists down all pending friend requests. Accepting and declining any friend requests from that page will result in the same process as explained above.

### e) Users can chat in realtime with friends

Graphical user interface, text, application

Description automatically generated

Figure 2e.1 – user dave’s friend list page (friends/templates/friends/friend\_list.html)

Graphical user interface, text, application

Description automatically generated

Figure 2e.2 – new chat between user ‘dave’ and ‘jem\_lkw’ (friendchat/templates/friendchat/chat.html)

Graphical user interface, text, application

Description automatically generated

Figure 2e.3 – chat between user ‘dave’ and ‘jem\_lkw’ (friendchat/templates/friendchat/chat.html)

Graphical user interface, text, email

Description automatically generated

Figure 2e.4 – chat list page (friendchat/templates/friendchat/chatlist.html)

A screenshot of a computer

Description automatically generated

Figure 2e.5 – Channel and Message Model (friendchat/models.py) line 33 - 50

The Channel model will have name and users field to store the necessary information about the channel that users will be connected to and will also an objects field based on the ChatManager model manager. The Message model will have foreign keys for both the channel and sender field and a content field of TextField() to record what sender is sending.

A screenshot of a computer

Description automatically generated

Figure 2e.6 – ChatManager (friendchat/models.py) line 5 - 24

The ChatManager consists of 2 functions get\_or\_create\_personal\_channel() or by\_user(). The get\_or\_create\_personal\_channel function first retrieves all channel objects with get\_queryset() and filters those objects to get distinct channels that have user1 and user2 in the users list. We will also filter the channels based on the number of user count in the users field and they must be 2.

We then check if any channels exist based on those filters and if any are found, we will return the first channel object in that list as there would only be one possible channel that would match the filter criteria based on private chat messaging. If no channels are found after filtering, we will begin to create a near channel.

Each channel is given a unique name which is formulated by first sorting the users by their email initials before ‘@’ alphabetically. This sorted list will help us determine which users’ pk and email initial will come first. If user1 is alphabetically in front, the channel name will be: [user1\_pk] + [user1\_email\_initial] + [user2\_pk] + [user2\_email\_initial] and if the opposite is true, the user order is reversed.

Once the channel is created, we will add both users into the users field and return the created channel.

Text

Description automatically generated

Figure 2e.7 – ChatManager (friendchat/models.py) line 5 - 24

When a user clicks the ‘Send a message’ button in the friend list page or ‘Chat’ button in the friend chat list page, that user will be redirected to friend\_chat url with the pk of the friend object as the friend\_id keyword. This path will then call the chat\_view view function.

A screenshot of a computer

Description automatically generated

Figure 2e.8 – friendchat app urls (friendchat/urls.py) line 6 - 9

A screenshot of a computer

Description automatically generated

Figure 2e.9 – chat view function (friendchat/views.py) line 8 - 28

Similar to other functions, we will first check if the user is authenticated and proceed if true. Next, we will retrieve the friend\_id keyword argument and retrieve the user object for that friend id. We will then pass both the current user and friend user object to the get\_or\_create\_personal\_channel() function in the ChatManager. After the result for the channel object is returned, we will check to ensure that the channel is not empty.

If the program runs correctly, we will assign the channel and friend object to their key ‘channel’ and ‘friend’ respectively. We will then filter all messages based on the channel and assign it to the ‘messages’ key. If the request method is “GET”, the chat.html page renders with the context key values.

If the request method is “POST”, this means that the user has sent a message on the chat.html page. We will get the content that the user has post and create the Message object with the current user as the sender. We will then update the context.

This allows users to send message to one another.

It was intended to make this chat feature an async websocket real-time chatting feature but while coding along with the methods used as taught in the coursera video, the json context data that was passed from the views to the chatroom.html does not allow Messages objects to be passed along with it and that means that existing messages in the database will not be shown. Several attempts have been made to search for solutions as to how to pass the messages objects as json objects but none have worked and thus, I have decided to create a chat system that can correctly store and display messages between users instead of a real time chat with no capabilities of displaying the stored messages in the database.

### f) Users can add status updates to their home page

For part f and g, I have opted for an approach which works similarly to facebook’s post feature where users may choose to post a status post with text only, or they may choose to post a photo along with a status/caption with it. I will be explaining how they work in part g.

### g) Users can add media (such as images to their account and these are accessible via their home page

When a user visits the home page, they will see all posts made by themselves and their friends where the latest posts will be at the top while the oldest posts will be at the bottom. They will also be able to upload image and post status through the form at the top.

Graphical user interface, text, application

Description automatically generated

Figure 2g.1 – user dave’s view of homepage (userposts/templates/userposts/home\_page.html)

Graphical user interface

Description automatically generated

Figure 2g.2 – uploaded image and written post, yet to click post (userposts/templates/userposts/home\_page.html)

Graphical user interface, application

Description automatically generated

Figure 2g.3 –post button clicked (userposts/templates/userposts/home\_page.html)

Graphical user interface, text, website

Description automatically generated

Figure 2g.4 – user post without image (userposts/templates/userposts/home\_page.html)

A screenshot of a computer

Description automatically generated

Figure 2g.5 – url paths (SocialNetApp/urls.py) line 21 - 28

When users visit the index url path ‘http://127.0.0.1:8080/’, they will be redirected to the url path of name ‘home’ which runs the home\_page view function in userposts/views.py.

A screenshot of a computer

Description automatically generated

Figure 2g.6 – home\_page view function (userposts/views.py) line 24 – 48

When users visit the index url path ‘http://127.0.0.1:8080/’, they will be redirected to the url path of name ‘home’ which runs the home\_page view function in userposts/views.py. When the user visits the home page and have not submitted a post form, we will check if the user is authenticated and retrieve the user’s friend list if true.

If the friend list is not empty, we will include the current user into the friend list and retrieve all user posts that are made by any of the users in the friend list. This queryset is then sorted by the ‘timestamp’ field and reversed to allow the latest posts to be at the top of the home page. This reversed order can actually be achieved through the reversed() function or by simple adding a ‘-‘ at the front of the timestamp field (...order\_by(‘-timestamp’).

If the friend list is empty, we will simply retrieve user posts that are made by the current user and at the retrieved posts are assigned to the ‘posts’ key of context. The homepage is then rendered.

A screenshot of a computer

Description automatically generated

Figure 2g.7 – home\_page view function (userposts/views.py) line 9 – 23

When a user clicks on the post button and the request method is ‘POST’, we will validate the inputs from user to the UserPostForm. This form ensures that the ‘content’ field is not empty. A user may upload a post without an image, but the text must not be empty.

If the form is valid, we will first retrieve the uploaded file from request.FILES with the specified ‘post\_image\_file\_selector’ key based on name field of the <input> for files. We will then retrieve the url for the file that we will be uploading it to based on the MEDIA\_ROOT path written in settings.

Uploaded posts will be saved in the path and format of ‘media\_cdn/<user.pk>/<user.pk>.png’. When a file of the same name is saved into the folder, the new file is renamed with an additional ‘\_[7\_unique\_random\_characters]’. For example, if a user pk is 2, and the user already has a post image with the name of 2.png, a new file may be named as ‘2\_d6REOCx.png’. After the image is saved, a new user post is created and the remaining code as seen in figure 2g.6 will run and re-render the home page.

If no file is found, the exception is caught, and we will simply create a UserPost object without image.

Editing Profile and Profile Image:

Users may edit their profile information by clicking the profile icon and clicking profile on the header bar. In the profile page, click update button and the user will be brought to the edit user page.

Graphical user interface, text

Description automatically generated

Figure 2g.8 – user profile page (users/templates/users/profile.html)

A screenshot of a computer

Description automatically generated with medium confidence

Figure 2g.9 – user profile page, update button clicked (users/templates/users/edit\_user.html)

Graphical user interface, text, application

Description automatically generated

Figure 2g.10– user profile page, image uploaded but save button not clicked (users/templates/users/edit\_user.html)

Graphical user interface, application, Word, PowerPoint

Description automatically generated

Figure 2g.11 profile\_image in media\_cdn for current user changed (users/templates/users/edit\_user.html)

After uploading the image, the image will only be saved if the save button is clicked. As we click the save button, we can notice that the profile picture does not seem to have been changed. However, if we go to the media\_cdn folder itself, we can see that the profile picture is changed. I have found that this is an issue related to how image data are cached. After clearing my browser cache data and refreshing the profile page, the updated profile image can be seen.

Graphical user interface, text, application

Description automatically generated

Figure 2g.12 image updated after clearing browser cache

A screenshot of a computer

Description automatically generated

Figure 2g.13 profile\_view (users/views.py) line 56 - 75

Like other function views, when the user is redirected to the profile page, the profile function-based view runs. We will first get the user\_id keyword argument and retrieved the user object of that keyword. We will then assign the necessary field values of the user object to the key value pairs of context, which is used to display the user information in the profile page.

Next, we will retrieve the friend list of the user object which will be used to display the number of friends the user has in the profile page. We will also be assigning friends variable with all the friends in the friend list.

A screenshot of a computer

Description automatically generated

Figure 2g.14 profile\_view (users/views.py) line 77 - 110

Next, we will declare variables with some initial values that we will be using. If the current user is authenticated and the viewed profile is not the current user, we will set is\_self to false. If the current user exists in the list of friends, we will set is\_friend to true.

Else, we will set is\_friend to false and check if any existing friend request is sent to the current user by the viewed profile user. If true, we will assign the enum value of FRIEND\_REQUEST\_TO\_USER to request\_sent.

Else if there is any existing friend request sent by current user to the viewed profile user, we will assign the enum value of FRIEND\_REQUEST\_FROM\_USER to request\_sent.

If not requests are made, we will simply assign the enum value of NO\_FRIEND\_REQUEST to request\_sent.

If current user is not authenticated, we will simply set is\_self to false and no friend request information is displayed.

If user is authenticated but current user is the viewed profile user, we will simply retrieve the friend requests where current user is the receiver and the request is pending.

Finally, we will assign all those values to the context keys and render the profile page with those context values.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 2g.15 edit\_user\_view (users/views.py) line 133 - 163

We will first check if the user is authenticated and if not the user is redirected to the login path. If user is authenticated, we will retrieve the user\_id keyword argument and get the user object. If user object is not the same as the current user, we will return a http response stating that a user can only edit their own profile.

If the request is post, we will validate the inputs using the UserUpdateForm and declare a Boolean variable image\_updated.

First, we check if there are any files uploaded from the ‘profile\_image\_file\_selector’ <input> in the html page when the user clicks the save button (invokes ‘POST’ request). If true, we will retrieve that file, define the profile url using MEDIA\_ROOT path and assign the path to the FileSystemStorage. Before we save the file to that path, we must first delete the user profile image. Once it is deleted, the file is saved to the system and the user’s profile\_image field is updated. Image\_updated is then set to true.

If form is valid, we will simply save the form and update the user information and the user will be redirected to the profile page.

Else if the form is not valid but image was updated, user will also be redirected to the profile page.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 2g.16 edit\_user\_view (users/views.py) line 164 - 185

If the form is invalid and no image was updated, we will reset the form fields to the current user’s information and pass on the form to context and the page is re-rendered, effectively resetting the page to when the user first visits it.

### h) correct use of models and migrations

Every time the model for a new app was written as shown in various figures above, I will run the manage.py makemigrations and migrate commands which generates the table into the database as seen below of a screenshot using DB browser for SQLite.

Graphical user interface, text, application, Word

Description automatically generated

Figure 2h.1 – DB browser for SQLite for db.sqlite3, list of tables in the database

Graphical user interface, text, application

Description automatically generated

Figure 2h.2 – DB browser for SQLite for db.sqlite3, list of all migrations

### i) correct use of form, validators and serialisation

As seen from screenshots of various figures above, forms, and form validations are used when appropriate to handle user inputs and any validation errors are caught are displayed to the users by re-rendering the current page with the updated context validation error in the key ‘form’.

UserCreationForm was used for UserRegistrationForm, where the email and username are validated using clean functions. forms.ModelForm used for various forms throughout the application to validate and process various user inputs at different portions of the application.

Validation errors are caught and are easily understandable by users to notify them of what input is causing the validation error.

A screenshot of a computer

Description automatically generated

Figure 2i.1 – User Serializers, users/serializers.py (line 4 to 12)

User and FriendList serializers are written for their respective APIs. UserSerializer will be used to display user information through the UserDetail API while UserCreationSerializer is used for the UserCreation API.

A screenshot of a computer

Description automatically generated

Figure 2i.2 – FriendList Serializer, friends/serializers.py (line 4 to 12)

The FriendList model will have UserSerializer as its nested serializer in order for the API to be more meaningful. Without the nested serializer and simply using the FriendList model, the API will return the id of the friend list for ‘user’ field, and a list of id for the ‘friends’ field.

### j) correct use of django-rest-framework

Function-based views of Django rest framework have been used to handle requests from users and to create the response to each of them in various pages of the application.

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

Figure 2j.1 – CreateUser api

A screenshot of a computer

Description automatically generated

Figure 2j.2 – CreateUser api, users/api.py (line 10 to 24)

The CreateUser API has a HTML form to enter the user information and POST button to create the user. The user will be created based on the UserRegistrationForm.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Figure 2j.3 – UserDetail api

The UserDetail API returns the json format data including the listed fields in the UserSerializer. More fields can be included as required but for security reasons, password should typically be hidden.

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

Figure 2j.4 – FriendDetail api

A screenshot of a computer

Description automatically generated

Figure 2j.4 – FriendDetail api, friends/api.py (line 9 to 14)

The FriendDetail API returns the json format data of the user and friends, where the user is the owner of the friend list. Friends field includes all users’ information who are friends with the user. This is the result of the nested serializer.

### k) correct use of URL routing

Users can be redirected to various paths in the application, or they may also enter the url routes into the browser manually which will bring them to the respective page. Their requests are processed and if necessary, the user may be redirected to other url routes.

### l) appropriate use of unit testing

A screenshot of a computer

Description automatically generated

Figure 2l.1 – FriendRequestTest (friends/tests.py) line 6 to 25

A screenshot of a computer

Description automatically generated

Figure 2l.2 – HomePageViewTest (userposts/tests.py) line 5 to 14

A screenshot of a computer

Description automatically generated

Figure 2l.3 – UsersViewTest (users/tests.py) line 5 to 15

Some unit tests are written to test and ensure that function-based views run as intended. Creation of users are also tested to ensure that the correct user of correct user type is created.

### m) An appropriate method for storing and displaying media files is given

As seen from previous figures, media files are uploaded and stored in the media\_cdn folder with static files being stored in the static folder.

# **Summary:**

In summary, SocialNetApp has achieved most of the functionalities required of this coursework apart from the real-time chat between 2 users. This feature works well when following along the coursera video on real-time chat, but the feature could not be completed due to my inability to find a solution to integrate the websocket chat with the database.

One other issue which I believe is due to caching issue of the web browser is that the profile image does not update when user profile image is edited until the browser cache is cleared. I have found that there are other developers also facing the same issue, but they have stated that in a fully deployed web application, this issue no longer occurs.

Some areas that can be improved is better image handling as the current profile image is simply resized to fit the defined width and height. A crop image feature would be a good addition to this project.

An area that can be improved is to write more tests as there are not many tests written for this coursework. More research will need to be done on my end to learn more on how to write meaningful and appropriate tests and the use of packages such as coverage to help with the testing process.

# **Appendix:**

Graphical user interface

Description automatically generated with low confidence

The above is the list of users and every user with the exception of the admin user have the password in the format of:

[email\_initials] + “password”

e.g. username: dave@gmail.com password: davepassword

accounts with most test data for demonstration:

1. dave@gmail.com, davepassword
2. jeremy@gmail.com, jeremypassword

For the admin account:

Superuser:

Email: admin@superemail.com

Username: admin

Password: theadminpassword