

Advanced Web Development Finals Coursework

For this coursework, we are tasked with building our own social network. Here is a list of functions required:

- 1) Users can create accounts
- 2) Users can log in and out
- 3) Users can add status updates to their home page
- 4) Users can add media
- 5) Users can search for other users
- 6) Users can add other users as friends
- 7) Users can chat in real-time with friends

Setup

To begin with this project, some packages had to be installed.

- pip install django
- pip install djangorestframework
- pip install channels
- pip install redis

For this project, I decided to create one application called `socialmedia_app` to store all my application functions. I then added some lines of code due to the new applications.

```
5 # Application definition
6
7 INSTALLED_APPS = [
8     'channels',
9     'socialmedia_app',
10    'django.contrib.admin',
11    'django.contrib.auth',
12    'django.contrib.contenttypes',
13    'django.contrib.sessions',
14    'django.contrib.messages',
15    'django.contrib.staticfiles',
16    'rest_framework',
17 ]
18
```

I created a file called `apps.py` and saved `socialmedia_app` as an `app.py`.

```
socialmedia_app > apps.py > ...
1  from django.apps import AppConfig
2
3
4  class SocialmediaAppConfig(AppConfig):
5      default_auto_field = 'django.db.models.BigAutoField'
6      name = 'socialmedia_app'
7
```

I also created the following directories in my CW2 folder.

- static
- static_cdn
- media
- media_cdn
 - o temp

The cdn files are created to simulate what an actual file hosting server does when static or media files are being collected. I added media files to the media folder to be used, and I used the command to collect the files and transfer them into the cdn folders.

```
2> python manage.py collectstatic
```

Also, I downloaded bootstrap to help simplify the process for creating html files. Bootstrap is a front-end framework and open sourced, hence I was able to use some of its HTML and CSS features for UI elements such as for the forms and buttons. I also used JavaScript extensions.

I then created a templates folder to store all my html code. In it, I created another snippets folder with html files that supplements the other webpage html files that I will create for each function later.

```
✓ templates
  ✓ snippets
    <> footer.html
    <> header.html
    <> home_css.html
    <> home_js.html
```

In this, there is a header and footer file that will be added to the base webpage.

All these files will be extended into every webpage such that the header acts as a navigation bar with links that allows the user to jump to any page from any page, and the base is in every webpage for aesthetic reasons. The screenshots below show how they look like when rendered.

Header



Footer



The html for header file is as follows. Line 9 shows how the link to the home page is added as a url, while lines 11-14 shows how the search box is created, and it acts as a form that will take in an input to query the database, before returning the desired webpage.

```
5
6      <!-- MEDIUM+ SCREENS -->
7      <div class="d-none d-md-flex flex-row my-auto flex-grow-1 align-items-center">
8          <h5 class="mr-3 font-weight-normal justify-content-start">
9              <a class="p-2 text-dark" href="{% url 'home' %}">Home</a>
10          </h5>
11          <form class="search-bar justify-content-start" onsubmit="return executeQuery();">
12              <input type="text" class="form-control" name="q" id="id_q_large" placeholder="Search...">
13          </form>
14      </div>
15
```

The code below shows how the user profile is shown in the most left corner. Also, the link before it takes the user to the user's own profile page, as specified in link 31. And the link before that takes the user to the chat room, as specified in line 25.

```
22
23      <div class="btn-group dropleft">
24          <div class="d-flex notifications-icon-container rounded-circle align-items-center mr-3" id="id_chat_notificat
25              <a href="{% url 'chat_selection' %}">Chatroom</a>
26          </div>
27      </div>
28
29      <div class="btn-group dropleft">
30          <div class="account-image rounded-circle m-auto d-block dropdown-toggle" id="id_profile_links" aria-haspo
31              <a href="{% url 'profile' user_id=request.user.id %}">Account</a>
32          </div>
33      </div>
34
```

When the site is not logged in by a user, it displays two links for registration and login instead.

```
41     </div>
42     {% else %}
43     <a class="p-2 text-dark" href="{% url 'login' %}">Login</a>
44     <a class="btn btn-outline-primary" href="{% url 'register' %}">Register</a>
45     {% endif %}
46 </nav>
```

The screenshot below shows how the header looks like when no one is logged in.



The code below shows how the footer is rendered.

```
6
7 <!-- Footer -->
8
9 <div class="d-flex flex-row align-items-center footer bg-white shadow-lg mt-1">
10 |   <p class="m-auto">Social Media App | 2022</p>
11 </div>
12 <!-- End Footer -->
```

The index.html file is a template that holds combines the previous header and footer templates to a webpage, to be adhered to every single other function-based html website later. Line 22 and 23 calls in the header and home styling templates and appends it to the top, before adding a main body with lines 33 and 37 and adding appending the footer in line 41.

```
20
21     <!-- Header -->
22     {% include 'snippets/header.html' %}
23     {% include 'snippets/home_css.html' %}
24
25     <!-- Ensures that footer stays at the bottom of webpage -->
26     <style type = 'text/css'>
27         .main{
28             min-height:100vh;
29             height: 100%;
30         }
31     </style>
32
33     <!-- Main Body -->
34     <div class="main block">
35         {% block content %}
36
37         {% endblock content %}
38     </div>
39
40     <!-- Footer -->
41     {% include 'snippets/footer.html' %}
42
```

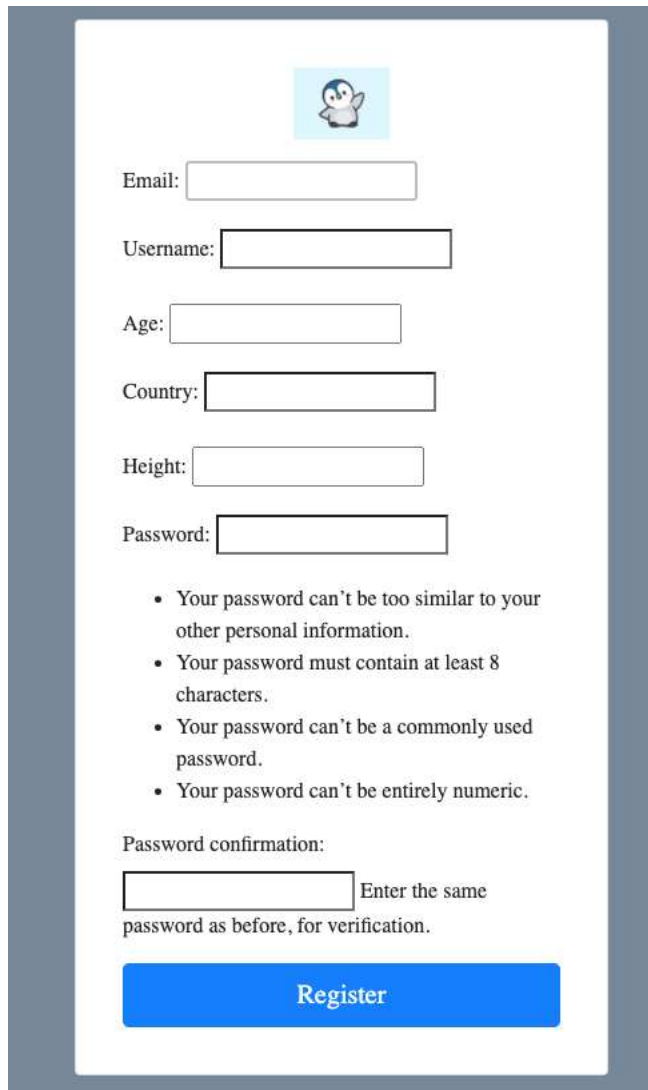
The code below shows how javascript files are being used from bootstrap to help decorate the site.

```
42
43     <!-- using javascript files from bootstrap -->
44     <script src="{% static 'bootstrap/js/jquery.min.js' %}"></script>
45     <script src="{% static 'bootstrap/js/bootstrap.bundle.min.js' %}"></script>
46 </body>
```

In a sense, this is the creation of the base styling of the entire social media app.

Registration

The following screenshot shows the page rendered for registration. It uses a card for styling and has a form to take in information about a user.

A screenshot of a registration form styled as a card. The card has a light blue border and a white background. At the top center is a small square icon of a penguin. Below the icon are six input fields, each with a label to its left: 'Email:', 'Username:', 'Age:', 'Country:', 'Height:', and 'Password:'. The 'Password:' field is followed by a bulleted list of four password requirements. Below the list is a 'Password confirmation:' section with an input field and the text 'Enter the same password as before, for verification.' At the bottom of the card is a blue button with the text 'Register' in white.

The following screenshots are from registration.html file. Line 36 shows how it is styled in a card while the following shows how the csrf token and form.as_p are used to take in the user input into the forms. Line 42 gives the penguin logo for the register card.

```
36  iv class="card-body">
37    <form class="form-signin" method="post">
38      {% csrf_token %}
39      <!-- Logo for login card -->
40      <div class="d-flex flex-column pb-3">
41        <!-- 
44      </div>
45      <div>
46        {{form.as_p}}
47      </div>
48    </form>
```

The code below shows how input is fed into the registration forms.

```
48
49     {% for field in registration_form %}
50     <p>
51         {% for error in field.errors %}
52         <p style="color: red">{{ error }}</p>
53         {% endfor %}
54     </p>
55     {% endfor %}
56     {% if registration_form.non_field_errors %}
57     <div style="color: red">
58         <p>{{registration_form.non_field_errors}}</p>
59     </div>
60
61     {% endif %}
62     <button class="btn btn-lg btn-primary btn-block" type="submit">Register</button>
63 </form>
```

On the other hand, model.py has a model called MyAccountManager. This creates users with the following fields in line 11. Lines 13 to 17 shows how the email address and username are compulsory fields.

```
9
10 class MyAccountManager(BaseUserManager):
11     def create_user(self, user_email, username, age, country, height, password=None):
12         # email is a required field
13         if not user_email:
14             raise ValueError("Email address is a compulsory field.")
15         # username is a required field
16         if not username:
17             raise ValueError("Username is a compulsory field.")
```


The following fields in line 20 to 27 belong to the user model. Lines 28 to 29 sets the password and saves the user as self.

```
19         # if both conditions are present, create user
20         user = self.model(
21             # normalize and make it lowercase
22             user_email=self.normalize_email(user_email),
23             username=username,
24             age=age,
25             country=country,
26             height=height,
27         )
28         user.set_password(password)
29         user.save(using=self._db)
30         return user
31
32     # customer user model to create a superuser
33     def create_superuser(self, user_email, username, password):
34         user = self.create_user(
35             user_email=self.normalize_email(user_email),
36             username=username,
37             password=password,
38         )
39         user.is_admin = True
40         user.is_staff = True
41         user.is_superuser = True
42         user.save(using=self._db)
43         return user
44
```

Lines 33 to 43 creates a superuser with the following fields: username and password, and stores the fields is_admin, is_staff, and is_superuser as true as they are all true fields for superusers only.

In forms.py, the class below saves the fields into an imported Django UserCreationForm that helps create a form using the model Account and takes in the following fields.

```
6
7     class RegistrationForm(UserCreationForm):
8         age = forms.IntegerField()
9         country = forms.CharField(max_length=255)
10        height = forms.IntegerField()
11
12        class Meta:
13            model = Account
14            fields = ['user_email', 'username', 'age',
15                    'country', 'height', 'password1', 'password2']
16
```

```
PS C:\Users\antho\Documents\Advanced Web Development\awd_finalterm\CW2> python manage.py createsuperuser
Email: admin@mail.com
Username: admin
Password:
Password (again):
The password is too similar to the username.
This password is too common.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
PS C:\Users\antho\Documents\Advanced Web Development\awd_finalterm\CW2> 
```

Login

The following screenshot shows how the rendered file for login.html looks like.



The following screenshots are from login.html file. Line 30 shows how it is styled in a card while the following shows how the csrf token and form.as_p are used to take in the user input into the forms. Line 34 gives the penguin logo for the register card.

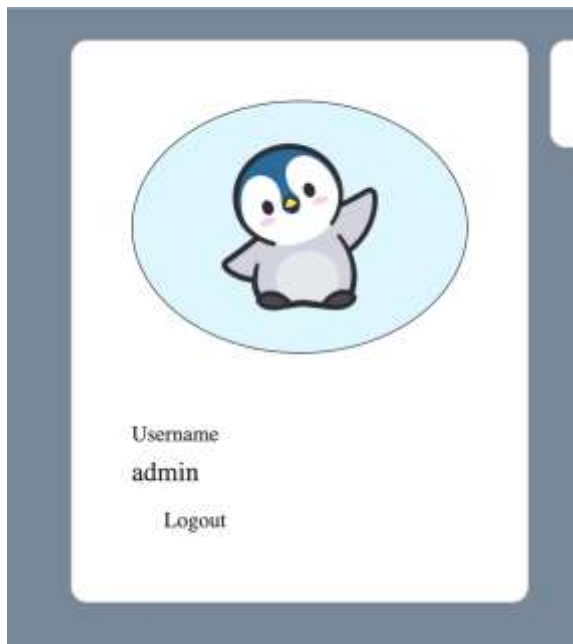
```
30 <div class="card-body">
31   <form class="form-signin" method="post">{% csrf_token %}
32     <div class="d-flex flex-column pb-3">
33       <!-- 
34       
35     </div>
36
37     <div>
38       {{form.as_p}}
39     </div>
40   </form>
41 </div>
```

The code below shows how input is fed into the login forms.

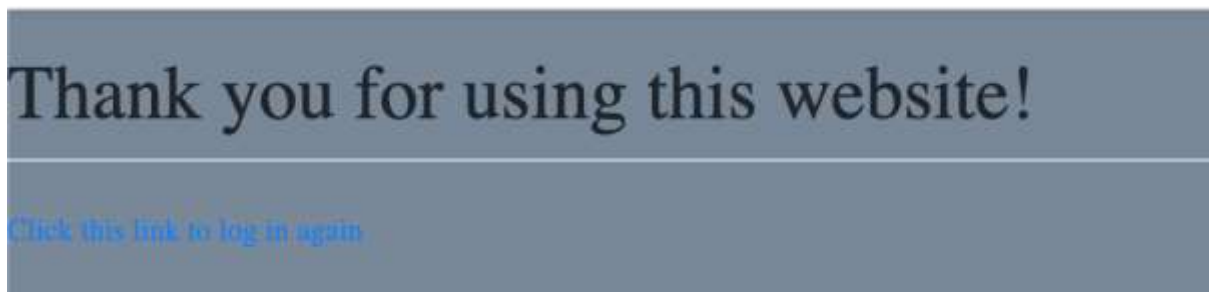
```
36
37     <div>
38         {{form.as_p}}
39     </div>
40
41     {% for field in login_form %}
42     <p>
43         {% for error in field.errors %}
44         <p style="color: red">{{ error }}</p>
45         {% endfor %}
46     </p>
47     {% endfor %}
48     {% if login_form.non_field_errors %}
49     <div style="color: red">
50     <p>{{login_form.non_field_errors}}</p>
51     </div>
52
53     {% endif %}
54
55     <button class="btn btn-lg btn-primary btn-block" type="submit">Log in</button>
56 </form>
57
```

Logout

The profile page has a link that allows users to logout when clicking on it. It will be elaborated in the html file for profile later,



Clicking on the link will log the user out and render the following page. They can also click on the login or registration button in the header again if they wish to re-enter the site.

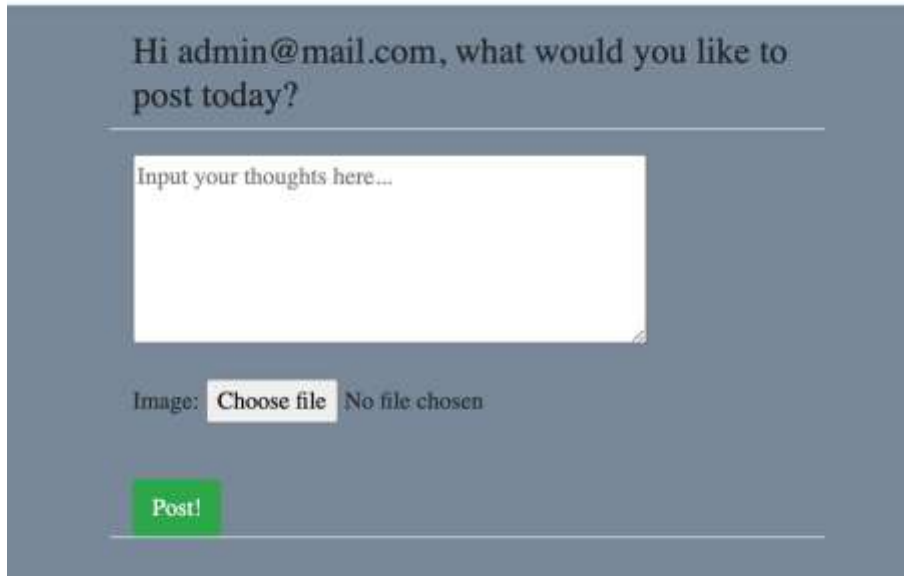


The code below shows how the link is rendered in login.html file. Line 6 shows the link from url.py to re login when clicked.

```
1  {% extends "index.html" %}
2  {% block content %}
3      <h2>Thank you for using this website!</h2>
4      <div class="border-top pt-3">
5          <small class="text-muted">
6              <a href="{% url 'login' %}">Click this link to log in again</a>
7          </small>
8      </div>
9  {% endblock content %}
10
```

Add status updates to their home page

The screenshot below shows the site rendered for the home page. This allows users to update their statuses or input their thoughts and post it onto the status update wall. This is public, so all users can see this.



Hi admin@mail.com, what would you like to post today?

Input your thoughts here...

Image: No file chosen

The posts below show how the post was uploaded into the home page publicly.



admin2@mail.com March 27, 2022, 1:25 p.m.

what a lovely day!

admin@mail.com March 27, 2022, 10:54 p.m.

Oh, it's raining!

This is rendered by the home.html file, where a post header in line 8 is created, which takes in the field user from models account to personalize page. Line 13 to 21 shows how the form for posting is displayed. The form method is post.

```
{% csrf_token %}
```

The csrf token tag in line 16 refers to the middleware in the settings.py file, and it cross checks this token for any requests entering with token from the server side during the rendition of the site. This helps prevent malicious attacks when using forms.

```
{{ form.as_p }}
```

The form tag above from line 17 renders the Django form as a paragraph. This is to make sure that all the input from the forms enter the paragraph tags respectively.

Line 20 creates a button to post.

```
5      <!-- creating a post header -->
6      <div class="row justify-content-center mt-3">
7          <div class="col-md-5 col-sm-12 border-bottom">
8              <h4> Hi {{user}}, what would you like to post today? </h4>
9          </div>
10     </div>
11
12     <!-- displaying form for post -->
13     <div class="row justify-content-center mt-3 mb-5">
14         <div class="col-md-5 col-sm-12 border-bottom">
15             <form method="POST" enctype="multipart/form-data">
16                 {% csrf_token %}
17                 {{ form.as_p }}
18                 <!-- create a post button -->
19                 <div class="d-grid gap-5">
20                     <button class="btn btn-success mt-3">Post!</button>
21                 </div>
22             </form>
23         </div>
24     </div>
```

The code below will display each post saved in `post_list`, which will be explained later. It will render the username and date of post as shown in line 30.

```
25
26     <!-- displaying each post in post_list -->
27     {% for post in post_list %}
28     <div class="row justify-content-center mt-3">
29         <div class="col-md-5 col-sm-12 border-bottom">
30             <p><strong>{{ post.user }}</strong> {{ post.date_Post }}</p>
31             <!-- if youre posting an image -->
32             {% if post.image %}
33             
34             {% endif %}
35             <p>{{ post.text }}</p>
36         </div>
37     </div>
38     {% endfor %}
```

In `model.py`, the model `Post` is created to store post data. The following fields are required for each post.

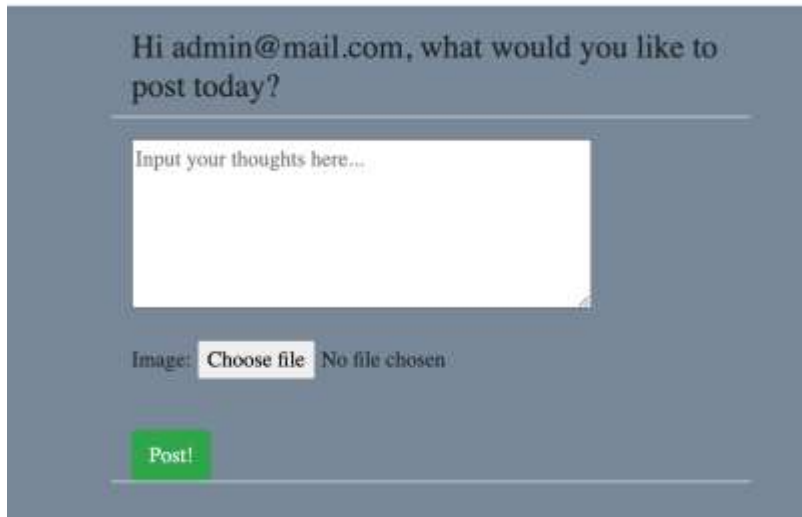
```
# model for each post
class Post(models.Model):
    post_Id = models.AutoField(primary_key=True)
    user = models.ForeignKey(Account, on_delete=models.DO_NOTHING, related_name='posts')
    date_Post = models.DateTimeField(default=timezone.now)
    text = models.CharField(max_length=250)
    image = models.ImageField(upload_to='post_image', blank=True, null=True)
```

`forms.py` then inherit this model to create a Django `ModelForm` called `NewPostForm`, to link the post form to the model fields.

```
16
17 class NewPostForm(forms.ModelForm):
18     text = forms.CharField(
19         label='',
20         widget=forms.Textarea(attrs={
21             'rows': '5',
22             'placeholder': 'Input your thoughts here...'
23         }))
24
```


Users can add media

In addition to the previous section on posting text, users can also post pictures as well. The home page below shows how users can upload a chosen file and post or update their status there.



Hi admin@mail.com, what would you like to post today?

Input your thoughts here...




Image: No file chosen

The post below shows how an image was uploaded to the home page and thus status was updated.



The following codes were added to home.html. Lines 32 to 35 are tags that take in images from the forms and post them to the page.

```
25
26     <!-- displaying each post in post_list -->
27     {% for post in post_list %}
28     <div class="row justify-content-center mt-3">
29         <div class="col-md-5 col-sm-12 border-bottom">
30             <p><strong>{{ post.user }}</strong> {{ post.date_Post }}</p>
31             <!-- if youre posting an image -->
32             {% if post.image %}
33             
34             {% endif %}
35             <p>{{ post.text }}</p>
36         </div>
37     </div>
38     {% endfor %}
```

The model below shows how image is a field in the Post model in post.py.

```
# model for each post
class Post(models.Model):
    post_Id = models.AutoField(primary_key=True)
    user = models.ForeignKey(Account, on_delete=models.DO_NOTHING, related_name='posts')
    date_Post = models.DateTimeField(default=timezone.now)
    text = models.CharField(max_length=250)
    image = models.ImageField(upload_to='post_image', blank=True, null=True)
```

form.py also takes in the image field as shown in line 24 and 28.

```
16
17     class NewPostForm(forms.ModelForm):
18         text = forms.CharField(
19             label='',
20             widget=forms.Textarea(attrs={
21                 'rows': '5',
22                 'placeholder': 'Input your thoughts here...'
23             }))
24         image = forms.ImageField(required=False)
25
26         class Meta:
27             model = Post
28             fields = ['text', 'image']
29
```

Search for other users: searching

The search bar is rendered in the header. The code was explained earlier in the header.html section.



After searching, it will render the search_result webpage. It looks like the screenshot below.



The code snippet for it to work is as shown below. It is from the search_result.html file. Line 39 links the user back to their profile page with the url account. Line 41 loads the profile image of the user into the card.

```
35     {% if accounts %}
36     <div class="d-flex flex-row flex-wrap">
37     {% for account in accounts %}
38     <div class="card flex-row flex-grow-1 p-2 mx-2 my-2 align-items-center">
39         <a class="profile-link" href="{% url 'profile' user_id=account.0.id %}">
40             <div class="card-image m-2">
41                 
42             </div>
43         </a>
44         <a class="profile-link" href="{% url 'profile' user_id=account.0.id %}">
45             <div class="card-center px-2">
46                 <h4 class="card-title">{{account.0.username}}</h4>
47                 {% if account.1 %}
48                 <p class="card-text"><a href="#" onclick="createOrReturnPrivateChat('{{account.0.id}}')">Ser
49                 {% endif %}
50             </div>
51         </a>
```

Search for other users: profile page

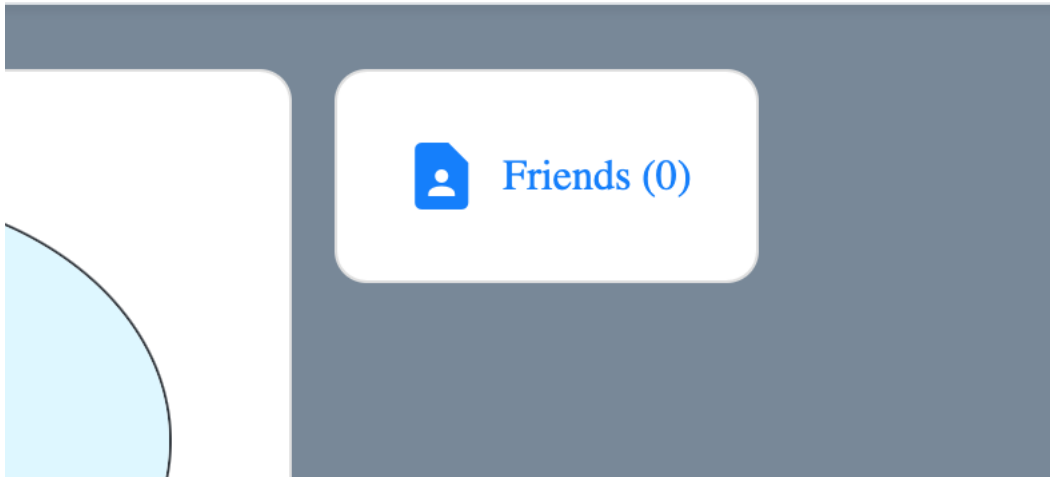
Clicking on the username from the search results in the previous section will navigate the user to the search person's profile page. Alternatively, users can click on the top right account link to navigate to their own user page if they wish. The profile page has the username of the user and a default picture. It also has a link for users to log out with.



The following code is from account.html. It renders the account of the user by detecting if the user profile being viewed as the logged in user first. This is checked in line 49. Line 41 to 43 renders the default picture while line 51 renders the logout link that links users to the logout page.

```
39
40
41     <!-- first row with image -->
42     <div class="image-container mx-auto mb-4">
43         
45     </div>
46
47     <p class="mt-4 mb-1 field-heading">Username</p>
48     <h5>{{username}}</h5>
49     <!-- check if the user is seeing their own profile -->
50     {% if is_self %}
51     <!-- logout button -->
52     <a class="dropdown-item" href="{% url 'logout' %}">Logout</a>
53     {% endif %}
54 </div>
55 </div>
```

The screenshot shows how the profile page includes a link to the friends list.



The code below shows that if the user is authenticated, they are able to check their friends list from 64 to 68.

```
57
58     {% if request.user.is_authenticated %}
59     <div class="d-flex flex-column mb-4">
60
61         <div class="card m-2 px-4 pb-4">
62             <!-- link to friends list -->
63             <div class="d-flex flex-column pt-4">
64                 <a href="#">
65                     <div class="d-flex flex-row align-items-center justify-content-center icon-conta
66                         <span class="material-icons mr-2 friends-icon">contact_page</span><span
67                         class="friend-text">Friends ({{friends|length}})</span>
68                 </div>
69                 </a>
70             </div>
71         </div>
```

Add other users as friends

Unfortunately, I am facing some issues with the friends systems and due to time constraints, had to remove the entire system so as to be able to run my code. However, I do have a model of FriendList as shown below. It takes in the fields user and friends. It has the functions to add friends and to check if the user is their friend.

```
96 class FriendList(models.Model):
97     # one friendlist per user
98     user = models.OneToOneField(
99         settings.AUTH_USER_MODEL, on_delete=models.CASCADE, related_name="user")
100     friends = models.ManyToManyField(
101         settings.AUTH_USER_MODEL, blank=True, related_name="friends")
102
103     # returns the username
104     def str(self):
105         return self.user.username
106
107     def add_friend(self, account):
108         # if user account is not part of friendlist, add friend
109         if not account in self.friends.all():
110             self.friends.add(account)
111             self.save()
112
113     def is_mutual_friend(self, friend):
114         # check if the user is a friend
115         if friend in self.friends.all():
116             return True
117         return False
118
```

The FriendRequest model takes in the sender, receiver, is_active and timestamp fields to be used to create a database for friend requests.

```
118
119 class FriendRequest(models.Model):
120     sender = models.ForeignKey(
121         settings.AUTH_USER_MODEL, on_delete=models.CASCADE, related_name="sender")
122     receiver = models.ForeignKey(
123         settings.AUTH_USER_MODEL, on_delete=models.CASCADE, related_name="receiver")
124     # checks if friend request is active
125     is_active = models.BooleanField(blank=False, null=False, default=True)
126     # time entry when friend request was sent
127     timestamp = models.DateTimeField(auto_now_add=True)
128
129     # returns the username
130     def str(self):
131         return self.sender.username
132
```


Chat in real-time with friends

```
5 # Application definition
6
7 INSTALLED_APPS = [
8     'channels',
9     'socialmedia_app',
10     'django.contrib.admin',
11     'django.contrib.auth',
12     'django.contrib.contenttypes',
13     'django.contrib.sessions',
14     'django.contrib.messages',
15     'django.contrib.staticfiles',
16     'rest_framework',
17 ]
18
```

Firstly, I installed channel into the settings.py file. Then I added the wsgi and asgi applications. This is to replace the traditional Django development server with the channels development server.

```
59
60 WSGI_APPLICATION = 'CW2.wsgi.application'
61 ASGI_APPLICATION = 'CW2.routing.application'
62
```

I created a file called routing.py to route the application above.

```
1 from channels.auth import AuthMiddlewareStack
2 from channels.routing import ProtocolTypeRouter, URLRouter
3
4 import socialmedia_app.routing
5
6 application = ProtocolTypeRouter ({
7     'websocket': AuthMiddlewareStack(
8         URLRouter(
9             socialmedia_app.routing.websocket_urlpatterns
10         ),
11     ),
12 })
13
```

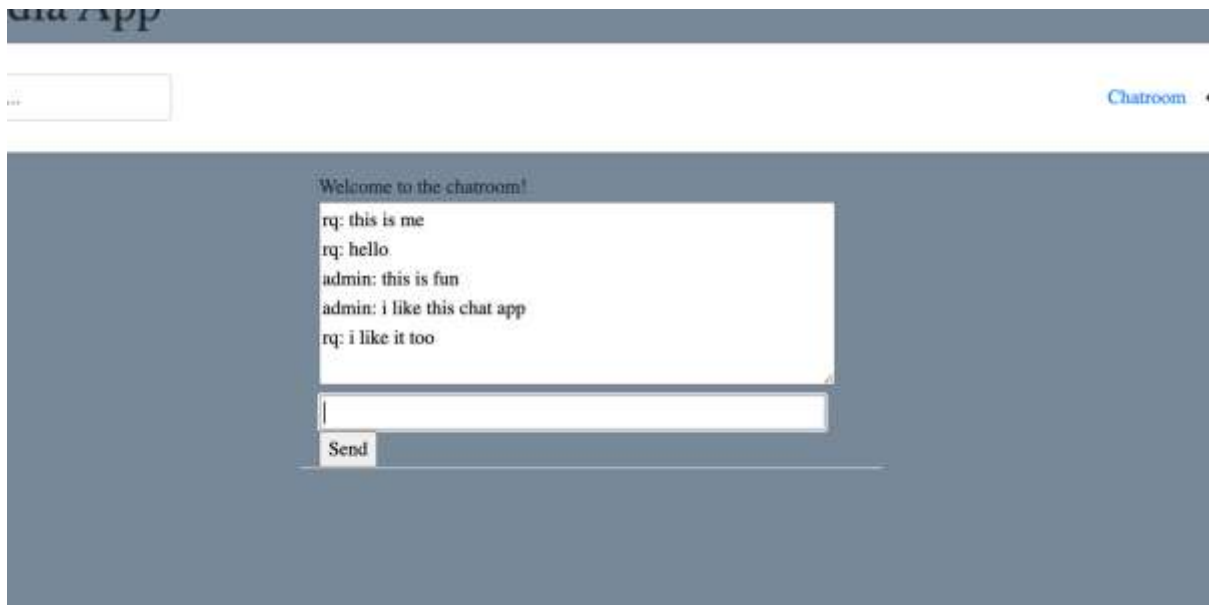
Then I created the chat_index.html file. This was a waiting room page that allowed users to enter a chat room based on the name.

What chat room would you like to enter?

The html file looks the screenshot below. The script with lines 13 to 25 shows how the user can enter a chatroom with `querySelector` to submit a room name. It will then save it into the variable `roomName` as seen in line 22 before directing users to the chatroom with the directory 'chat' + `roomName` in line 23.

```
9
10      What chat room would you like to enter?<br>
11      <input id="room-name-input" type="text" size="50"><br>
12      <input id="room-name-submit" type="button" value="Enter">
13  <script>
14      document.querySelector('#room-name-input').focus();
15      document.querySelector('#room-name-input').onkeyup = function (e) {
16          if (e.keyCode === 13) { // enter, return
17              document.querySelector('#room-name-submit').click();
18          }
19      };
20
21      document.querySelector('#room-name-submit').onclick = function (e) {
22          var roomName = document.querySelector('#room-name-input').value;
23          window.location.pathname = '/chat/' + roomName;
24      };
25  </script>
```


The screenshot below shows how the chat server looks like. Without logging out of two accounts, they can continuously send messages to each other.



The chat_room.html looks like the screenshot below. Lines 12 to 14 creates the buttons and chat boxes rendered visually. Line 15 tag is the implementation of the chat server, where if the room name matches, it creates a web socket called chatSocket. It stores the host and the users.

```
9
10 Welcome to the chatroom!<br>
11
12 <textarea id="chat-log" cols="50" rows="6"></textarea><br>
13 <input id="chat-message-input" type="text" size="50"><br>
14 <input id="chat-message-submit" type="button" value="Send">
15 {{ room_name|json_script:"room-name" }}
16 <script>
17     const roomName = JSON.parse(document.getElementById('room-name').textContent);
18
19     const chatSocket = new WebSocket(
20         'ws://'
21         + window.location.host
22         + '/ws/'
23         + roomName
24         + '/'
25     );
26
```

chatSocket appends the message into the chat log in line 29 with the function `querySelector`. Lines 36 to 41 takes in user input and stores it as the chat-message-input into the document. Upon submitting, it is then stored into the constant value `messageInputDom` and `message`, where the message is stored as it is and saved into the backend.

```
26
27     chatSocket.onmessage = function(e) {
28         const data = JSON.parse(e.data);
29         document.querySelector('#chat-log').value += (data.message + '\n');
30     };
31
32     chatSocket.onclose = function(e) {
33         console.error('Chat socket closed unexpectedly');
34     };
35
36     document.querySelector('#chat-message-input').focus();
37     document.querySelector('#chat-message-input').onkeyup = function(e) {
38         if (e.keyCode === 13) { // enter, return
39             document.querySelector('#chat-message-submit').click();
40         }
41     };
42
43     document.querySelector('#chat-message-submit').onclick = function(e) {
44         const messageInputDom = document.querySelector('#chat-message-input');
45         const message = messageInputDom.value;
46         chatSocket.send(JSON.stringify({
47             'message': message
48         }));
49         messageInputDom.value = '';
50     };
51 </script>
```

The consumer has to be implemented in `consumers.py`. The functions below allow users to either join a room group or disconnect from the room.

```
3
4 class ChatConsumer(AsyncWebsocketConsumer):
5     async def connect(self):
6         self.room_name = self.scope['url_route']['kwargs']['room_name']
7         self.room_group_name = 'chat_%s' % self.room_name
8
9         # Join room group
10        await self.channel_layer.group_add(
11            self.room_group_name,
12            self.channel_name
13        )
14        await self.accept()
15
16    async def disconnect(self, close_code):
17        await self.channel_layer.group_discard(
18            self.room_group_name,
19            self.channel_name
20        )
21
```

The functions below shows how the message is received and how the chat messages are sent.

```
21     )
22     # receive the message
23     async def receive(self, text_data):
24         username = self.scope["user"].username
25         data = json.loads(text_data)
26         message = data['message']
27         message = {username + ': ' + message}
28
29         await self.channel_layer.group_send(
30             self.room_group_name,
31             {
32                 'type': 'chat_message',
33                 'message': message,
34             }
35         )
36
37     async def chat_message(self, event):
38         message = event['message']
39         await self.send(text_data=json.dumps({
40             'message': message,
41         }))
42     })
```

Also, the channel layer implemented are asynchronous as they have to constantly update live when messages are sent.

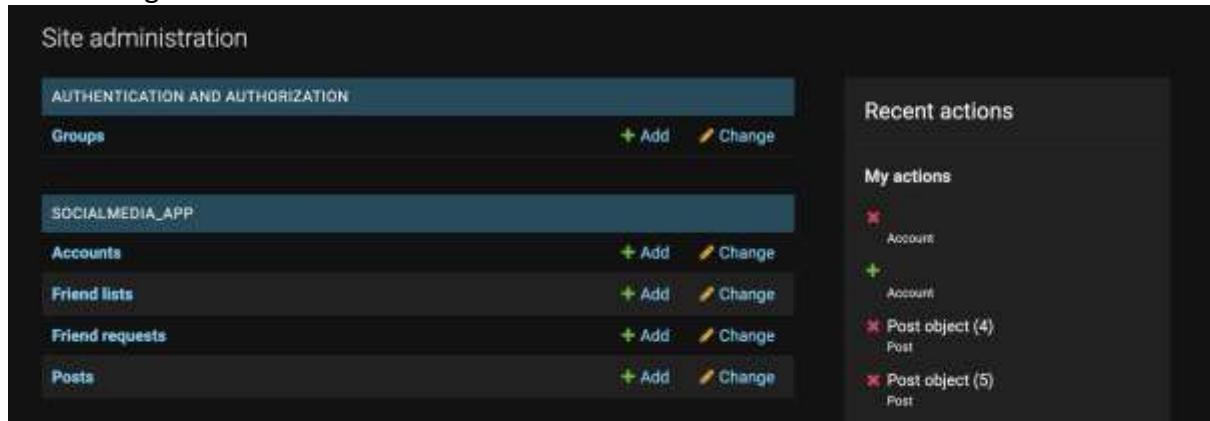
Admin

The admin.py file will be rendered to the url file later, as admin is a page on the server. The screenshot below shows the three admin classes created, the AccountAdmin, the FriendRequestAdmin, and the FriendList admin, where the last two were unable to be used.

```
7
8 class AccountAdmin(UserAdmin):
9     # define which fields are shown in django admin
10    list_display = ('user_email', 'username', 'date_joined', 'last_login', 'is_admin', 'is_staff')
11    # fields used for searching in django admin
12    search_fields = ('user_email', 'username',)
13    # information on each user is added
14    readonly_fields = ('id', 'date_joined', 'last_login')
15
16    # remove filter section
17    filter_horizontal = ()
18    list_filter = ()
19    fieldsets = ()
20
```

```
20
21 class FriendRequestAdmin(admin.ModelAdmin):
22     list_filter = ['sender', 'receiver']
23     list_display = ['sender', 'receiver',]
24     # search fields used in admin page search bar
25     search_fields = ['sender__username', 'receiver__username']
26
27     class Meta:
28         model = FriendRequest
29
30 class FriendListAdmin(admin.ModelAdmin):
31     list_filter = ['user']
32     list_display = ['user']
33     search_fields = ['user']
34     readonly_fields = ['user',]
35
36     class Meta:
37         model = FriendList
38
39 admin.site.register(Account, AccountAdmin)
40 admin.site.register(FriendList, FriendListAdmin)
41 admin.site.register(FriendRequest, FriendRequestAdmin)
```

The site is generated and looks like this.



URL

The urls.py file looks like the screenshot below. Each line is a path that when added, it will call the respective views and render the page according to the views and html.

```
10 urlpatterns=[
11     # path('', views.home_screen_view, name='home'),
12     path('', home_screen_post.as_view(), name='home'),
13
14     path('register/', views.register_view, name="register"),
15     path('login/', auth_views.LoginView.as_view(template_name='login.html'), name='login'),
16     path('logout/', auth_views.LogoutView.as_view(template_name='logout.html'), name='logout'),
17     # path('login/', views.login_view, name="login"),
18     # path('logout/', views.logout_view, name="logout"),
19
20     path('account/<user_id>', views.account_view, name='profile'),
21     path('search/', views.search_user, name="search"),
22     path('friend/friend_request/', views.send_friend_request, name='friend-request'),
23     path('chat/', views.index, name='chat_selection'),
24     path('chat/<str:room_name>', views.room, name='room'),
25
26     path('api/account/<int:user>', api.account_view, name='account'),
27     path('api/post/<str:user>', api.post_view, name='post'),
28
29 ]
30
```

Views

The following views are used respectively for each function.

```
2
3 class home_screen_post(View):
4
5 # get user input as post
6     def get(self, request, *args, **kwargs):
7         posts = Post.objects.all()
8         form = NewPostForm()
9
10        context = {
11            'post_list': posts,
12            'form': form,
13        }
14
15        return render(request, 'home.html', context)
16
17 def post(self, request, *args, **kwargs):
```

```
36
37     def post(self, request, *args, **kwargs):
38         logged_in_user = request.user
39         posts = Post.objects.all()
40         form = NewPostForm(request.POST, request.FILES)
41
42         if form.is_valid():
43             new_post = form.save(commit=False)
44             new_post.user = logged_in_user
45             new_post.save()
46
47         context = {
48             'post_list': posts,
49             'form': form,
50         }
51
52         return render(request, 'home.html', context)
53
```



```

55 def register_view(request, *args, **kwargs):
56     user = request.user
57     if user.is_authenticated:
58         return HttpResponseRedirect("You are already authenticated as " + str(user.user_email))
59
60     context = {}
61     if request.POST:
62         form = RegistrationForm(request.POST)
63         if form.is_valid():
64             form.save()
65
66             return redirect('home')
67         else:
68             context={
69                 'form' : form
70             }
71     else:
72         form = RegistrationForm()
73         context={
74             'form' : form
75         }
76     return render(request, 'register.html', context)

```

```

85
86 def account_view(request, *args, **kwargs):
87
88     context = {}
89     user_id = kwargs.get("user_id")
90     try:
91         account = Account.objects.get(pk=user_id)
92     except:
93         return HttpResponseRedirect("Something went wrong.")
94     if account:
95         context['id'] = account.id
96         context['username'] = account.username
97         context['email'] = account.user_email
98         context['profile_image'] = account.profile_image.url
99         context['hide_email'] = account.hide_email
100
101     try:
102         # retrieve friendlist of another user
103         friend_list = FriendList.objects.get(user=account)
104     except Exception as e:

```

```

163
164 def search_user(request, *args, **kwargs):
165     context = {}
166     if request.method == "GET":
167         search_query = request.GET.get("q")
168         print(search_query)
169         if len(search_query) > 0:
170             search_results = Account.objects.filter(user_email__icontains=search_query).filter(username__ico
171             user = request.user
172             accounts = [] # [(account1, True), (account2, False), ...]
173             for account in search_results:
174                 accounts.append((account, False)) # you have no friends yet
175             context['accounts'] = accounts
176
177     return render(request, "search_result.html", context)
178
179

```


Serializers and API

The serializer below will be sent to the API.

```
7
8
9 class AccountSerializer(serializers.ModelSerializer):
10     class Meta:
11         model = Account
12         fields = ['user_email', 'username', 'date_joined', 'last_login_date', 'is_admin', 'is_active',
13                 'is_staff', 'is_superuser', 'profile_image', 'hide_email']
14
15 #serializer for post model
16 class PostSerializer(serializers.ModelSerializer):
17     class Meta:
18         model = Post
19         fields = ['post_id', 'user', 'date_post', 'text', 'image']
20
```

The api.py file below uses get django rest framework to render the APIs.

```
11
12
13 @api_view(['GET'])
14 def account_view(request, user):
15     user = Account.objects.filter(user=user)
16     serializer = AccountSerializer(user, many=True)
17     return Response(serializer.data)
18
19 # creates the api to return post information
20 @api_view(['GET'])
21 def post_view(request, user):
22     user = Post.objects.filter(user=user)
23     serializer = PostSerializer(user, many=True)
24     return Response(serializer.data)
25
26
27
```

Account View

GET /api/account/2/

HTTP 200 OK

Allow: GET, OPTIONS

Content-Type: application/json

Vary: Accept

```
[
  {
    "user_email": "admin@mail.com",
    "username": "admin",
    "date_joined": "2022-03-27T11:32:38.382366Z",
    "last_login_date": "2022-03-27T11:32:38.386296Z",
    "is_admin": true,
    "is_active": true,
    "is_staff": true,
    "is_superuser": true,
    "profile_image": "/media/icon/penguin.png",
    "hide_email": true
  }
]
```

Post View

GET /api/post/2/

HTTP 200 OK

Allow: GET, OPTIONS

Content-Type: application/json

Vary: Accept

```
[
  {
    "post_Id": 1,
    "user": 2,
    "date_Post": "2022-03-27T11:25:55Z",
    "text": "just completed the mandalorian today!",
    "image": "/media/post_image/babyyoda_cZeeDNN.png"
  },
  {
    "post_Id": 3,
    "user": 2,
    "date_Post": "2022-03-27T22:54:22.360710Z",
    "text": "Oh, it's raining!",
    "image": null
  },
  {
    "post_Id": 10,
    "user": 2,
    "date_Post": "2022-03-28T10:05:26.775452Z",
    "text": "i want to visit london..."
  }
]
```

Tests

```
20
21 class test(APITestCase):
22
23     def test_AccountReturnSuccess(self):
24         account_factory = Account_Factory.create()
25         url = reverse('account', kwargs={'user': '1'})
26         response = self.client.get(url)
27         response.render()
28         self.assertEqual(response.status_code, 200)
29
30     class Meta:
31         model = Account
32
33     def test_homeview(self):
34         url = reverse('home')
35         response = self.client.get(url)
36         self.assertEqual(response.status_code, 200)
```

The following tests were created and successful when ran.

```
HTTP GET /static/test_framework/img/grid.png 304 [0.00, 127.0.0.1]
^C(CW2) bash-3.2$
(CW2) bash-3.2$ python manage.py test
Found 2 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
..
-----
Ran 2 tests in 0.054s

OK
Destroying test database for alias 'default'...
(CW2) bash-3.2$ █
```