Also, include instructions on how to install and run your app:

- A list of all packages and the versions they used for you implementation
  - requirements.txt in 'thesocialnetwork' folder
- a list of your development environment i.e. the operating system you used and the version of python
  - Microsoft Windows Version 22H2 (OS Build 19045.3448)
  - Python 3.11.4
- Instruction for logging into the django-admin site i.e. username and password
  - check superuser login details under 'Accounts'
  - django-admin site url: *127.0.0.1:8000/admin*
- Include how to run the unit tests and the location of the data loading script
  - In terminal: *python manage.py test*
  - tests.py location under 'the_app'

<u>How to run</u>

1. Extract the 'thesocialnetwork' folder and open it in the terminal or IDE of your choice.
2. Create a virtual environment through the terminal: *python -m venv .venv*
3. Activate the virtual environment: *.venv\scripts\activate*
4. Install all the requirements from the requirements.txt: *pip install -r requirements.txt*
5. Run the server, the redis server (ASGI/Channels) should also be running together with the server: *python manage.py runserver*
6. Below should be the output when the server has successfully started

```
(.venv) C:\Users\USER\Desktop\cm3035_awd_final\thesocialnetwork>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
September 18, 2023 - 10:17:01
Django version 4.2.4, using settings 'thesocialnetwork.settings'
Starting ASGI/Channels version 3.0.5 development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Accounts

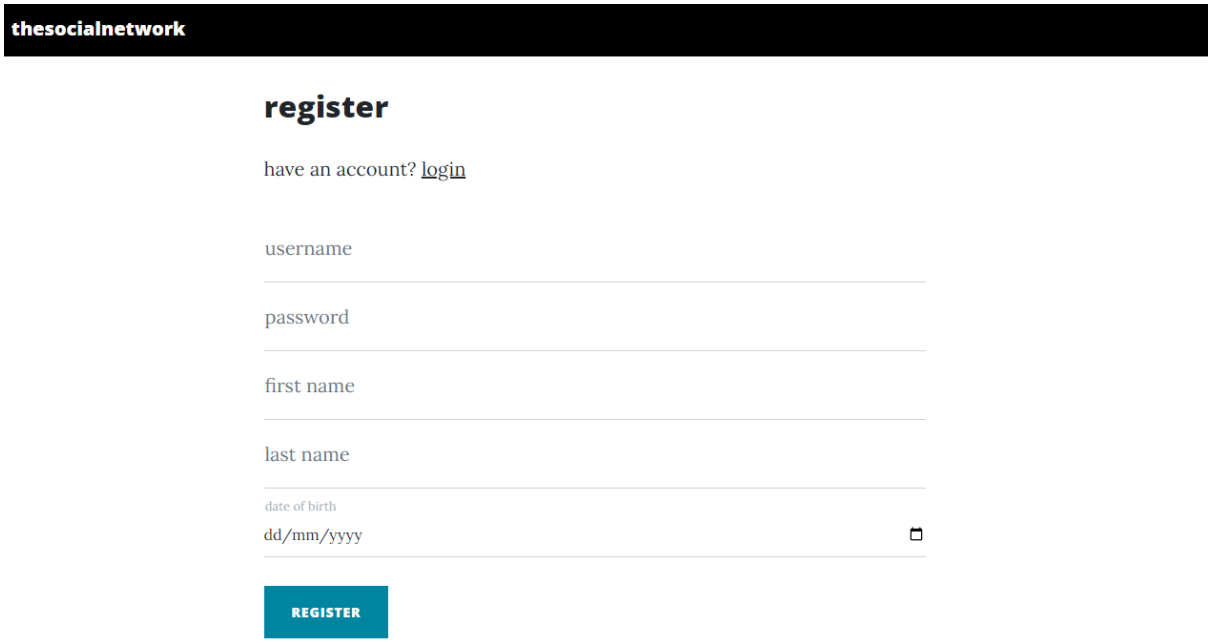
Username	Password	Account Type
user	password	superuser
pika	password	normal
char	password	normal
bulb	password	normal
squirtle	password	normal

**superuser does not have an object in the AppUser model please do not use the account in the website*

Requirements

R1: The application contains the functionality requires

a) Users can create accounts



templates/register.html registration page

```python
# user details model
class AppUser(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE, related_name="profile")
    first_name = models.CharField(max_length=50, blank=True, null=True)
    last_name = models.CharField(max_length=50, blank=True, null=True)
    dob = models.DateField(null=True, blank=True)
    pfp = models.ImageField(upload_to='profile_image', null=True, blank=True)

    def __str__(self):
        return self.user.username
```

models.py AppUser model

This model utilizes the Django User model and has a OneToOne relationship to its 'user' field. User registration details will be saved into this model and the Django User table, the reason for using another table is so we can save additional details like dob (date of birth), pfp (profile image) or any other details that are not included in the Django User model.

```python
class UserForm(forms.ModelForm):
    password = forms.CharField(widget=forms.PasswordInput())
    class Meta:
        model = User
        fields = ('username', 'password')

class UserProfileForm(forms.ModelForm):
    first_name = forms.CharField(required=False, widget=forms.TextInput(attrs={'class':'profile-update'}), label='first name')
    last_name = forms.CharField(required=False, widget=forms.TextInput(attrs={'class':'profile-update'}), label='last name')
    dob = forms.DateField(widget=forms.DateInput(format='%Y-%m-%d',attrs={'class': 'profile-update', 'type': 'date'}), label='date of birth')
    pfp = forms.ImageField(label='profile image', required=False)
    class Meta:
        model = AppUser
        fields = ['first_name', 'last_name', 'dob', 'pfp']
```

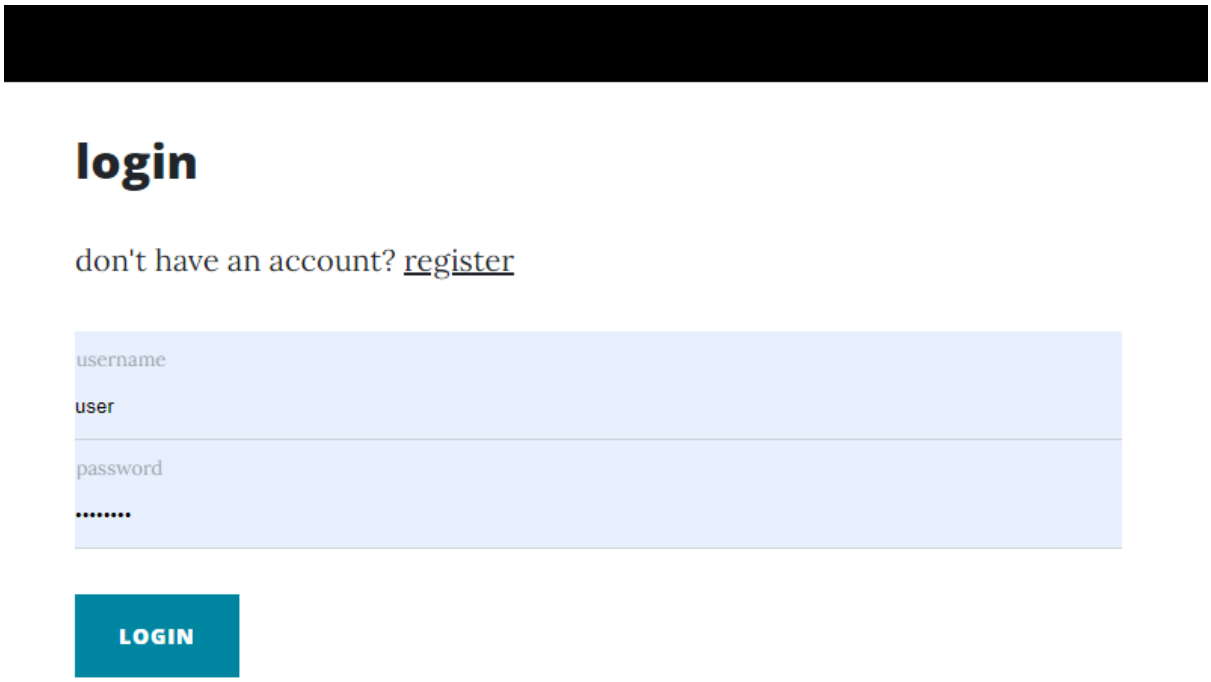
forms.py UserForm & UserProfileForm

```python
def register(request):

    registered = False

    if request.method == 'POST':
        user_form = UserForm(data=request.POST)
        profile_form = UserProfileForm(data=request.POST)

        if user_form.is_valid() and profile_form.is_valid():
            user = user_form.save()
            user.set_password(user.password)
            user.save()
            profile = profile_form.save(commit=False)
            profile.user = user
            profile.save()
            registered = True
        else:
            print(user_form.errors, profile_form.errors)
            messages.error(request, user_form.errors)
            messages.error(request, profile_form.errors)
    else:
        user_form = UserForm()
        profile_form = UserProfileForm()

    return render(request, 'register.html', {'user_form': user_form, 'profile_form': profile_form, 'registered': registered})
```

views.py register view

The register view function is responsible in the rendering of the UserForm & UserProfileForm, also for parsing the POST data back to the database. When the user submits their registration details (POST) it will check if the data is valid, if there are errors it will display an error message alert and will return the user back to the registration page, else it will save the data into both the User & AppUser models. Upon successful registration the ‘registered’ variable will be set to true, and the user will be shown a successful registration page.

b) Users can log in and log out



login.html login page

navbar.html navigation bar when logged in

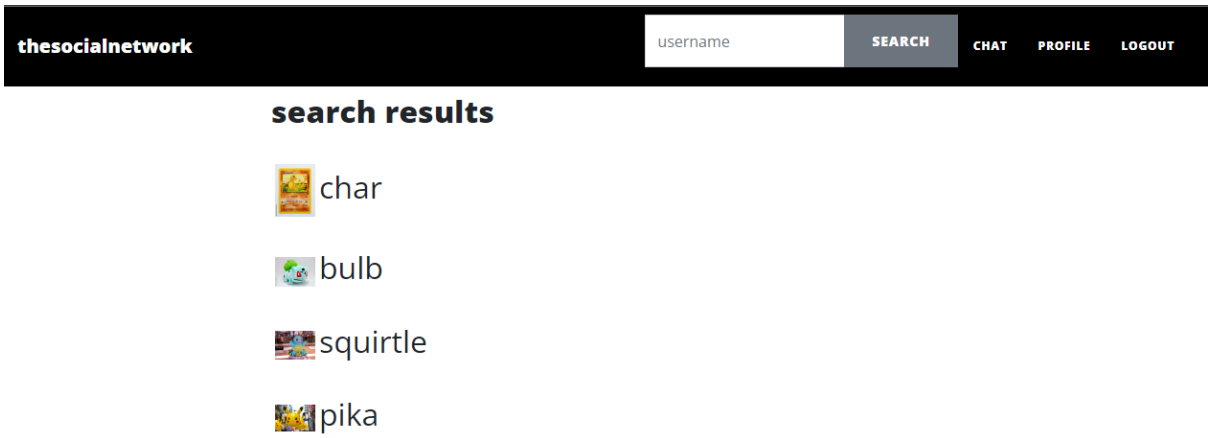
```python
def user_login(request):
    if request.method == 'POST':
        username = request.POST['username']
        password = request.POST['password']
        user = authenticate(username=username, password=password)

        if user:
            # log the user in and go to the homepage
            login(request, user)
            return redirect('home')
        else:
            messages.error(request, "username or password is incorrect")
            return render(request, 'login.html', {})
    else:
        return render(request, 'login.html', {})

@login_required
def user_logout(request):
    # log the user our and return success message
    logout(request)
    messages.success(request, 'you have been logged out')
    return redirect('/')
```

views.py user_login & user_logout

When the user attempts to log in there will be a POST request, the username and password will be fetched, and pass through the authenticate function. If the users' credentials are in the database, it will log them in and redirect them to the homepage (home.html), else it will alert an error message.

The user can then click the 'LOGOUT' button in the navigation bar to log themselves out, an alert will be shown once done and will redirect them to the homepage.

c) Users can search for other users



search.html search page

```python
@login_required
def user_search(request):
    context = {}
    search_result = None
    if request.method == "POST":
        search = request.POST.get('search')
        if search:
            # filter AppUser whose user's username contains the search query (case-insensitive)
            search_result = AppUser.objects.filter(user__username__icontains=search).exclude(user=request.user)
        else:
            # if search bar is blank, retrieve all AppUsers excluding the current user
            search_result = AppUser.objects.all().exclude(user=request.user)

        result_with_pfp = []

        for user in search_result:
            if user.pfp:
                result_with_pfp.append((user.user, user.pfp.url))
            else:
                result_with_pfp.append((user.user, None))

    context = {
        'search_result': result_with_pfp,
    }

    return render(request, "search.html", context)
```

views.py search function

To search, I used the filter function on the AppUser model to filter out the users that matches the input data in the search bar, it will also exclude the current user from the search result. If there is no data in the search bar it will display all users in the AppUser model. The for loop gets the profile images of the filtered results and is returned together with the filtered results for rendering.

```html
<h2 class="post-title">search results</h2>
<p class="post-subtitle">
    {% for user, pfp_url in search_result %}
    <div class="">
        <div class="d-flex align-items-center">
            {% if pfp_url %}
                <img src="{{ pfp_url }}" alt="user profile picture" class="img-fluid p-1" width="50" height="50" />
            {% else %}
                <!-- Display a default image or placeholder for followings with no profile picture -->
                <img src="/static/profile_image/blank_profile_image.png" alt="blank profile picture" class="img-fluid p-1" width="50" height="50" />
            {% endif %}
            <h2 class="post-subtitle" style="font-weight: normal;"><a href="/viewing/{{ user }}/">{{ user }}</a></h2>
        </div>
    </div>
    <br>
    {% endfor %}
</p>
```

search.html

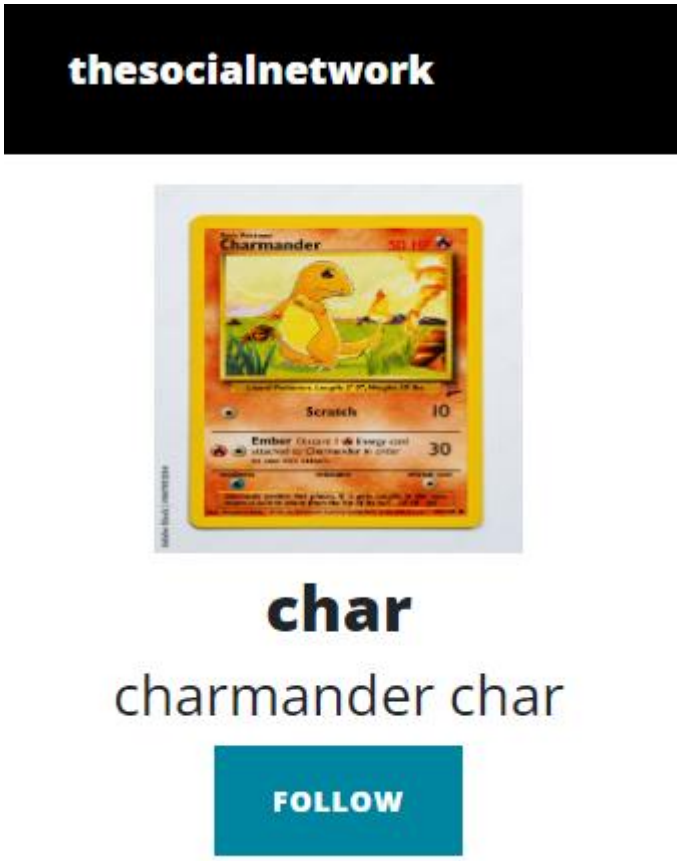
The results are the rendered on this search.html page, if the current index of the loop does not have a profile image it will display the default blank profile image stored in static/profile_image. Their usernames are also linked to their profile pages.

d) Users can add other users as friends

```python
# following/follower model
class Followers(models.Model):
    user = models.ForeignKey(User, related_name='followers', on_delete=models.CASCADE)
    follower = models.ForeignKey(User, related_name='following', on_delete=models.CASCADE)

    def __str__(self):
        return u'user: %s, follower: %s' % (self.user.username, self.follower.username)

```

models.py Followers model

```html
{% if isFollowing %}
<form class="follow" action="{% url 'viewing' username %}" method="POST">
  {% csrf_token %}
  <input type="hidden" name="user" value="{{ username }}">
  <input type="hidden" name="follower" value="{{ user }}">
  <button type="submit" name="button" class="btn btn-secondary text-uppercase">Following<i class="fi-check"></i></button>
</form>
{% else %}
<form class="follow" action="{% url 'viewing' username %}" method="POST">
  {% csrf_token %}
  <input type="hidden" name="user" value="{{ username }}">
  <input type="hidden" name="follower" value="{{ user }}">
  <button type="submit" name="button" class="btn btn-primary text-uppercase">Follow<i class="fi-plus"></i></button>
</form>
{% endif %}
```



viewing.html follow button

```python
@login_required
def viewing(request, username):
    context = {}
    context['username'] = username
    # convert to object to be read when filtering as it is a foreign key in the models
    username = User.objects.get(username=username)
    # get viewing user's details
    profile = AppUser.objects.get(user=username)
    context['profile'] = profile
    # redirect back to own profile if own username is clicked
    if username == request.user:
        return redirect('profile')
    else:
        # when follow button is pressed
        if request.method == 'POST':
            if Followers.objects.filter(user=username, follower=request.user):
                Followers.objects.filter(user=username, follower=request.user).delete()
                isFollowing = False
            else:
                Followers.objects.create(user=username, follower=request.user)
                isFollowing = True

            context['isFollowing'] = isFollowing
```

views.py viewing function

```python
    path('viewing/<str:username>/', views.viewing, name='viewing'),
```

urls.py viewing url for other users

Users may follow each other by visiting other users' profile pages and pressing the follow button. Once following the page will refresh and render the other users's profile page again to display that they have been followed by the current user, users may also unfollow the user by clicking the follow (will be displayed as 'following') button again. Posts of your followed users will then be displayed on the homepage together with your posts ordered by id, descending (when it was created).

e) Users can chat in realtime with friends

```python
# chatrooms model
class Chatroom(models.Model):
    chatroom = models.CharField(max_length=255, unique=True, blank=False)
    user = models.ManyToManyField(User, blank=True)
    message = models.TextField()
    timestamp = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.chatroom

# chatroom messages
class ChatroomMessages(models.Model):
    id = models.AutoField(primary_key=True)
    chatroom = models.ForeignKey(Chatroom, related_name='messages', on_delete=models.CASCADE)
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    message = models.TextField()
    timestamp = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.message
```

models.py Chatroom & ChatRoomMessages models

create chatroom (no spaces)

CREATE

chatrooms

testchatroom

chats.html chatrooms page

Chatroom: testchatroom

you are logged in as **user**

bulb: hello :D
char: char char char
bulb: how are you char?
char: im gud hbu??
bulb: good too bulbaaa
pika: hi is anyone there?
char: hi pikachu whats up?
pika: nothing much hbu???
char: just at the volcanoes chilling hehe
pika: hahaha xD

type your message here...

SEND

chatroom.html chatroom page

```python
@login_required
def chats(request):
    context = {}
    if request.method == "POST":
        chatroomForm = ChatroomForm(request.POST)
        if chatroomForm.is_valid():
            chatroomForm.save()
            chatroom = request.POST['chatroom']
            messages.success(request, 'Chatroom created successfully')
            return redirect('/chats/' + chatroom)
        else:
            messages.error(request, 'There was an error when creating the chatroom')
            return redirect('chats')
    else:
        chatroom = Chatroom.objects.all()
        # show create chatroom form
        chatroomForm = ChatroomForm()
        context['chatroom'] = chatroom
        context['chatroomForm'] = chatroomForm

    return render(request, 'chats.html', context)

@login_required
def chatroom(request, chatroom):
    context = {}
    if request.method == "GET":
        # retrieve and display chat history for the current chatroom
        chat_history = ChatroomMessages.objects.filter(chatroom__chatroom=chatroom).order_by('timestamp')
        context['chatroom'] = chatroom
        context['chat_history'] = chat_history
        return render(request, 'chatroom.html', context)
```

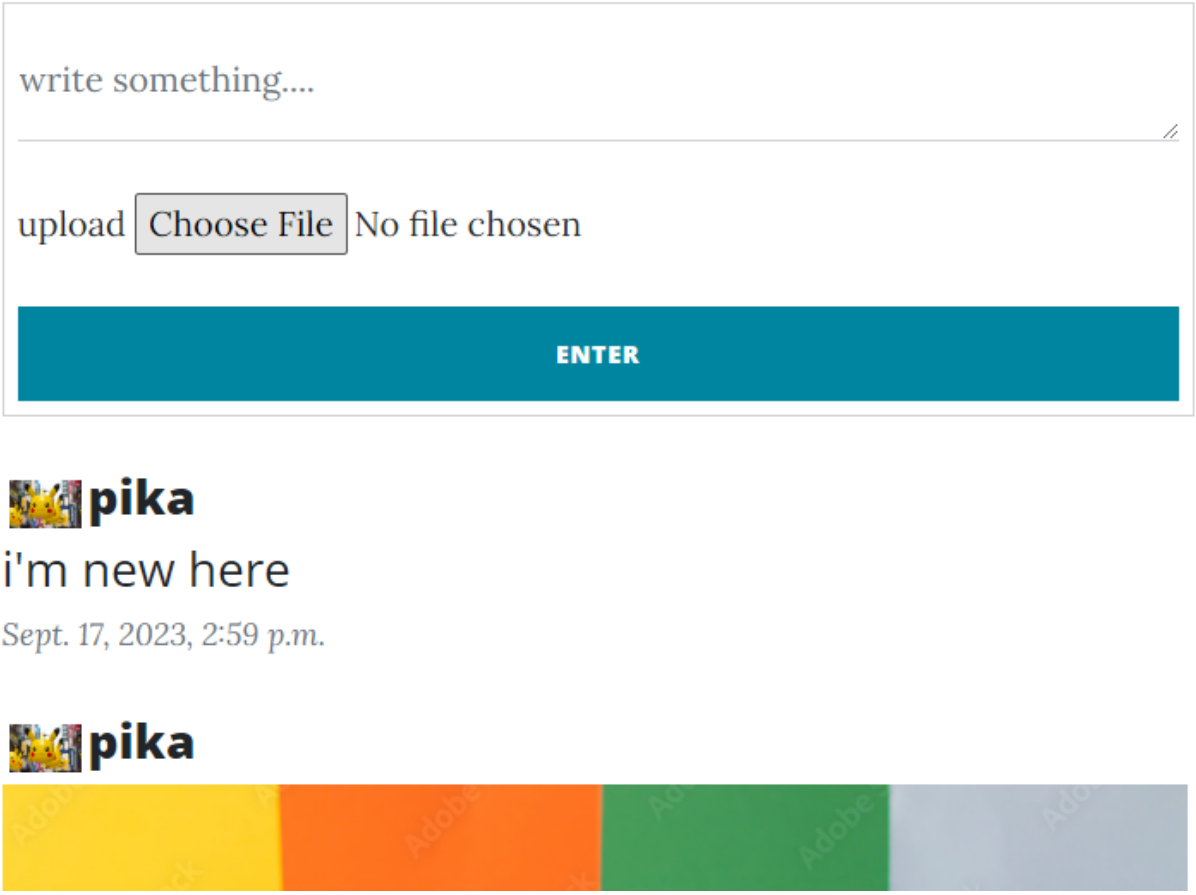
views.py chats & chatroom function

```python
    @sync_to_async
    def get_chat_history(self):
        return ChatroomMessages.objects.filter(chatroom__chatroom=self.chatroom).order_by('timestamp')

    async def send_chat_history(self):
        chat_history = await self.get_chat_history()
        for message in chat_history:
            await self.send(text_data=json.dumps({
                'type': 'chat.message',
                'user': message.user.username,
                'message': message.message,
            }))

    @sync_to_async
    def save_chat_message(self, user, chatroom, message):
        return ChatroomMessages.objects.create(user=user, chatroom=chatroom, message=message)

    async def receive(self, text_data):
        user = self.scope["user"]
        message = text_data

        # extract the message content without additional processing
        message_content = message.strip().split(': ', 1)[1]  # Split and take the part after the first ": "
        # remove the trailing "}" from message_content if it exists
        message_content = message_content.rstrip('"}')

        chatroom = await sync_to_async(Chatroom.objects.get)(chatroom=self.chatroom)

        try:
            # Create and save the chat message (wrapped in sync_to_async)
            await self.save_chat_message(user, chatroom, message_content)

            # Send the message to the room group
            await self.channel_layer.group_send(
                self.room_group_name,
                {
                    'type': 'chat.message',
                    'user': user.username,
                    'message': message_content,
                }
            )
        except Exception as e:
            # Handle exceptions, log them, or send an error message to the client
            error_message = {"error": str(e)}
            await self.send(text_data=json.dumps(error_message))
```

consumers.py get_chat_history & save_chat_message function

Users may enter the public chatrooms through the 'CHAT' button in the navigation bar. They will be displayed the chatrooms that have been created by other users or they may create a new one at the top. I have amended the consumers.py to save the history of chats through the save_chat_message function, each time a message is sent it will be saved into the ChatroomMessages model together with the current chatroom name. If a user enters a chatroom with history the function send_chat_history will run and display in the textarea, new messages sent will be displayed below the chat history.

f) Users can add status updates to their home page



home.html homepage when logged in

```python
# posts model
class Posts(models.Model):
    id = models.AutoField(primary_key=True)
    user = models.ForeignKey(User, on_delete=models.CASCADE, related_name="user")
    text = models.CharField(max_length=10000)
    # stores the image if there is any
    image = models.ImageField(upload_to='post_image', null=True, blank=True)
    timestamp = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.user.username
```

models.py posts model

```python
def home(request, *args, **kwargs):
    context = {}
    user = request.user
    # check if request method is post and that the user is authenticated
    if request.method == "POST" and user.is_authenticated:
        # get the user post form
        post_form = PostsForm(request.POST, request.FILES)
        context['post_form'] = post_form

        # check if the post_form is valid
        if post_form.is_valid():
            # set the user field of the new post
            post_form.instance.user = user
            # save the form, including the image, to the database
            post_form.save()
            messages.success(request, 'Successfully uploaded a new post')
            return HttpResponseRedirect("/")
        else:
            messages.success(request, post_form.errors)
            return HttpResponseRedirect("/")
```

view.py home POST function

```python
class PostsForm(forms.ModelForm):
    text = forms.CharField(required=True, widget=forms.TextInput(attrs={'class':'form-control',}), label='')
    image = forms.ImageField(required=False, label='upload', widget=forms.FileInput(attrs={'class': 'form-control-file','id': 'image',}))

    class Meta:
        model = Posts
        fields = ['text', 'image']
```

forms.py PostsForm

Users can add status updates in the home page, including images (will be saved under media/post_image). The home function will post data into the Posts model if the form in the home page is valid, it will save the text, image (url), user data and the timestamp of when the post was created.

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

pika

pika pika

EDIT PROFILE

pika

first name: pika

last name: pika

date of birth: 07/07/2000

profile image: Currently:
profile_image/1000_F_299395589_CKn80lOnyTu5pQ0ww45wHi6rkt07oogr_gzdivYe.jpg
☐ Clear
Change: Choose File No file chosen

UPDATE CANCEL

profile.html edit profile button *edit_profile.html editing page*

```
@login_required
def edit_profile(request):
    if request.method == 'POST':
        profile_form = UserProfileForm(request.POST, request.FILES, instance=request.user.profile)

        if profile_form.is_valid():
            profile_form.save()
            # return success message
            messages.success(request, 'Your profile has been updated successfully')
            return redirect('profile')
        else:
            # else return error message
            messages.error(request, 'There was an error when updating your profile')
            return redirect('edit_profile')
    else:
        profile_form = UserProfileForm(instance=request.user.profile)
        context = {
            'profile_form': profile_form,
        }
    return render(request, 'edit_profile.html', context)
```

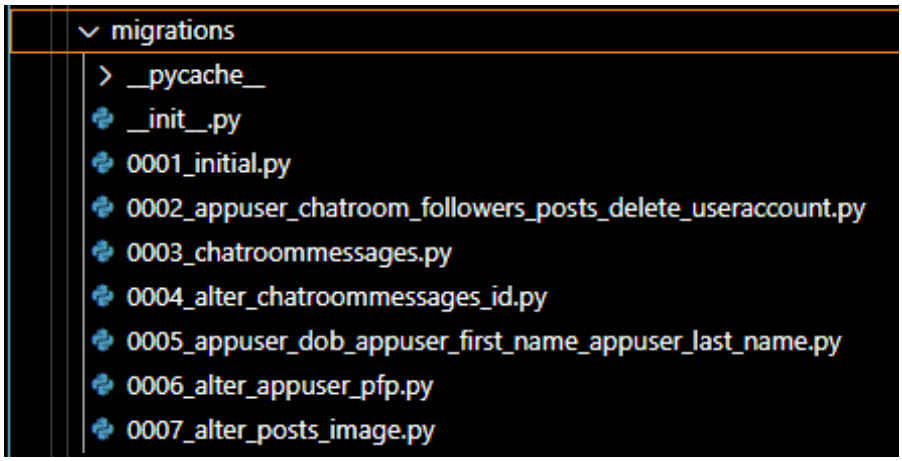
views.py edit_profile function

```
class UserProfileForm(forms.ModelForm):
    first_name = forms.CharField(required=False, widget=forms.TextInput(attrs={'class':'profile-update'}), label='first name')
    last_name = forms.CharField(required=False, widget=forms.TextInput(attrs={'class':'profile-update'}), label='last name')
    dob = forms.DateField(widget=forms.DateInput(format='%Y-%m-%d',attrs={'class': 'profile-update', 'type': 'date'}), label='date of birth')
    pfp = forms.ImageField(label='profile image', required=False)
    class Meta:
        model = AppUser
        fields = ['first_name', 'last_name', 'dob', 'pfp']
```

forms.py UserProfileForm

On the profile page, users may edit their profile details as well as add and remove their profile image (will be saved in media/profile_image). The data is retrieved from the UserProfileForm and updated into the database. They can then see their profile image beside their username in the homepage, else a default blank profile image will be displayed as their profile image.

h) correct use of models and migrations



migrations folder

thesocialnetwork > the_app > models.py > ...

```python
from django.db import models
from django.contrib.auth.models import User

# user details model
class AppUser(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE, related_name="profile")
    first_name = models.CharField(max_length=50, blank=True, null=True)
    last_name = models.CharField(max_length=50, blank=True, null=True)
    dob = models.DateField(null=True, blank=True)
    pfp = models.ImageField(upload_to='profile_image', null=True, blank=True)

    def __str__(self):
        return self.user.username

# posts model
class Posts(models.Model):
    id = models.AutoField(primary_key=True)
    user = models.ForeignKey(User, on_delete=models.CASCADE, related_name="user")
    text = models.CharField(max_length=10000)
    # stores the image if there is any
    image = models.ImageField(upload_to='post_image', null=True, blank=True)
    timestamp = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.user.username

# following/follower model
class Followers(models.Model):
    user = models.ForeignKey(User, related_name='followers', on_delete=models.CASCADE)
    follower = models.ForeignKey(User, related_name='following', on_delete=models.CASCADE)

    def __str__(self):
        return u'user: %s, follower: %s' % (self.user.username, self.follower.username)

# chatrooms model
class Chatroom(models.Model):
    chatroom = models.CharField(max_length=255, unique=True, blank=False)
    user = models.ManyToManyField(User, blank=True)
    message = models.TextField()
    timestamp = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.chatroom

# chatroom messages
class ChatroomMessages(models.Model):
    id = models.AutoField(primary_key=True)
    chatroom = models.ForeignKey(Chatroom, related_name='messages', on_delete=models.CASCADE)
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    message = models.TextField()
    timestamp = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.message
```

models.py

i) correct use of form, validators and serialisation

```python
from django import forms
from .models import *
from django.forms import ModelForm
from django.contrib.auth.models import User

class UserForm(forms.ModelForm):
    password = forms.CharField(widget=forms.PasswordInput())
    class Meta:
        model = User
        fields = ('username', 'password')

class UserProfileForm(forms.ModelForm):
    first_name = forms.CharField(required=False, widget=forms.TextInput(attrs={'class':'profile-update'}), label='first name')
    last_name = forms.CharField(required=False, widget=forms.TextInput(attrs={'class':'profile-update'}), label='last name')
    dob = forms.DateField(widget=forms.DateInput(format='%Y-%m-%d',attrs={'class': 'profile-update', 'type': 'date'}), label='date of birth')
    pfp = forms.ImageField(label='profile image', required=False)
    class Meta:
        model = AppUser
        fields = ['first_name', 'last_name', 'dob', 'pfp']

class PostsForm(forms.ModelForm):
    text = forms.CharField(required=True, widget=forms.TextInput(attrs={'class':'form-control',}), label='')
    image = forms.ImageField(required=False, label='upload', widget=forms.FileInput(attrs={'class': 'form-control-file','id': 'image',}))

    class Meta:
        model = Posts
        fields = ['text', 'image']

class ChatroomForm(forms.ModelForm):
    chatroom = forms.CharField(required=True,)
    class Meta:
        model = Chatroom
        fields = ['chatroom']
```

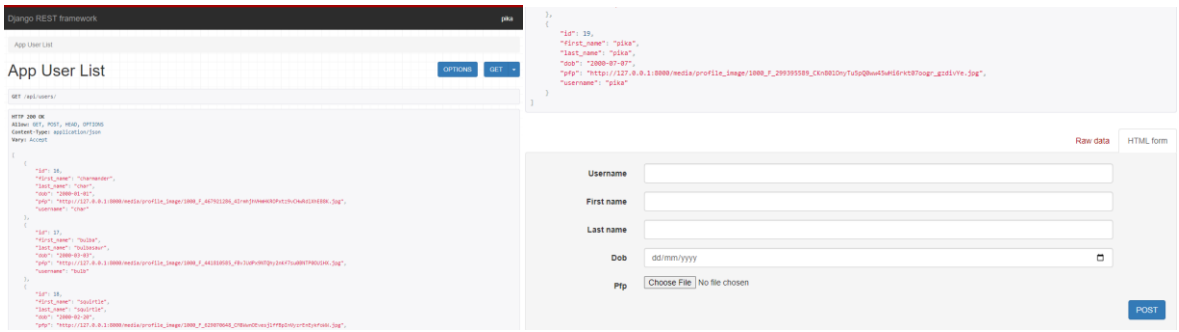
forms.py

```python

class AppUserSerializer(serializers.ModelSerializer):
    # Define a writable username field in the serializer
    username = serializers.CharField(write_only=True)

    class Meta:
        model = AppUser
        fields = ['id', 'username', 'first_name', 'last_name', 'dob', 'pfp']

    def to_representation(self, instance):
        data = super().to_representation(instance)

        # Check if the instance has a user associated with it
        if instance.user:
            data['username'] = instance.user.username

        return data

    def create(self, validated_data):
        # Extract the username from the validated data
        username = validated_data.pop('username', None)

        # Create a new user or get an existing user by username
        user, created = User.objects.get_or_create(username=username)

        # Create the AppUser instance with the associated user
        app_user = AppUser.objects.create(user=user, **validated_data)
        return app_user

class PostsSerializer(serializers.ModelSerializer):
    # Add a read-only field for username
    username = serializers.ReadOnlyField(source='user.username')

    class Meta:
        model = Posts
        fields = ['id', 'username', 'text', 'image', 'timestamp']

    def create(self, validated_data):
        # Access the user from the request's context
        user = self.context['request'].user

        # Add the user to the validated data before creating the post
        validated_data['user'] = user

        post = Posts.objects.create(**validated_data)
        return post
```

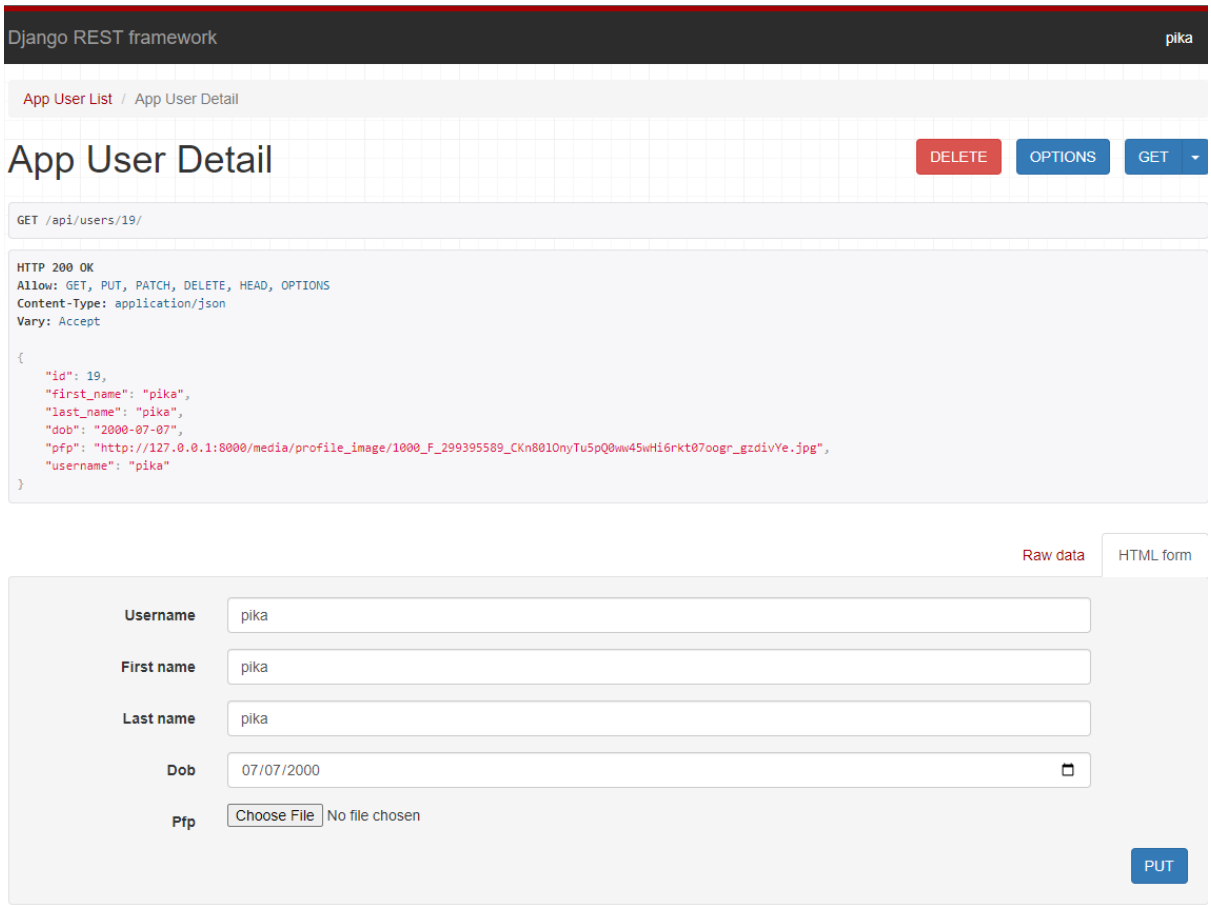
serializers.py

Serializers currently only being used for APIs in Django Rest Framework, could have been used to pull data and rendered as JSON formats to avoid issues when rendering data as the user fields in all models are foreign keys related to the Django user table.

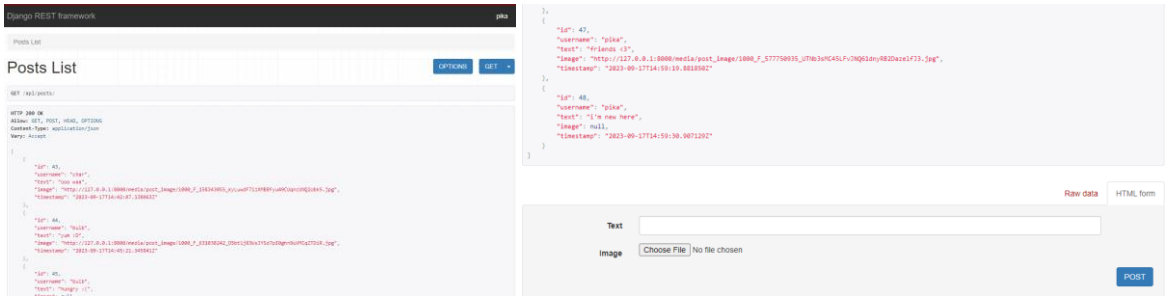
j) correct use of django-rest-framework



api/users url AppUser List



api/users/<int:pk> url AppUser details



api/posts url Posts List

Django REST framework pika

Posts List / Posts Detail

Posts Detail

DELETE OPTIONS GET

```
GET /api/posts/43/
```

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

{
    "id": 43,
    "username": "char",
    "text": "ooo waa",
    "image": "http://127.0.0.1:8000/media/post_image/1000_F_158343055_kyLuwdF71zXMBBFyu49CUqncUNQ1Ubk5.jpg",
    "timestamp": "2023-09-17T14:42:07.138063Z"
}
```

Raw data HTML form

Text: ooo waa

Image: Choose File No file chosen

PUT

api/posts/<int:pk> url Posts details

```
class AppUserList(generics.ListCreateAPIView):
    queryset = AppUser.objects.all()
    serializer_class = AppUserSerializer

class AppUserDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = AppUser.objects.all()
    serializer_class = AppUserSerializer

class PostsList(generics.ListCreateAPIView):
    queryset = Posts.objects.all()
    serializer_class = PostsSerializer

class PostsDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Posts.objects.all()
    serializer_class = PostsSerializer
```

views.py API views

```python
    # api for users
    # List and create users
    path('api/users/', views.AppUserList.as_view(), name='users-list'),
    # Retrieve, update, and delete a user by their ID
    path('api/users/<int:pk>/', views.AppUserDetail.as_view(), name='users-detail'),

    # api for posts
    # List all posts and create a new post
    path('api/posts/', views.PostsList.as_view(), name='posts-list'),
    # Retrieve, update, and delete a post by its ID
    path('api/posts/<int:pk>/', views.PostsDetail.as_view(), name='posts-detail'),
```

urls.py API urls

Data is pulled through the serializers in the views.py and the API Schema allows the users to access the data through JSON format. Users and posts from the AppUser and Posts models can be created or edited/deleted through the API by accessing by its primary key (id).

k) correct use of URL routing

```python
urlpatterns = [
    path('', views.home, name='home'),
    path('login/', views.user_login, name='login'),
    # for redirecting @login_required functions
    path('accounts/login/', views.user_login, name='login'),
    path('register/', views.register, name='register'),
    path('logout/', views.user_logout, name='logout'),
    path('search/', views.user_search, name='search'),
    path('profile/', views.profile, name='profile'),
    path('edit_profile/', views.edit_profile, name='edit_profile'),
    path('viewing/<str:username>/', views.viewing, name='viewing'),
    path('chats/', views.chats, name='chats'),
    path('chats/<str:chatroom>', views.chatroom, name='chatroom'),
    # path('chats/deleteChatroom/<str:chatroom>', views.deleteChatroom, name='deleteChatroom'),

    path('apischema/', get_schema_view(
        title="thesocialnetwork REST API",
        description="API for interacting with data", version="1.0.0"
    ), name='openapi-schema'),
    path('swaggerdocs/', TemplateView.as_view(
        template_name='swagger-docs.html',
        extra_context={'schema_url':'openapi-schema'}
    ), name='swagger-ui'),

    # api for users
    # List and create users
    path('api/users/', views.AppUserList.as_view(), name='users-list'),
    # Retrieve, update, and delete a user by their ID
    path('api/users/<int:pk>/', views.AppUserDetail.as_view(), name='users-detail'),

    # api for posts
    # List all posts and create a new post
    path('api/posts/', views.PostsList.as_view(), name='posts-list'),
    # Retrieve, update, and delete a post by its ID
    path('api/posts/<int:pk>/', views.PostsDetail.as_view(), name='posts-detail'),
]
```

urls.py

The ‘accounts/login/’ is created for those pages that requires the user to login to access, it will redirect users to the login page if they are not logged in

```python
@login_required
def viewing(request, username):
    context = {}
    context['username'] = username
    # convert to object to be read when filtering as it is a foreign key in the models
    username = User.objects.get(username=username)
    # get viewing user's details
    profile = AppUser.objects.get(user=username)
    context['profile'] = profile
    # redirect back to own profile if own username is clicked
    if username == request.user:
        return redirect('profile')
```

views.py viewing function

The viewing function has a redirect to the profile page of the current user if they click their own username in the home, search, or following/followers.

l) appropriate use of unit testing

```python
import factory
from django.contrib.auth.models import User
from .models import AppUser, Posts

class UserFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = User

    username = factory.Faker('user_name')

class AppUserFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = AppUser

    user = factory.SubFactory(UserFactory)
    first_name = factory.Faker('first_name')
    last_name = factory.Faker('last_name')
    dob = factory.Faker('date_of_birth')
    pfp = factory.django.ImageField(filename='test_image.jpg')

class PostsFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = Posts

    user = factory.SubFactory(UserFactory)
    text = factory.Faker('text')
    image = factory.django.ImageField(filename='post_image.jpg')
    timestamp = factory.Faker('date_time_this_century')

```

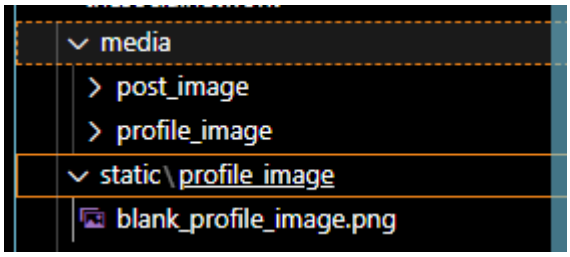
models_factories.py

```python
class AppUserTests(TestCase):
    def test_app_user_creation(self):
        app_user = AppUserFactory()
        self.assertTrue(isinstance(app_user, AppUser))
        self.assertTrue(isinstance(app_user.user, User))
        self.assertEqual(AppUser.objects.count(), 1)

class PostsTests(TestCase):
    def test_post_creation(self):
        post = PostsFactory()
        self.assertTrue(isinstance(post, Posts))
        self.assertTrue(isinstance(post.user, User))
        self.assertEqual(Posts.objects.count(), 1)

class AppUserUpdateTestCase(TestCase):
    def setUp(self):
        self.user = User.objects.create(username='testuser', password='testpassword')
        self.app_user = AppUser.objects.create(user=self.user)

    def test_update_app_user_data(self):
        client = APIClient()
        client.login(username='testuser', password='testpassword')

        # define the updated data
        updated_data = {
            'first_name': 'Updated First Name',
            'last_name': 'Updated Last Name',
            'dob': '1990-01-01',
            # 'pfp': '',
        }

        # make a PATCH request to update the user data
        response = client.patch(f'/api/users/{self.app_user.pk}/', updated_data, format='json')

        # check if the update was successful (status code 200)
        self.assertEqual(response.status_code, status.HTTP_200_OK)

        # refresh the app_user instance to get the latest data from the database
        self.app_user.refresh_from_db()

        # check if the user data has been updated
        self.assertEqual(self.app_user.first_name, 'Updated First Name')
        self.assertEqual(self.app_user.last_name, 'Updated Last Name')
        self.assertEqual(str(self.app_user.dob), '1990-01-01')  # Ensure it's a string in the expected format
```

tests.py

Tests and model factories for unit testing, testing of user & post creation, and updating user data.

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



media & static folders

Media and static folders, images are stored in post_image or profile_image and the static folder stores the default blank profile image if the user does not have a profile image set.