

Build a social network (CM3035)

Introduction

For the final project of Advance Web Development (CM3035) I have completed a social media app called Social. Social is a social media much like that of Instagram. The purpose of the Social is to create a platform where users can share interesting artwork or images they made with other like minded people online.

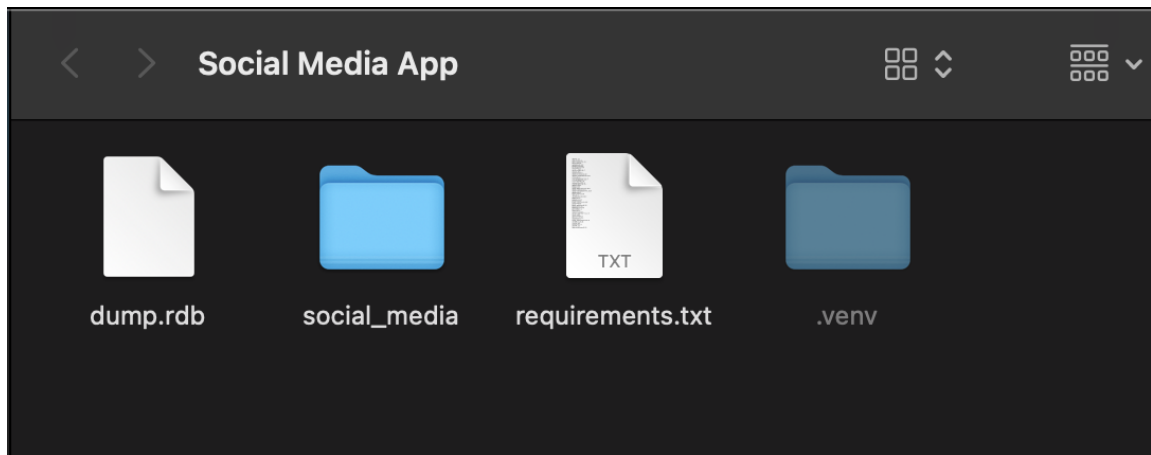
Several features of Social include:

- Posting images with descriptions for followers to view
- Follow and unfollow users on the platform
- Live chatting with users who are followed with stored chat history
- Personal page to display one's images and customisable biography section
- Creation of accounts
- Discovery section to 'discover' other users
- API interface where users can use to retrieve/post specific information

Overall, these features allow users to have a rich and engaging social experience on the platform and it meets the overall requirements of the coursework.

Running the application

1. First, unzipp the file that was sumbmitted along with this report, the file should contain the files shown in the image below.



2. Change directory to the folder that was unzipped within the terminal and activate the .venv file that is stored within the folder.

```
$ cd Social Media App  
$ source .venv/bin/activate
```

3. Run redis

```
$ redis-server
```

4. Initialize the server

```
$ cd social_media  
$ python manage.py runserver
```

5. The server should be initialized and running properly.

User Interface

This section will provide a comprehensive overview of the user interface (UI) of Social. I will be providing detailed description of the Social's UI and explanation of the design and implementation choices for each page currently available on Social.

Navigation Bar

The navigation bar is used to help users go to different pages on Social. It is clean and simple, making it easy for users to navigate the website. It contains different

links according to the state of the user as shown in the images below.



Image of Navigation bar before user logged in



Image of Navigation bar after user logged in

Home Page

The home page is the page where users will spend most of their time on Social. Its state changes depending on whether a user is logged in or not.

When user is not logged in, it will display a page which prompts user to login or signup for a Social account. The image below is a screenshot of the home page.



Social

Welcome to Social, the largest social webspace in the world to share images with your friends!

Login

Sign Up

Image of Home Page without any user logged in

When user is logged in, it displays a feed of images posted by users who the current user follows. Each image in the feed is accompanied by a brief caption, the username of the user who posted it and also the date of posting the image. Here users will also be able to post images and enter a short caption to their post. The image below is a screenshot of the home page after logging in.

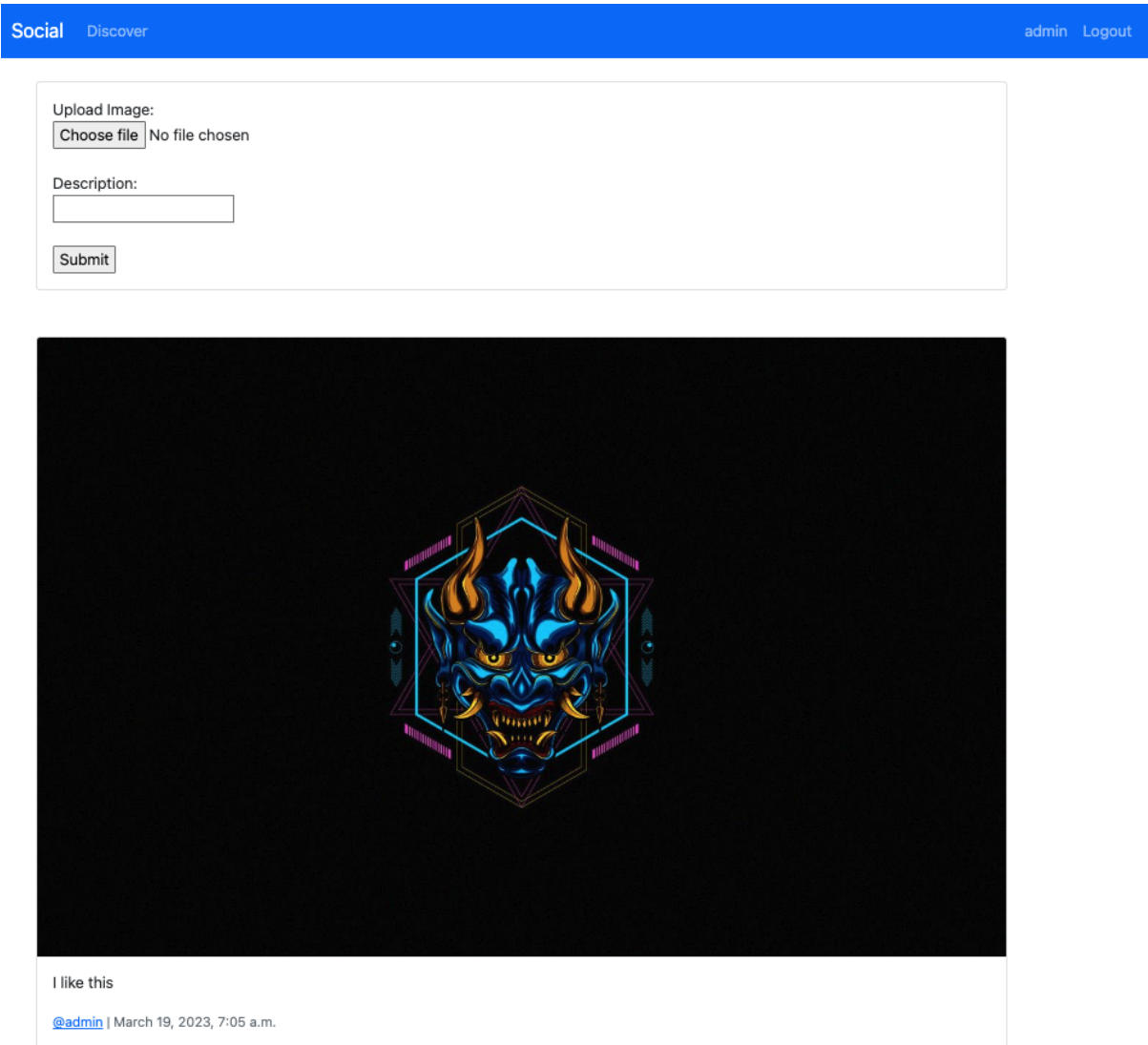


Image of Home Page with the user 'admin' logged in

Login Page

The login page is where users are able to login to their accounts. It is a simple page where users can enter their login credentials. The image below is a screenshot of the login page.



Social Login Sign Up

Login

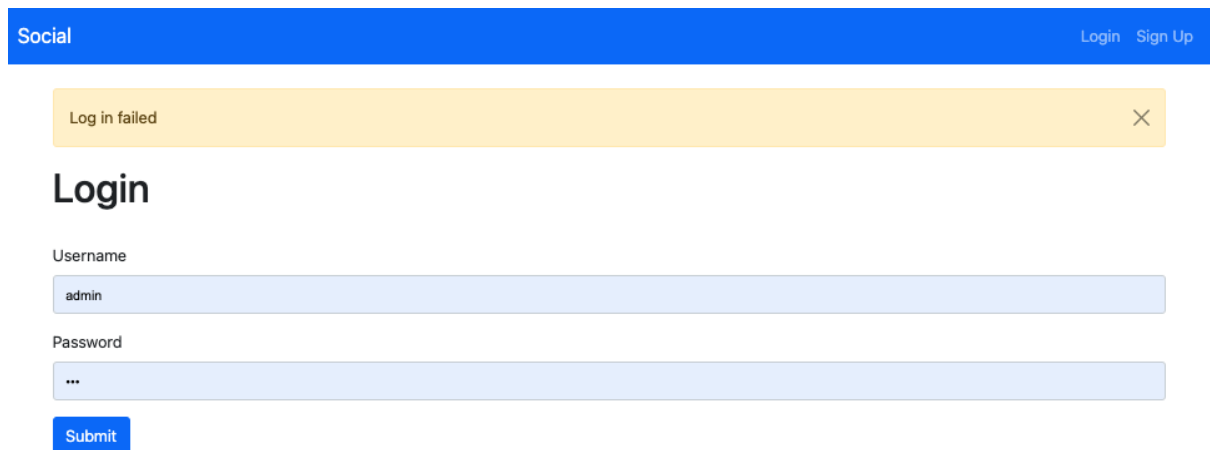
Username

Password

Submit

Image of login page

The login page also checks whether the login credentials of the user is valid or not. When login fails, it will display a notice of failure at the top. This is accomplished using from Django messages framework.



Social Login Sign Up

Log in failed

Login

Username

Password

Submit

Image of login failure when wrong credentials are entered

If login is successful, users will be redirected to the homepage.

Sign Up Page

The sign up page is where the new users are able to create a new account. Users have to create an account before they can use any existing features of Social.

Sign up

First Name

Last Name

Username

Email

Password

Confirm your password

Image of the Sign Up page

Each entry is validated using both HTML5 and Javascript to ensure that all entries are filled up properly. Checks are also carried out with the database to ensure that duplicate usernames are not being created. The image below shows an error message that will be sent to the user if their password is not the retyped properly.

Social

127.0.0.1:8000 says
Passwords do not match

Login Sign Up

Sign up

First Name

Password

A stylized illustration of a person with a speech bubble, surrounded by various colorful geometric shapes like circles, triangles, and a star. The person is wearing a blue shirt and green pants, and is holding a blue bag.

Image Error message if entry is invalid

Profile Page

The profile page is where users can view their own images. The profile page displays user's images and also allow them to edit their profile.

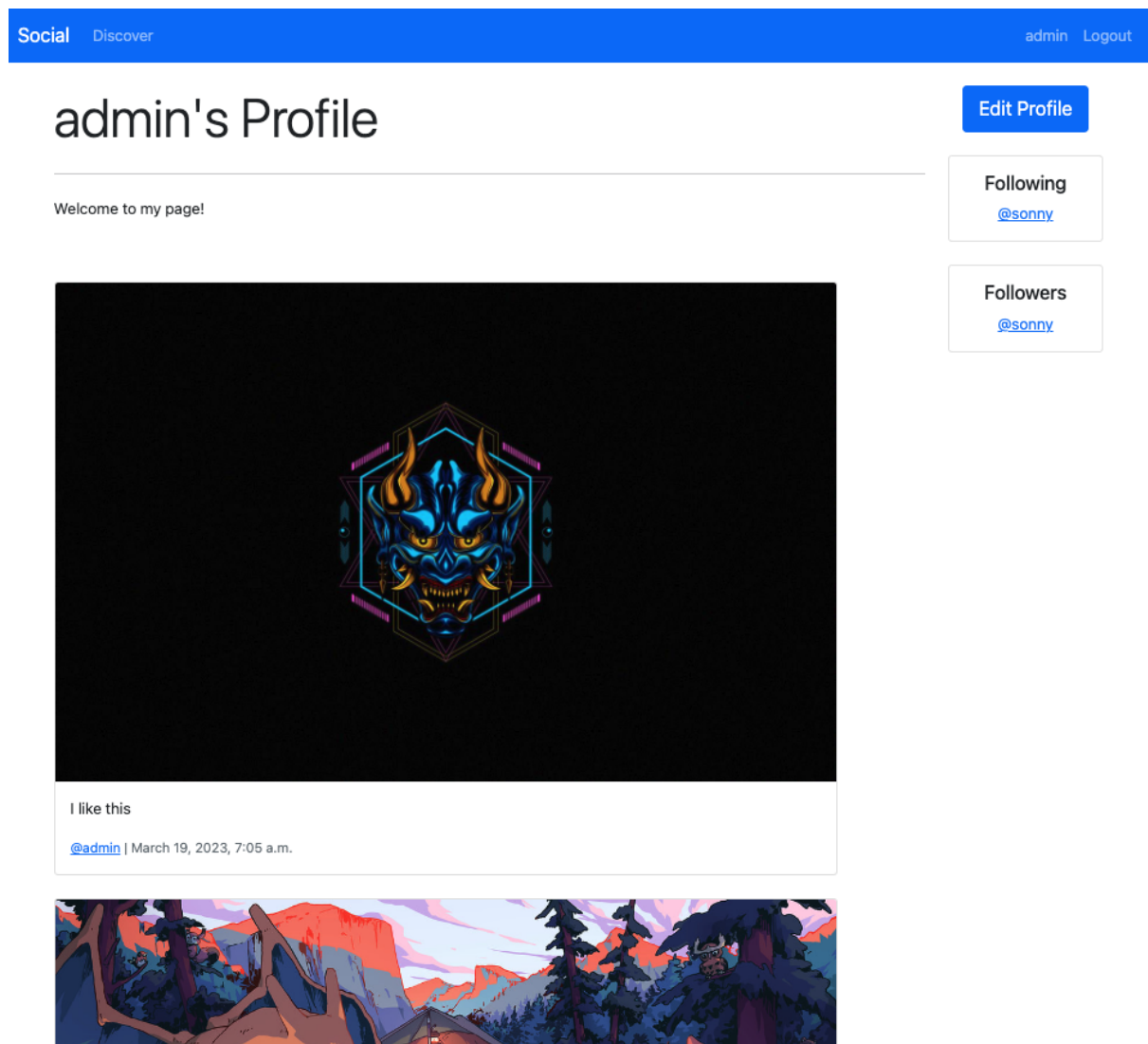


Image of Admin's Profile Page

By clicking on the edit profile button, a pop-up will appear for users to edit the introduction of their profile. The image below shows the pop-up after clicking on the edit profile button.

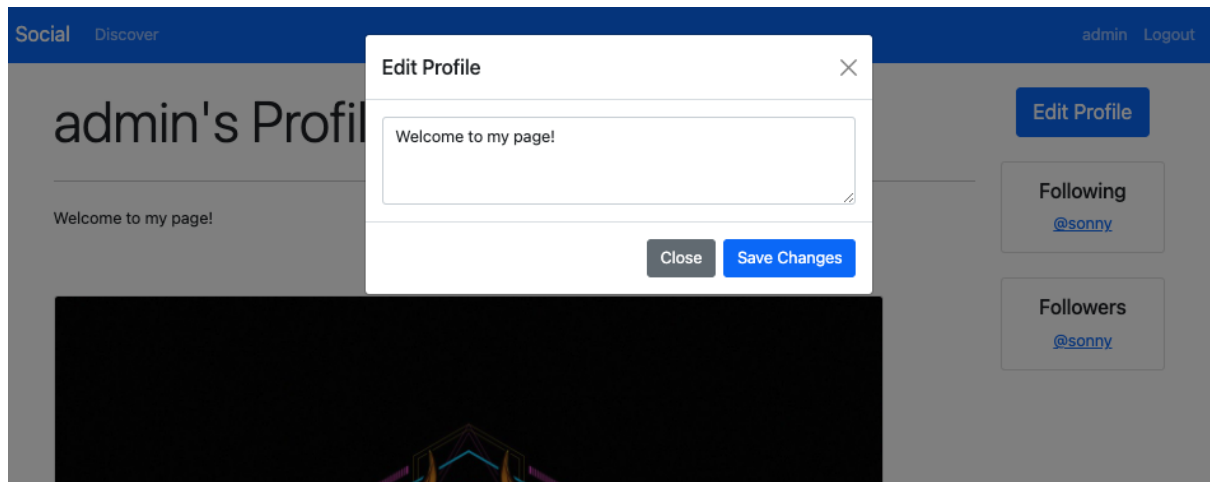


Image of edit profile function at Admin's Profile Page

Unfortunately, due to time constraint of the project, I was unable to implement more features for the edit profile function, ideally it should be able to edit password, profile image and also other details.

Users can also visit other users' profile page and follow or chat with other users. In the image below admin have visited sonny's page and is given the options to unfollow sonny and also chat with him.

sonny's Profile



COYS!

[@sonny](#) | Feb. 28, 2023, 2:14 p.m.

[Chat](#)

Following

[@admin](#)

Followers

[@admin](#)

Unfollow [@sonny](#)

Image of Sonny's Profile Page

Chat Page

The chat page is where the users are able to do live chat with each other. This is accomplished using WebSocket through the use of Django Channels as taught in week 12. The image below is a screenshot of admin's chat with sonny.

sonny

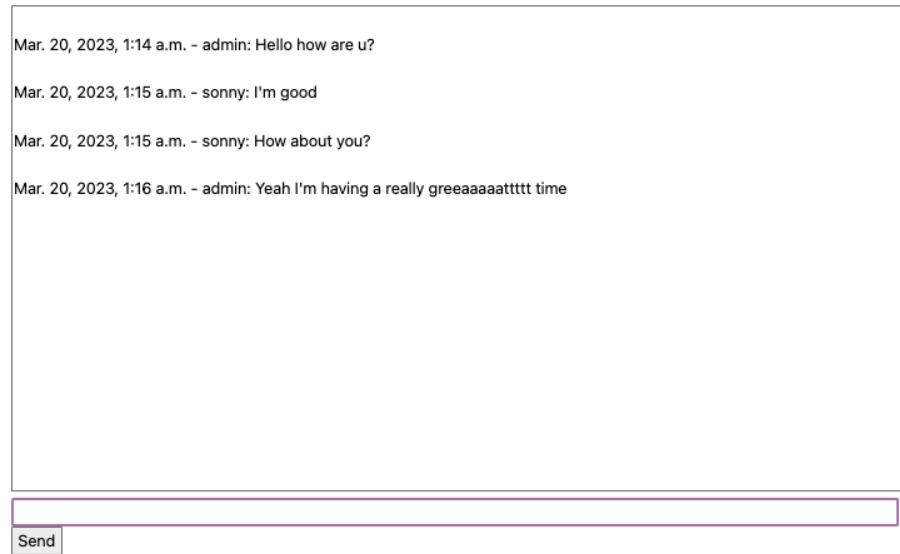


Image of admin's chat with with sonny

From the image above the chat interface prominently displays the chat recipient at the top of the page. Text on the page is dynamically updated during the conversation and chat messages are stored in a database and can be accessed by the user each time they access the chat page.

Discover Page

The Discover Page is a page used to discover different users on the platform. User can visit other user's profile in the discover page.

Discover

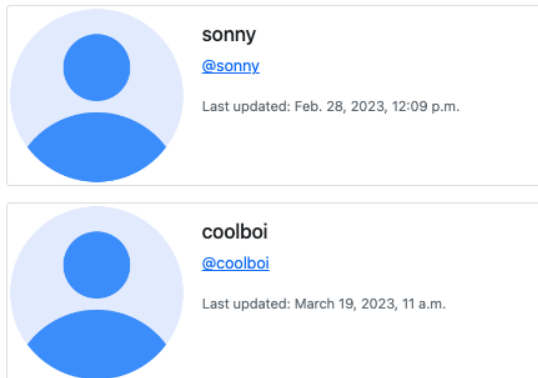


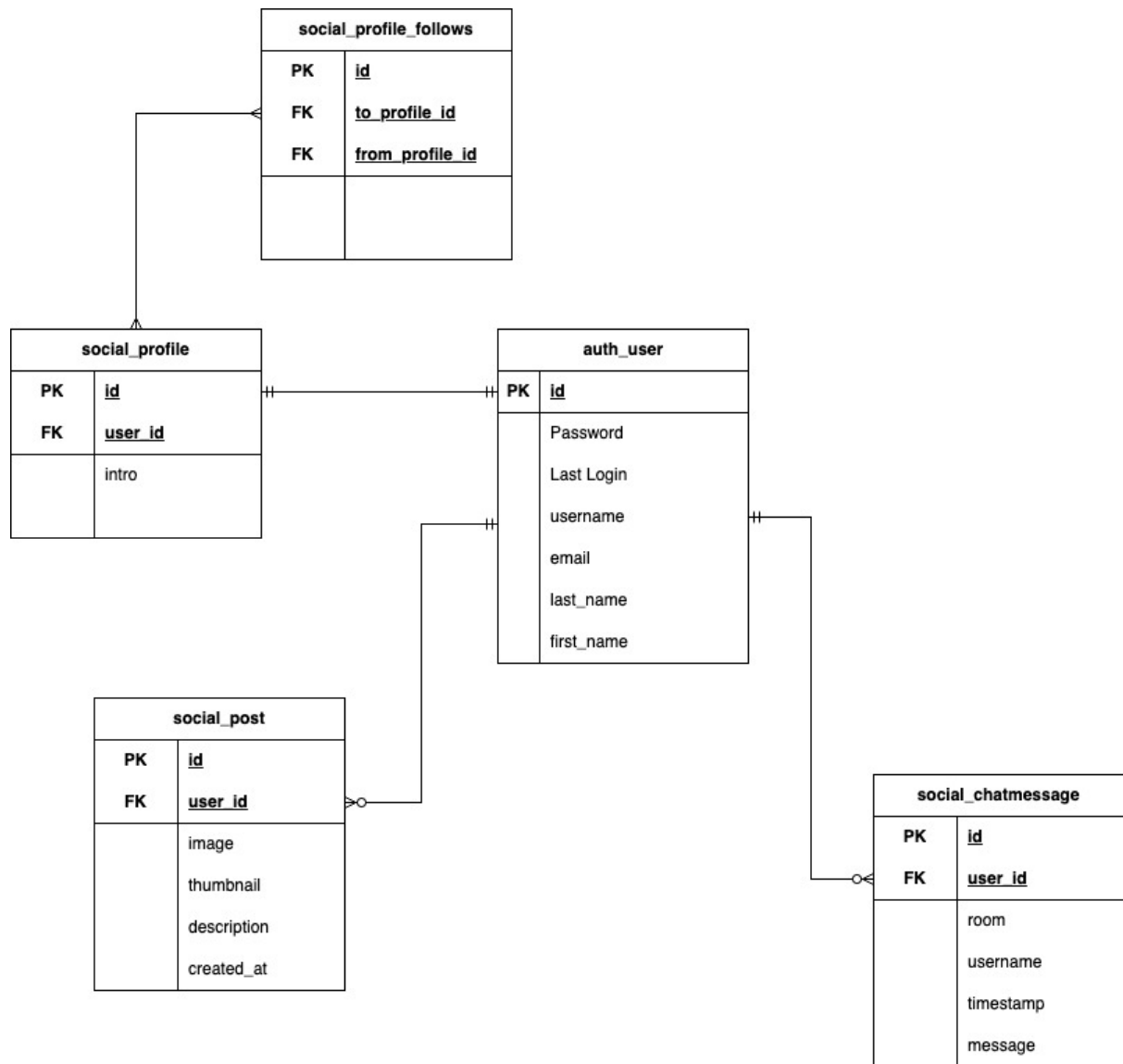
Image of Discover Page where users are able to find other users

Backend

This section will provide a comprehensive overview of the backend of Social. I will be providing detailed description of the Social's backend functions, explanation on how each feature of Social are implemented and optimised to provide a seamless user experience.

Database

The model is the component that represents the data layer of the Django application. It defines the structure and behaviour of the database tables. The image below is a entity relation diagram depicting the relation between different tables within Social's database defined by the Django model of Social.



From the image above, Social utilise 5 different tables to store all data related to users. They are `auth_user`, `social_profile`, `social_profile_follows`, `social_post`, and `social_chatmessage`. The purpose of the tables are as follow.

auth_user

Stores all user information such as password, username, email and name

social_profile

Stores user's customizable profile such as intro.

social_profile_follows

Stores user's following list and follower list.

social_post

Stores user's image posts on Social

social_chatmessage

Stores user's chat messages with other users.

Overall the database designed above allows Social to store all data related to social dynamically. It was designed in a way that stores minimal redundant data and allows for efficient queries and scalability. Furthermore, using Django's built-in authentication system and user model provides a safe and efficient method of managing user authentication and authorization. This system enables Social to manage user accounts and access control.

Model

This section will describe the codes within `models.py`.

Imports

Within `model.py` we made use of the following imports.

```
from django.db import models
from django.contrib.auth.models import User
from django.db.models.signals import post_save
from django.dispatch import receiver
```

As seen from the code extract, we have imported `User` from `django.contrib.auth.models`. This is because Social will be using the in built Django `User` model to store user's data.

Post

The following code extract from `models.py` is used to define the `social_post` table within the database. The following model uses a foreign key to the User Model so as to draw reference to the user for each post the user made. An override to the return statement was also made such that details of the post can be returned easily when searched from the database.

```
class Post(models.Model):
    user = models.ForeignKey(
        User,
        related_name="posts",
        on_delete=models.DO_NOTHING
```

```

)
description = models.CharField(max_length=256, db_index=True)
image = models.FileField(blank=False)
thumbnail = models.FileField(null=True)
created_at = models.DateTimeField(auto_now_add=True)

def __str__(self):
    # Return a string containing user, date, and description
    return(
        f"{self.user} "
        f"({self.created_at:%Y-%m-%d %H:%M}) : "
        f"{self.description}"
    )

```

Profile

The `Profile` model has a one to one relation with the `User` model. In this model we defined the entries to be stored within the `social_profile` table and the `social_profile_follows` table.

```

#Create a user profile model
class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    follows = models.ManyToManyField("self",
                                    related_name="followed_by",
                                    symmetrical=False,
                                    blank=True)
    date_modified = models.DateTimeField(User, auto_now=True)
    intro = models.CharField(max_length=100, blank=True)
    def __str__(self):
        return self.user.username

```

As defined in the code above, the `Profile` model consists of the fields: `user`, `follows`, `date_modified`, and `intro`. The `follows` field is defined to be in a many to many relationship with the Profile model. Hence the `social_profile_follows` table was created by Django to store information related to this many to many relationship.

The Profile model also uses receiver. It listens to the `post_save` event of the `User` model. Whenever a new `User` is created, the signal handler `create_profile` is called and a new Profile linked to the `User` model will be created. As seen from the code extract below, the newly created user profile is also set to follow themselves. This is so that users will be able to see what they post in their feed.

```

#Create profile when new user signs up
@receiver(post_save, sender=User)
def create_profile(sender, instance, created, **kwargs):
    if created:

```

```

user_profile = Profile(user=instance)
user_profile.save()
#have the user follow themselves
user_profile.follows.set([instance.profile.id])
user_profile.save()

```

Chat Message

The `ChatMessage` model is a model used to store chat messages. When users sent a message in the chat room, the `user_id` and `username` of the user who sent the message, `room`, `message` and `timestamp` of the message are being stored as an entry within the `social_chatmessage` table.

```

# Chat Message
class ChatMessage(models.Model):
    user = models.ForeignKey(
        User,
        on_delete=models.DO_NOTHING
    )
    username = models.CharField(max_length=255)
    room = models.CharField(max_length=255)
    message = models.TextField()
    timestamp = models.DateTimeField(auto_now_add=True)

    class Meta:
        ordering = ['-timestamp']

```

Functionality of Each Page

Index (Home Page)

The home page is defined in `view.py` as the `index`. The function of the home page is to generate a feed containing the posts of all users.

views.py

```

#Home Page
def index(request):
    if request.user.is_authenticated:
        posts = Post.objects.all().order_by("-created_at")

        return render(request, 'index.html', {"posts": posts})
    else:
        return render(request, 'index.html', {})

```

When a get request is sent to the URL of the homepage, the function `index()` defined in the `views.py` above will check if the user is authenticated.

If user is authenticated it will retrieve all posts within the database and render it onto the HTML in the front end. Else, it will render a blank list onto the HTML.

index.html

```
{% extends 'base.html' %}
{% load static %}

{% block content %}
{% if user.is_authenticated %}
<div class="card" style="width: 90%;">
  <div class="card-body" id="input">
    <form class="card-text" action="/api/createpost/" method="post" enctype="multi
part/form-data">
      {% csrf_token %}
      <label for="image">Upload Image:</label><br>
      <input type="file" id="image" name="image"><br><br>
      <label for="description">Description:</label><br>
      <input type="text" id="description" name="description"><br><br>
      <input type="submit" value="Submit">
    </form>
  </div>
</div>
</div>

{% else %}
<div class="px-4 py-5 my-5 text-center">
  
  <h1 class="display-5 fw-bold">Social</h1>
  <div class="col-lg-6 mx-auto">
    <p class="lead mb-4">Welcome to Social, the largest social webspace in the wor
ld to share images with your
    friends!</p>
    <div class="d-grid gap-2 d-sm-flex justify-content-sm-center">
      <a href="{% url 'login' %}" type="button" class="btn btn-outline-primary b
tn-lg px-4 gap-3">Login</a>
      <a href="{% url 'signup' %}" type="button" class="btn btn-outline-secondar
y btn-lg px-4">Sign Up</a>
    </div>
  </div>
</div>
</div>
{% endif %}
<br>
<br>

{% for post in posts %}
<div class="card" style="width: 90%;">
  
  <div class="card-body">
    <p class="card-text">{{post.description}}</p>
    <small class="text-muted">
```



```

        <a href="{% url 'profile' post.user.id %}">@{{ post.user.username }}</a> |
        {{ post.created_at }}
    </small>
</div>
</div>

<br>
<br>
{% endfor %}

{% endblock %}

```

In the HTML page, Django template language is used to display the appropriate frontend view for the user. From the `index.html` code above, it checks whether the user is authenticated using Django template language. If user is authenticated, `index.html` will generate a view displaying that of a list of images.

However, if users is not authenticated, it will generate a page which request for user to login or sign up for a new account.

Images of both different views is within the User interface section of this report

User List (discovery)

views.py

The discovery page is defined in `view.py` as the `user_list`. The function of the discovery page is to allow users to find and interact with other users.

```

#Discovery Page
def user_list(request):
    if request.user.is_authenticated:
        profiles = Profile.objects.exclude(user=request.user)
        return render(request, 'discover.html', {"profiles": profiles})
    else:
        messages.success(request, ("Please try again after logging in.))
        return redirect('/')

```

When a get request is sent to the URL of the discovery page, the function `user_list()` defined in the `views.py` above will check if the user is authenticated. If user is not authenticated he will be redirected to the home page and along with the message "Please try again after logging in."

Else, users will be taken to the discovery page. The function above will retrieve all profiles within the database and render them on the discovery page.

discover.html

```
{% extends 'base.html' %}
{% load static %}

{% block content %}
<h1>Discover</h1>

<br>
{% if profiles %}
{% for profile in profiles %}
<div class="card mb-3" style="max-width: 540px;">
    <div class="row g-0">
        <div class="col-md-4">
            
        </div>
        <div class="col-md-8">

            <div class="card-body">
                <h5 class="card-title">{{ profile.user.username }}</h5>
                <p class="card-text">
                    <a href="{% url 'profile' profile.user.id %}">
                        @{{ profile.user.username|lower }}
                    </a>
                </p>
                <p class="card-text">{{ profile.intro }}</p>
                <p class="card-text"><small class="text-muted">Last updated: {{ profil
e.date_modified }}</small></p>

            </div>
        </div>
    </div>
</div>
{% endfor %}
{% endif %}

{% endblock %}
```

Within the `discover.html` , Django template language is used to iterate through the list of user profiles rendered by `views.py` and display informations from each user profile.

User Profile

The user profile page is defined in `view.py` as the `user_profile` . The function of the user profile page is to display user's images and allow interaction between different users.

views.py

```

#User Profile Page
def user_profile(request, pk):
    if request.user.is_authenticated:
        profile = Profile.objects.get(user_id=pk)
        posts = Post.objects.filter(user_id=pk).order_by('-created_at')

        if request.method == "POST":
            current_user_profile = request.user.profile
            action = request.POST['follow']
            # Follow or unfollow user
            if action == "unfollow":
                current_user_profile.follows.remove(profile)
            elif action == "follow":
                current_user_profile.follows.add(profile)
            # Save profile
            current_user_profile.save()

        return render(request, 'profile.html', {"profile": profile, "posts": posts})

    else:
        messages.success(request, ("Please try again after logging in. "))
        return redirect('/')

```

In the above code snippet of `views.py`, the `user_profile` function retrieves all posts created by the user specified in the argument of the URL and render it on the front end. The profile of the targeted user is also retrieved from the database. The list of followers and following of the user profile is also rendered onto the front end.

profile.html

```

{% extends 'base.html' %}
{% load mathfilters %}
{% block content %}
{% if profile %}
<div class="container">

    <div class="row">

        <div class="col-10">
            <div class="jumbotron">
                <h1 class="display-4">{{profile.user.username}}'s Profile</h1>
                <p class="lead"></p>
                <hr class="my-4">
                <p>{{profile.intro}}</p>
            </div>

            <br>
            <br>

            {% if posts %}
            {% for post in posts %}

```

```

        <div class="card" style="width: 90%;">
            
            <div class="card-body">

                <p class="card-text">{{post.description}}</p>
                <small class="text-muted">
                    <a href="{% url 'profile' post.user.id %}">@{{ post.user.username }}</a> |
                        {{ post.created_at }}
                </small>
            </div>
        </div>
    <br>
{% endfor %}
{% endif %}

</div>

<div class="col-2 text-center">
    {% if profile.user.id == user.profile.user_id %}
    <!-- Button trigger modal -->
    <button type="button" class="btn btn-primary btn-lg" data-bs-toggle="modal" data-bs-target="#exampleModal">
        Edit Profile
    </button>

    <!-- Modal -->
    <div class="modal fade" id="exampleModal" tabindex="-1" aria-labelledby="exampleModalLabel"
        aria-hidden="true">
        <div class="modal-dialog">
            <div class="modal-content">
                <div class="modal-header">
                    <h5 class="modal-title" id="exampleModalLabel">Edit Profile
                </h5>
                <button type="button" class="btn-close" data-bs-dismiss="modal"
                    aria-label="Close"></button>
                </div>
                <form action="{% url 'profile_update' profile.user.id %}" method="post">
                    enctype="multipart/form-data">
                    {% csrf_token %}
                    <div class="modal-body">
                        <div class="form-group">
                            <textarea class="form-control" id="intro" name="intro"
                                placeholder="Write your introduction here."
                                rows="3">{{ profile.intro }}</textarea>
                        </div>
                    </div>
                    <div class="modal-footer">
                        <button type="button" class="btn btn-secondary" data-bs-dismiss="modal">Close</button>
                    </div>
                </form>
            </div>
        </div>
    </div>

```

```

        <button type="submit" class="btn btn-primary">Save Changes</button>

        </div>
    </form>
</div>
</div>
</div>

{% else %}

    <a href="{% url 'chatroom' user.profile.user_id|mul:profile.user.id profile.user.id %}" class="btn btn-primary btn-lg">Chat</a>
{% endif %}

<br>
<br>

<div class="card">
    <div class="card-body">
        <h5 class="card-title">Following</h5>
        {% for following in profile.follows.all %}
            {% if following.user.id != profile.user.id %}
                <a href="{% url 'profile' following.user.id %}" class="card-link">@{{following}}</a><br>
            {% endif %}
        {% endfor %}
    </div>
</div>

<br>

<div class="card">
    <div class="card-body">
        <h5 class="card-title">Followers</h5>
        {% for following in profile.followed_by.all %}
            {% if following.user.id != profile.user.id %}
                <a href="{% url 'profile' following.user.id %}" class="card-link">@{{following|lower}}</a><br>
            {% endif %}
        {% endfor %}
    </div>
</div>

<br>

{% if profile.user.id != user.profile.user_id %}
    <form method="post">
        {% csrf_token %}
        {% if profile in user.profile.follows.all %}
            <button type="submit" class="btn btn-outline-danger" name="follow" value="unfollow">
                Unfollow @{{profile.user.username|lower}}
            </button>
        {% else %}
            <button type="submit" class="btn btn-outline-success" name="follow" value="follow">

```

```

        Follow @{{profile.user.username|lower}}
    </button>
    {% endif %}
</form>
{% endif %}

</div>
</div>
</div>

{% endif %}

{% endblock %}

```

The above HTML code of `profile.html` will then make use of Django template language to display lists of data that were rendered by `view.py`.

Post request can also be triggered within the profile page through buttons on the HTML page. This will be captured by `user_profile()` function of `view.py` if the action is 'follow' or 'unfollow'. When trigger the targeted user will be followed or unfollowed by the authenticated user.

User Login

The user login page is where users are able to log in to their accounts. It is defined in `view.py` as `user_login`. The `user_login` function in `views.py` renders a log in page for the users and also listens for post request coming from user login page URL. If user is successfully logged in, it will redirect them to the home page, else a message will be sent to the users telling them that "Log in failed".

views.py

```

#User Login Page
def user_login(request):
    if request.method == "POST":
        username = request.POST['username']
        password = request.POST['password']
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user)
            messages.success(request, ("You are logged in"))
            return redirect('/')
        else:
            messages.success(request, ("Log in failed "))

    return render(request, "login.html", {})

```

login.html

```
{% extends 'base.html' %}

{% block content %}
{% if user.is_authenticated %}
    <h1>You are already logged in</h1>
{% else %}
<h1>Login</h1>
<br>
<form method="post" action="{% url 'login' %}">
    {% csrf_token %}
    <div class="mb-3">
        <label class="form-label">Username</label>
        <input type="text" class="form-control" name="username" placeholder="Enter your username">
    </div>
    <div class="mb-3">
        <label class="form-label">Password</label>
        <input type="password" class="form-control" name="password" placeholder="Enter your password">
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

{% endif %}
{% endblock %}
```

The `user_login` function will render the above HTML on the front end when users visit the login page URL.

User Logout

The `user_logout` function in the `views.py` is to log the user out. When user clicked on the URL linked to `user_logout` function they will be logged out and redirected to the home page.

views.py

```
def user_logout(request):
    logout(request)
    messages.success(request, ("Successfully logged out"))
    return redirect('/')
```

User Sign Up

The `user_signup` function in the `views.py` is to render a page for user to sign up for a new account.

views.py

```
def user_signup(request):  
    return render(request, "signup.html", {})
```

This function does not contain much as for the sign up function I made use of an API to trigger account creation process. This api is triggered by the form group within the HTML.

form.py

```
<form class="mx-1 mx-md-4" name="signup" action="{% url 'signup_api' %}" method="post"  
onsubmit="return validateForm()">  
    {% csrf_token %}  
    <!-- First Name -->  
    <div class="d-flex flex-row align-items-center mb-4">  
        <i class="fas fa-user fa-lg me-3 fa-fw"></i>  
        <div class="form-outline flex-fill mb-0">  
            <input class="form-control" type="text" name="first_name" required  
/>  
            <label class="form-label" for="first_name">First Name</label>  
        </div>  
    </div>  
    <!-- Last Name -->  
    <div class="d-flex flex-row align-items-center mb-4">  
        <i class="fas fa-user fa-lg me-3 fa-fw"></i>  
        <div class="form-outline flex-fill mb-0">  
            <input class="form-control" type="text" name="last_name" required  
/>  
            <label class="form-label" for="last_name">Last Name</label>  
        </div>  
    </div>  
    <!-- Username -->  
    <div class="d-flex flex-row align-items-center mb-4">  
        <i class="fas fa-user fa-lg me-3 fa-fw"></i>  
        <div class="form-outline flex-fill mb-0">  
            <input class="form-control" type="text" name="username" required>  
            <label class="form-label" for="username">Username</label>  
        </div>  
    </div>  
    <!-- Email -->  
    <div class="d-flex flex-row align-items-center mb-4">  
        <i class="fas fa-envelope fa-lg me-3 fa-fw"></i>  
        <div class="form-outline flex-fill mb-0">
```



```

        <input class="form-control" type="email" name="email" required>
        <label class="form-label" for="email">Email</label>
    </div>
</div>

<!-- Password1 -->
<div class="d-flex flex-row align-items-center mb-4">
    <i class="fas fa-lock fa-lg me-3 fa-fw"></i>
    <div class="form-outline flex-fill mb-0">
        <input class="form-control" type="password" name="password" requir
ed>

        <label class="form-label" for="password">Password</label>
    </div>
</div>

<!-- Password2 -->
<div class="d-flex flex-row align-items-center mb-4">
    <i class="fas fa-key fa-lg me-3 fa-fw"></i>
    <div class="form-outline flex-fill mb-0">
        <input class="form-control" type="password" name="password2" requi
red>

        <label class="form-label" for="password2">Confirm your
        password</label>
    </div>
</div>

<div class="d-flex justify-content-center mx-4 mb-3 mb-lg-4">
    <button class="btn btn-primary btn-lg" type="submit" value="Submit">Si
gn up</button>
</div>

</form>

```

The reason is because I intend to make use of APIs in this use case for users to create account, this is so that I won't be creating duplicate functions of similar functionality.

Chat room

The chat room is implemented similar to that taught by the coursera. I have made some changes to it. Each user is given a unique user id.

```

<a href="{% url 'chatroom' user.profile.user_id|mul:profile.user.id profile.user.id %}"
    class="btn btn-primary btn-lg">Chat</a>

```

With the above url, the authenticated user's id is multiplied together with the targeted user's id to generate a unique room number. Within the chat room, only both users with the combination of id multiplied together can enter.

views.py

```
# Chat room
def chatroom(request, room_name, pk):
    # Retrieve chat history from database
    if request.user.is_authenticated:
        profile = Profile.objects.get(user_id=pk)

    chat_messages = ChatMessage.objects.filter(room=room_name).order_by('timestamp')

    return render(request, 'chat/room.html', {
        'room_name': room_name,
        'chat_messages': chat_messages,
        'profile': profile
    })
```

The chatroom function above retrieves all chat with the room name from the database and render it as chat history.

consumers.py

```
import json
from channels.db import database_sync_to_async
from channels.generic.websocket import AsyncWebsocketConsumer
from .models import ChatMessage
from django.contrib.auth.models import User

class ChatConsumer(AsyncWebsocketConsumer):
    async def connect(self):
        self.room_name = self.scope['url_route']['kwargs']['room_name']
        self.room_group_name = 'chat_%s' % self.room_name

        await self.channel_layer.group_add(
            self.room_group_name,
            self.channel_name
        )

        await self.accept()

    async def disconnect(self, close_code):
        await self.channel_layer.group_discard(
            self.room_group_name,
            self.channel_name
        )

    async def receive(self, text_data):
        data = json.loads(text_data)
        message = data['message']
        username = data['username']

        # Store the message in the database
```

```

        await self.save_chat_message(username, message)

        await self.channel_layer.group_send(
            self.room_group_name,
            {
                'type': 'chat_message',
                'message': message,
                'username': username # include the username in the message event
            }
        )

    async def chat_message(self, event):
        # Send message to WebSocket
        await self.send(text_data=json.dumps(event))

    @database_sync_to_async
    def save_chat_message(self, username, message):
        #Search User_id from User object to store in DB
        user = User.objects.get(username = username)
        # Create a new ChatMessage object and save it to the database
        ChatMessage.objects.create(user=user, room=self.room_name, message=message, username=username)

```

Within the `consumers.py` the function `save_chat_message` is added, and it automatically save messages to the database when a message is sent.

room.js

```

const roomName = JSON.parse(document.getElementById('room-name').textContent);

const chatSocket = new WebSocket('ws://' + window.location.host + '/ws/' + roomName + '/');
document.querySelector('#chat-log').scrollTop = document.querySelector('#chat-log').scrollHeight;

chatSocket.onmessage = function (e) {
    // Forcing Date sent through Websocket to be same as retrieved from Django
    const data = JSON.parse(e.data);

    const months = ["Jan.", "Feb.", "Mar.", "Apr.", "May", "Jun.", "Jul.", "Aug.", "Sep.", "Oct.", "Nov.", "Dec."];
    const now = new Date();
    const month = months[now.getMonth()];
    const day = now.getDate();
    const year = now.getFullYear();
    const hours = now.getHours();
    const minutes = now.getMinutes();
    const ampm = hours >= 12 ? "p.m." : "a.m.";
    const formattedHours = hours % 12 || 12;
    const formattedMinutes = minutes < 10 ? "0" + minutes : minutes;
    const formattedTime = month + " " + day + ", " + year + ", " + formattedHours + ":" + formattedMinutes + " " + ampm;
    document.querySelector('#chat-log').value += (formattedTime + ' - ' + data.username + '\n');
}

```

```

e + ': ' + data.message + '\n\n');
    document.querySelector('#chat-log').scrollTop = document.querySelector('#chat-log').scrollHeight;

};

chatSocket.onclose = function (e) {
    console.error('Chat socket closed unexpectedly');
};

document.querySelector('#chat-message-input').focus();
document.querySelector('#chat-message-input').onkeyup = function (e) {
    if (e.keyCode === 13) { // enter, return
        document.querySelector('#chat-message-submit').click();
    }
};

document.querySelector('#chat-message-submit').onclick = function (e) {
    const messageInputDom = document.querySelector('#chat-message-input');
    const message = messageInputDom.value;
    const username = document.getElementById("chat-message-submit").getAttribute("name");
    chatSocket.send(JSON.stringify({
        'message': message,
        'username': username
    }));
    messageInputDom.value = '';
};

```

The javascript code is similar to that taught in lectures but I added on the username and date to be rendered alongside with the message when message is sent.

This is an extremely rudimentary way of creating a private chatrooms. There can certainly be better way of creating a private chatroom through the use of more complicated encryption method to set the room number, and also specific headers combinations such that only user's who sent the correct header within the request can enter the chat room. However, due to time limitation, it is more important to create the basic structure of the chatroom. The structure of the code is designed such that it is easy to improve on top of it to create a more secure chat room.

API

This section will be going through all the rest API that is available on Social. For the APIs, I have written in depth comments on the utility of each line of codes. These functions are meant to provide the users with additional flexibility on how they can use Social.

Create Post

The create post api is used to create a new post for the user. The API is design to only work if a user is authenticated. Users who are not authenticated will be unable to use this API.

```
# api/createpost/
class CreatePost(mixins.CreateModelMixin, generics.GenericAPIView):
    # Define the Post queryset
    queryset = Post.objects.all()
    # Define the PostSerializer
    serializer_class = PostSerializer

    # perform_create() is called when a new post is created
    def perform_create(self, serializer):
        if self.request.user.is_authenticated:
            # Get the current user and assign the new post to them
            current_user = User.objects.get(username=self.request.user)
            record = serializer.save(user=current_user)
            # Create a thumbnail for the new post asynchronously
            make_thumbnail.delay(record.pk)

    # Override create method to handle redirection after a new post is created
    def create(self, request, *args, **kwargs):
        if self.request.user.is_authenticated:
            #Check if all data/arguments is valid befor creating post
            serializer = self.get_serializer(data=request.data)
            serializer.is_valid(raise_exception=True)
            #Create post
            response = super(CreatePost, self).create(request, *args, **kwargs)

            # Redirect to the home page after a new post is created
            return HttpResponseRedirect(redirect_to='/')
        else:
            return Response({'error': 'User is not authenticated'}, status=status.HTTP_
_401_UNAUTHORIZED)

    # Handle POST requests
    def post(self, request, *args, **kwargs):
        if not self.request.user.is_authenticated:
            return Response({'error': 'User not authenticated'}, status=status.HTTP_40
1_UNAUTHORIZED)

        return self.create(request, *args, **kwargs)
```

List Posts

This API is used to show all posts within Social.

```
# api/posts/
class PostList(generics.ListAPIView):
```

```
queryset = Post.objects.all()
serializer_class = PostListSerializer
```

List Users

This API is used to show all users within Social.

```
# URL: api/users/
class UserList(generics.ListAPIView):
    queryset = User.objects.all()
    serializer_class = UserListSerializer
```

User detail

This API is used to show details of specific users within Social.

```
# URL: api/user/<int:pk>
class UserDetail(generics.RetrieveAPIView):
    queryset = User.objects.all()
    serializer_class = UserListSerializer
```

User Sign Up

This api is used to sign up a new account in social.

```
# URL: api/signup/
class User_signup(generics.CreateAPIView):
    queryset = User.objects.all()
    serializer_class = UserSignupSerializer

    def post(self, request, *args, **kwargs):
        # Extract user data from the request.POST QueryDict and make a mutable copy
        username = request.POST.get('username')
        email = request.POST.get('email')
        password = request.POST.get('password')
        first_name = request.POST.get('first_name')
        last_name = request.POST.get('last_name')

        print(first_name, last_name)

        try:
            # Attempt to create a new user with the provided data
            new_user = User.objects.create_user(
                username=username,
                email=email,
                password=password,
                first_name=first_name,
                last_name=last_name
```

```

    )

    except IntegrityError:
        # If a user with the same username or email already exists, show an error
        message and redirect to signup page
        messages.error(
            request, 'An account with that username or email already exists.')
        return HttpResponseRedirect(redirect_to='/signup')

    # Check if user is authenticated, and log them out if necessary
    if request.user.is_authenticated:
        logout(request)

    # Authenticate and log in the new user
    user = authenticate(username=username, password=password)
    login(request, user)

    # Send success message to the front end and prepare response data
    messages.success(request, 'Welcome to Social!')
    response_data = {'message': 'User created successfully!'}

    return HttpResponseRedirect(redirect_to='/users')

```

User Profile

This API is use to get user profile and also update user profile.

```

# api/profile/<int:pk>
class UserProfile(mixins.RetrieveModelMixin, mixins.UpdateModelMixin, generics.Generic
APIView):
    serializer_class = ProfileSerializer

    def get_object(self):
        # Retrieve the user ID from URL parameters
        user_id = self.kwargs.get('pk')
        if user_id:
            # Get the profile of the user with that ID if it exists
            return get_object_or_404(Profile, user_id=user_id)
        # If no user ID was provided, return the profile of the authenticated user
        return self.request.user.profile

    def get(self, request, *args, **kwargs):
        # Get the profile data using the serializer and return it in a response
        response_data = self.serializer_class(self.get_object()).data
        return Response(response_data)

    def post(self, request, *args, **kwargs):
        # Get the profile object to update
        profile = self.get_object()
        # Create a mutable copy of the request data
        data = request.data.copy()
        # Set the user ID of the profile object on the copied data to ensure it is ass
ociated with the correct user
        data['user'] = profile.user.id

```

```

        # Use the serializer to validate the updated data and save it to the database
        serializer = self.serializer_class(profile, data=data)
        if serializer.is_valid():
            serializer.save()
            # Redirect to the URL for the updated profile
            url = reverse('profile', args=[profile.user.id]) # get URL for user_profile view
            return HttpResponseRedirect(redirect_to=url)
        else:
            # If the serializer is not valid, return an error response with the validation errors
            return Response(serializer.errors, status=400)

```

Show list of Messages in room

This function is used to show all chat messages within a chat room.

```

# api/chat/<str:room>/
class ChatMessageList(generics.ListAPIView):
    serializer_class = ChatMessageSerializer

    def get_queryset(self):
        room = self.kwargs['room']
        return ChatMessage.objects.filter(room=room)

```

Test

Running the Test

To run the test, run the following commands within the terminal.

```

$ cd social_media
$ python manage.py test

```

Create Post Test

This test is used to test the create post API that I have created. First, this test if a new post will be created when this API is run when user is authenticated. Next, this test

test if a new post will be created if user is not authenticated.

```
class CreatePostTest(APITestCase):
    def setUp(self):
        #Login
        self.user = User.objects.create_user(
            username='testuser',
            password='testpass'
        )
        self.client.force_authenticate(user=self.user)

    def test_create_post(self):
        self.client.force_authenticate(user=self.user)

        # Open the image file and create a Django File object
        image_file = open(os.path.join(settings.BASE_DIR, 'social', 'static', 'image
s', 'user.png'), 'rb')

        django_file = File(image_file)

        # Define the post data, including the image file
        post_data = {
            'description': 'Test post',
            'image': django_file
        }

        # Send a POST request to the API
        response = self.client.post('/api/createpost/', post_data, format='multipart')

        # Check that the response status code is 302 (redirect)
        self.assertEqual(response.status_code, status.HTTP_302_FOUND)

        #Check if post is created
        self.assertEqual(Post.objects.count(), 1)

        #Check if post entries is same as what we posted
        post = Post.objects.first()
        self.assertEqual(post.description, 'Test post')
        self.assertEqual(post.user, self.user)

    def test_create_post_unauthenticated(self):
        self.client.force_authenticate(user=None)

        # Open the image file and create a Django File object
        image_file = open(os.path.join(settings.BASE_DIR, 'social', 'static', 'image
s', 'user.png'), 'rb')
        django_file = File(image_file)

        # Define the post data, including the image file
        post_data = {
            'description': 'Test post',
            'image': django_file
        }

        # Send a POST request to the API
        response = self.client.post('/api/createpost/', post_data, format='multipart')
```

```
# Check that the response status code is 401 (redirect)
self.assertEqual(response.status_code, status.HTTP_401_UNAUTHORIZED)
```

Post List Test

This test is to check if all posts are returned from the posts_api API. It posts multiple images and checks if the details returned in the post list are similar to that was created. It also checks if the URL for posts_api is valid.

```
class PostListTest(APITestCase):
    def setUp(self):
        #Login
        self.user = User.objects.create_user(
            username='testuser',
            password='testpass'
        )
        self.client.force_authenticate(user=self.user)

    def testURL(self):
        #Test if URL is valid
        url = reverse('posts_api')
        response = self.client.get(url)
        self.assertEqual(response.status_code, status.HTTP_200_OK)

    def test_post_list(self):
        # Create multiple posts with images
        for i in range(3):
            # Open the image file and create a Django File object
            image_file = open(os.path.join(settings.BASE_DIR, 'social', 'static', 'images', 'user.png'), 'rb')
            django_file = File(image_file)

            # Define the post data, including the image file
            post_data = {
                'description': f'Test post {i}',
                'image': django_file
            }

            # Send a POST request to create the post
            self.client.post('/api/createpost/', post_data, format='multipart')

            # Check if the posts are listed in the API response
            url = reverse('posts_api')
            response = self.client.get(url)
            self.assertEqual(response.status_code, status.HTTP_200_OK)

            # Check if the number of posts listed in the API response matches the number of posts created
            self.assertEqual(len(response.data), Post.objects.count())

            # Check if all the posts have valid image URLs in the API response
```

```

for post in response.data:
    self.assertIsNotNone(post['image'])
    self.assertTrue(post['image'].startswith('http://'))

#Check if images returned equals 3 as posted earlier
images_count = sum('image' in post for post in response.data)
self.assertEqual(images_count, 3)

```

User List Test

This test creates multiple users and check if the user list returned via the API is the same as the user's who are created.

```

class UserListTest(APITestCase):
    def setUp(self):
        #Create 2 different users
        self.user1 = User.objects.create_user(
            username='testuser1', email='testuser1@example.com', password='testpassword1')
        self.user2 = User.objects.create_user(
            username='testuser2', email='testuser2@example.com', password='testpassword2')

    def testURL(self):
        #test if URL is valid
        url = reverse('users_api')
        response = self.client.get(url)
        self.assertEqual(response.status_code, status.HTTP_200_OK)

    def test_users_list(self):
        #test if user list return 2 users that was created earlier
        url = reverse('users_api')
        response = self.client.get(url)
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertEqual(len(response.data), 2)

```

User Detail Test

This test checks if user detail returned by the API is the same as that of newly created user.

```

class UserDetailTest(APITestCase):
    def setUp(self):
        # Create a user
        self.user = User.objects.create_user(
            username='testuser',
            password='testpass',
        )

```

```

def test_get_user_detail(self):
    # Retrieve the user detail using the API
    url = reverse('users_api', kwargs={'pk': self.user.pk})
    response = self.client.get(url)

    # Check that the response status code is 200
    self.assertEqual(response.status_code, status.HTTP_200_OK)

    # Check that the response data matches the expected data
    self.assertEqual(response.data['id'], self.user.id)
    self.assertEqual(response.data['username'], self.user.username)

```

User Sign Up Test

This test check if users account is created properly when they are sign up. The details sent by the api should be the same as the one directly retrived from the database.

```

class UserSignupTest(APITestCase):
    def test_user_signup(self):
        # Define the user data to be submitted in the request
        user_data = {
            'username': 'tester',
            'email': 'test@example.com',
            'password': 'testpass',
            'first_name': 'Test',
            'last_name': 'User'
        }

        # Make a POST request to the signup API with the user data
        url = reverse('signup_api')
        response = self.client.post(url, user_data, format='multipart')

        # Check that the response status code is 302 (redirect)
        self.assertEqual(response.status_code, status.HTTP_302_FOUND)

        # Check that a user with the given username and email was created
        user = User.objects.get(username=user_data['username'])
        self.assertEqual(user.email, user_data['email'])

        # Check that the user is logged in
        self.assertTrue(self.client.login(username=user_data['username'], password=user_data['password']))

```

Update Profile Test

This test checks if profile is being updated properly when the API is used to update the profile. It also checks if data returned by the profile API is the same as that in the

database.

```
class UpdateProfileTest(APITestCase):
    def setUp(self):
        # Define the user data to be submitted in the request
        user_data = {
            'username': 'tester',
            'email': 'test@example.com',
            'password': 'testpass',
            'first_name': 'Test',
            'last_name': 'User'
        }
        # Make a POST request to the signup API with the user data
        self.client.post(reverse('signup_api'), user_data, format='multipart')
        # Log the user in
        self.client.login(username='tester', password='testpass')
        # Get user details
        self.user = User.objects.get(username='tester')

        #Set URL for test usage
        self.url = reverse('profile_update', args=[self.user.id])

    def test_get_profile(self):
        # Login user
        self.client.force_login(self.user)
        # Get response after login
        response = self.client.get(self.url)
        #Check if login works
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        #Get expected data from serializer
        expected_data = ProfileSerializer(self.user.profile).data
        #Check if profile complies with serializer
        self.assertEqual(response.data, expected_data)

    def test_update_profile(self):
        #Login user
        self.client.force_login(self.user)
        data = {
            'intro': 'Test intro'
        }
        #Post Update
        response = self.client.post(self.url, data)
        # Get response
        self.assertEqual(response.status_code, status.HTTP_302_FOUND)
        #Check if user profile have been updated
        self.user.refresh_from_db()
        self.assertEqual(self.user.profile.intro, 'Test intro')
```

Self Evaluation

This project was completed successfully and have met all requirements of the project. However, there are still rooms for improvements.

1. Lack of consistency:

Some of the codes uses API to listen for post, update or get requests while some of the codes uses the view function to listen for post, update or get requests.

For example, the create user API is used by the form to create a new user when the submit button is clicked. While the follow/ unfollow button uses the views function to listen for post request.

These should be standardized so that the code can easily understood by others and scale up upon.

2. Lack of API security:

It is best practice for API to have some form of security in place to prevent unwanted users from using the API for malignant purposes such as DDOS.

This could be accomplished by implementing API keys, rate limit, etc.

3. Lack of Security in Chatroom:

The current method of creating a chatroom by using a combination of the sender's and recipient's user IDs is insecure, as users can easily guess chatroom numbers and enter private chatrooms.

To improve security, it is recommended to implement a more complex encryption method for the room number. This would make it impossible for unauthorized users to enter private rooms by guessing the room number. Additionally, a special header can be put in place to prevent unauthorized users from entering private rooms.

Overall, I believe that most of the core functions of the application work very well together. I am pleased with the intuitive user interface design that I personally created and am glad that it turned out the way I wanted it to. I would not change any part of my project if I attempted the project again.

Conclusion

To conclude, this project allowed me to expand my knowledge and skills in using Django for Web development. Throughout this project I have learnt many powerful functionality of Django that allows me to create a social media website with key features within a short time. I am proud of this project that I have created and it has given me the confidence and skills to take on more complex web development projects in the future.

References

- Flatplanet. (n.d.). *Musker*. GitHub. Retrieved March 20, 2023, from <https://github.com/flatplanet/musker>
- payal0908. (n.d.). *socialnet: A Social Network application created with Django*. GitHub. Retrieved March 20, 2023, from <https://github.com/payal0908/SocialNet>
- Django. *Django Project*. (n.d.). Retrieved March 20, 2023, from <https://docs.djangoproject.com/en/4.1/>