

Introduction/Overview

In this Final project, I have created a social media website (Named “Awesome Social Web”) with four different applications under the hood, The detail of the applications, pages and functionalities are explained in the paragraph to follow:

1. Social App [First Django App]:

- 1.1. **Index Page** - Landing page for the users, clicking “Get Started” will navigate to the login page




- 1.2. **Login Page** - Enable user to login to the application, Navigates to between unauthenticated pages such as index page (Landing Page), Login and Signup pages.



[Home](#) [Login](#) [Signup](#)



Sign in with 

Or

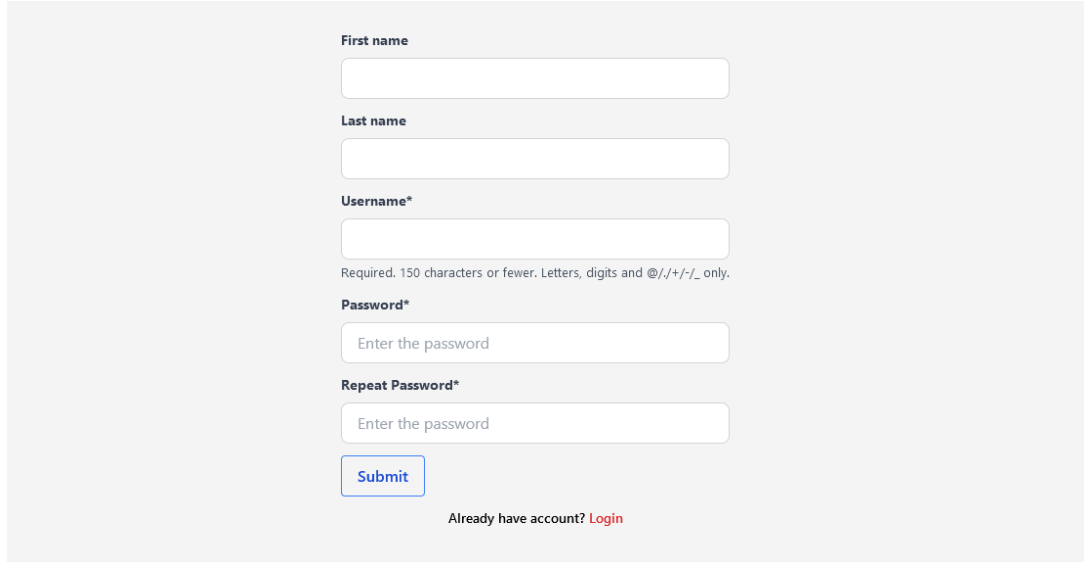
Username*

Password*

Login

Don't have an account? [Register](#)

- 1.3. **Resgiteration Page** - New users will get registered from this page and redirected to login page upon successful registration.



Registration form for Awesome Web. The form includes input fields for First name, Last name, Username*, Password*, and Repeat Password*. A Submit button is at the bottom. Below the Submit button is a link: "Already have account? [Login](#)".

First name

Last name

Username*

Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password*

Enter the password

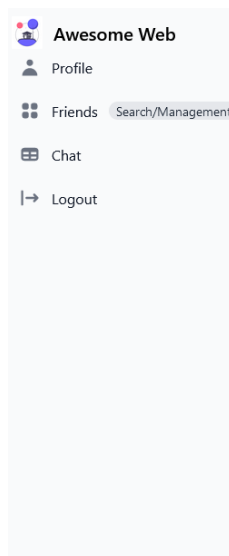
Repeat Password*

Enter the password

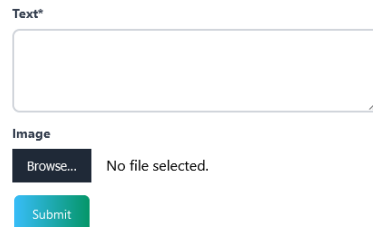
Submit

Already have account? [Login](#)

- 1.4. **Home Page** - One authenticated/Login, Users are navigated to the home page, which provide them with the authenticated sidebar for navigation between profile, friends, chat and logout from the application. This page allows the user to add a new post on the timeline.



Welcome **Ahmad** to Awesome Web



Post creation form for Ahmad. The form includes a Text* input field, an Image input field with a "Browse..." button, and a Submit button.

Text*

Image

Browse... No file selected.

Submit

- 1.5. **Create New Posts [Home Page]** - Users are able to create a new posts by adding text and images. Submitting the new post will get displayed on the home page as a list.

Text*


Image

Browse... No file selected.

Submit

Tweets

Ahmad has succesfully completed final term project for social web application



Edit Post Delete Post

- 1.6. **Delete/Update Post [Home Page]** - Post once added can be updated or delete completely from the database. Updating the Post and deleting is implemented dynamically using “HTMX” without loading the Home page. It asks for confirmation too before deleting the post from the database

Text*

Ahmad has succesfully completed final term project for social web application

Image

Browse... No file selected.

Save

Cancel

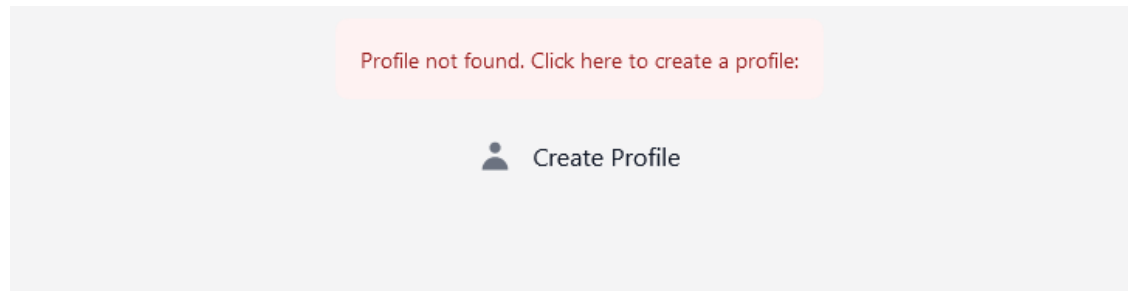
Are you sure you want to delete?

Ahmad has succesfully completed final term project for social web application

Confirm

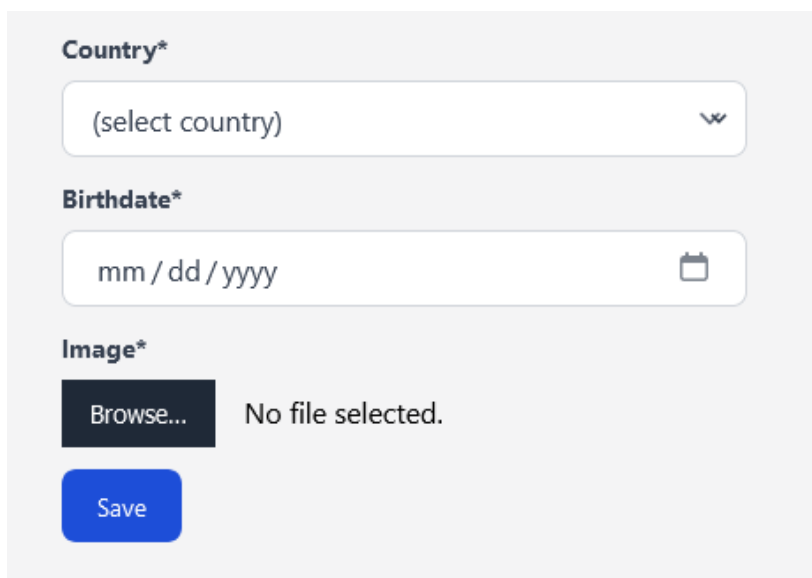
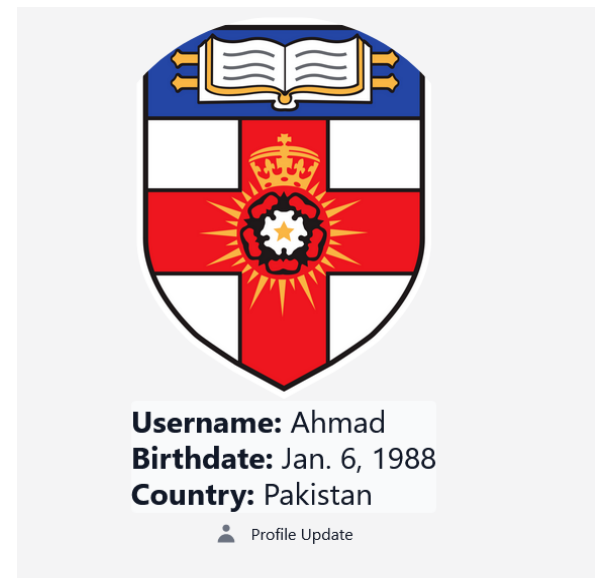
Cancel

- 1.7. **Profile** - There is separate page for the user to create a profile, It shows an empty profile on the navigation to the page for new users with a button to create a new profile.

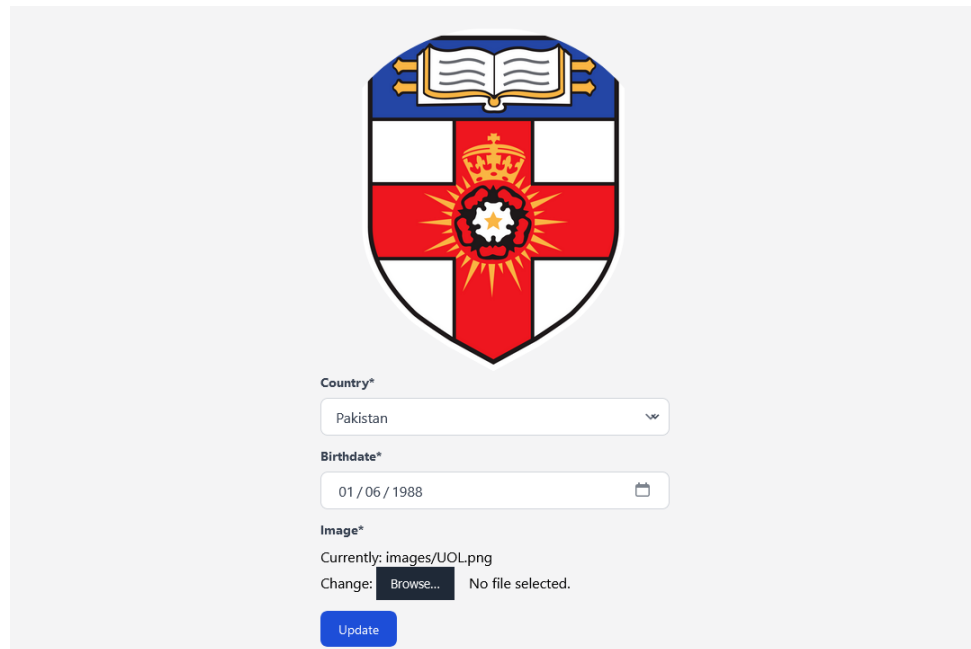


- 1.8. **Create Profile** - Profile create page shows a form with the following fields. Once profile is created will be available on profile page.

- Birthdate
- Country
- Image

A screenshot of a web form for creating a profile. It has three main sections: "Country*" with a dropdown menu showing "(select country)"; "Birthdate*" with a text input showing "mm / dd / yyyy" and a calendar icon; and "Image*" with a "Browse..." button, the text "No file selected.", and a blue "Save" button.

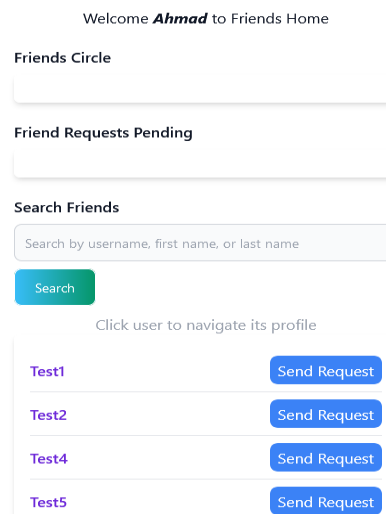
- 1.9. **Update Profile** - User can update the profile once created, there is profile update button automatically shown in the profile page, which could lead the user to profile update page. The same fields will be available to update.



A profile update form with a shield-shaped logo at the top. The logo features a red cross on a white background, with a crown and a gear in the center. Below the logo, there are three input fields: 'Country*' with a dropdown menu showing 'Pakistan', 'Birthdate*' with a date picker showing '01 / 06 / 1988', and 'Image*' with a text input showing 'Currently: images/UOL.png'. Below these fields is a 'Change:' label, a 'Browse...' button, and the text 'No file selected.' At the bottom is a blue 'Update' button.

2. Friends [Second Django App]:

- 2.1. **Friends Search** - User can search for the friends available in the database (*regardless search term is mentioned in upper case or lowercase, it will be converted to lower case before searching the user from the backend*), it will also show button to send the request to the users, the users to whom friend friends will fall into one of the following categories:
- Friend request pending for acceptance.
 - Friend with a button to delete from friend circle.



A 'Friends Home' page with a welcome message 'Welcome **Ahmad** to Friends Home'. It has three sections: 'Friends Circle' with an empty box, 'Friend Requests Pending' with an empty box, and 'Search Friends' with a search bar containing the placeholder 'Search by username, first name, or last name' and a green 'Search' button. Below the search bar is a link 'Click user to navigate its profile'. At the bottom is a table with five rows, each containing a username ('Test1', 'Test2', 'Test4', 'Test5') and a blue 'Send Request' button.

Username	Action
Test1	Send Request
Test2	Send Request
Test4	Send Request
Test5	Send Request

- Clicking the user name will also navigate to its own profile because in real-world applications users usually tend to check the profile of other users before sending any friend request.
- Sending friend requests will show the message on the top of page for users.

Friend request sent to Test16

Welcome **Ahmad** to Friends Home

- 2.2. **Accept Request:** Once the is sent, this will be showing to other users to accept the request, upon accepting the request it will start showing inside the Friend circle with an option to delete.

Welcome **Test16** to Friends Home

Friends Circle

Ola123

Delete

Test17

Delete

Friend Requests Pending

Ahmad

Accept

Search Friends

Search by username, first name, or last name

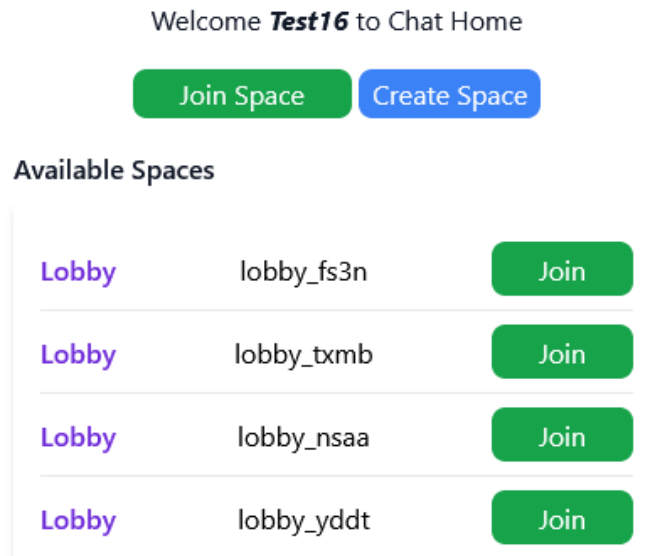
Search

3. Chat [Third Django App]:

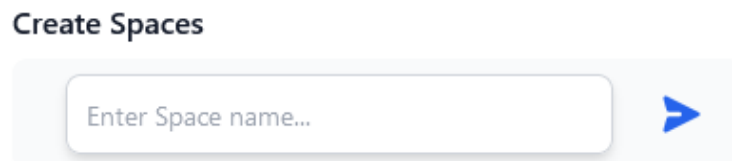
- 3.1. **Chat Home** - Chat app takes care the logic of chat between two users, it is implemented using Django channel and dephane ASGI server. Moreover, HTMX is users for users to chat with each

others. Following features are implemented for the chat application:

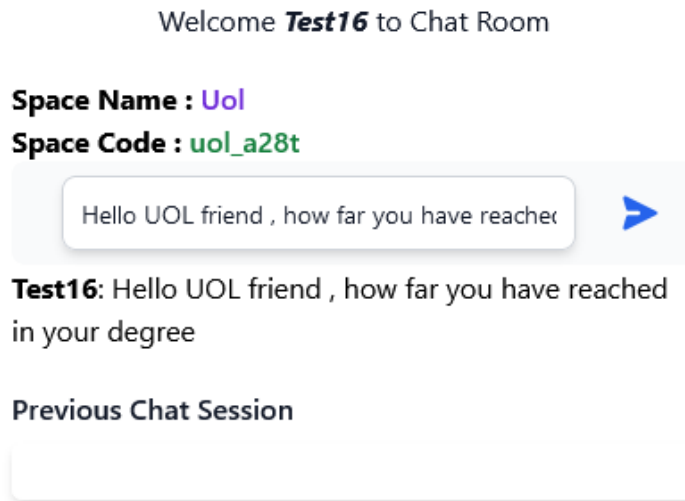
- Spaces[Rooms in normal chat app, following the concept from Twitter] could be created by users and then they will have the option to log into any room.
- All users will be able to chat with each other in the accessed space.
- Existing chat between users will be displayed upon accessing the room again.
- There will be a confirmation message is shown to the user (using ajax) before navigating to the selected space.
- There are unique code provided for each space because there is possibility of having multiple spaces with the same name.



- 3.2. **Create Space** - This let the users create a new space, it just requires single information (i.e Space name). Once created it will show in the list of available space.



- 3.3. **Chat Room** - Once, room is created it navigates to “Chat Room” Since, the room is new, therefore, no previous chat will be shown. However, interactions between users will be shown under “Previous chat history”. It has a feature of having live chat between the application users.



4. API [Forth Application]:

- 4.1. **API's** - There are different API created using Django Restframework. The brief description for the apis is as follows:

- User - API with list to show the users available in the application.

URL -> <http://127.0.0.1:8000/api/users/>

Response ->

GET /api/users/

HTTP 200 OK

Allow: GET, HEAD, OPTIONS

Content-Type: application/json

Vary: Accept

```
{
  "count": 32,
  "next": null,
  "previous": null,
  "results": [
```



```
{
  "username": "test4",
  "first_name": "test",
  "last_name": "test",
  "email": "test4@t.com"
},
{
  "username": "Test5",
  "first_name": "Test5",
  "last_name": "Test5",
  "email": "test4@t.com"
}, .....]
```

- Create user - API with Provide form to create a new user.

URL → http://127.0.0.1:8000/api/create_user/

Response →

HTTP 405 Method Not Allowed

Allow: POST, OPTIONS

Content-Type: application/json

Vary: Accept

```
{
  "detail": "Method \"GET\" not allowed."
}
```



- Create Profile - API to create the profile for the user already existed in the database.

URL → http://127.0.0.1:8000/api/create_profile/

Response →

HTTP 405 Method Not Allowed

Allow: POST, OPTIONS

Content-Type: application/json

Vary: Accept

```
{
  "detail": "Method \"GET\" not allowed."
}
```

The screenshot shows a web form titled 'User' for a Django REST framework application. The form includes input fields for Username, First name, Last name, Email address, Password, Birthdate (with a date picker icon), Image (with a 'Browse...' button and 'No file selected.' text), and Friends (a list box showing 'No items to select.'). A 'POST' button is at the bottom right. A dark banner at the top of the form area displays 'Django REST framework' on the left and 'Test16' on the right.

- Read Profile - Api to get the profile information already created for the user.

URL → http://127.0.0.1:8000/api/read_profile/Ahmad/

Response →

GET /api/read_profile/Ahmad/

HTTP 200 OK

Allow: GET, HEAD, OPTIONS

Content-Type: application/json

Vary: Accept

```
{
  "user": {
    "username": "Ahmad",
    "first_name": "Ahmad",
    "last_name": "Ahmad",
    "email": ""
  },
  "birthdate": "1988-01-06",
  "image": "http://127.0.0.1:8000/media/images/UOL.png",
  "friends": []
}
```

- User Posts - API to get the post available from the database

URL -> `http://127.0.0.1:8000/api/user_posts/Ahmad/`

Response ->

`GET /api/user_posts/Ahmad/`

HTTP 200 OK

Allow: GET, HEAD, OPTIONS

Content-Type: application/json

Vary: Accept

```
{
  "count": 1,
  "next": null,
  "previous": null,
  "results": [
    {
      "text": "Ahmad has succesfully completed final term
project for social web application",
      "image":
"http://127.0.0.1:8000/media/post_images/UOL.png"
    }
  ]
}
```

- 4.2. Unit Test** - Test are also defined in the application to test the basic functionality.

```
poetry run python manage.py test
Logging started on root for DEBUG
Found 6 test(s).
Creating test database for alias 'default'...
<function UserProfileCreate.form_valid at 0x000001AEB16E0AF0>
<function PostCreate.form_valid at 0x000001AEB16E0C10>
System check identified no issues (0 silenced).
.....
-----
Ran 6 tests in 2.443s
OK
Destroying test database for alias 'default'...
```

Technology

Backend - Django is used for backend, which is a highly sophisticated all-in-one web framework created in Python, is used to build the backend.

Routing, models, ORM, templating, database auto migrations, and are available in Django for developing backend APIs. Since it is being used for many years and has vast community members, therefore, many third-party packages are available which speed up the development process and also ensure reliability.

REST API endpoints, I have used Django Rest Framework (DRF), There are alternative options available such as Django ninja API, which is pretty much similar to fastapi. However, I have restricted myself with the DRF because of its popularity in the community.

Frontend - Django templating is used for the front end, it is mini language for creation of user-facing application layer. In common word it is normally referred to as Django template. There are alternative options available such as jinja2 which is quite similar to django templates. Anyone who understands HTML may manage templates; Python knowledge is not necessary. This promotes a clear division between application and presentation logic. Crispy form is used to get/post the information from frontend.

Scripting - Mix of HTMX and Ajax is used for scripting on the frontend of application, However, majorly dynamic rendering was performed using HMTX. HTMX is a JavaScript library that allows you to access modern browser features directly from HTML. It lets you perform AJAX requests, trigger CSS transitions, and invoke WebSocket and server-sent events directly from HTML elements, people with knowledge of HTML usually learn HTMX quickly and easily. This means you can build modern and powerful user interfaces with simple markups [source: <https://htmx.org/docs/>].

HTMX is quite small in size (~ 12k gzip) and in comparison to react application code base reduces to 67%.

Styling - Styling is handled using Tailwind CSS, which became quite popular in recent years, It provides utility classes similar to HTMX for user control, layout, colour, spacing etc. There are also prebuild

component based on tailwind. I have used both [Tailwind](#) and [flowbite](#) in the application, CDN are provided in the base template of the application for both Tailwind and flowbite.

[ASGI](#) - Django Channels, daphne, HTMX are used to add chat capability and support protocols other than HTTP, has been implemented.

The chat uses Websockets and requires daphne server to be running, since, the application is not of medium to large scale, therefore, depene is quite ok, however, for applications of large or production scale it is better to use one of the following:

- [Redies](#)
- [Channel_redies](#)

Navigation

Djnago is based on the concept of multipage application, therefore, Django URL's/routes are used for the navigation between different part/section of the application.

[Social App:](#)

[Landing page route](#)

```
path("", views.index, name="index"),
```

[Signup page route](#)

```
path("signup/", views.user_signup, name="signup"),
```

[Logout route](#)

```
path("logout/", views.user_logout, name="logout"),
```

[Login route](#)

```
path("login/", views.user_login, name="login"),
```

[Home page route - class based view](#)

```
path("home/", views.Home.as_view(), name="home"),
```

[User Profile Route - user name passed in URL](#)

```
path("profile/<str:username>/", views.user_profile, name="profile"),
```

[User Profile Update - class based view](#)

```
path("profile_update/", views.UserProfileEdit.as_view(),  
name="profile_update"),
```

[User Profile Create - class based view](#)

```
path("profile_create/", views.UserProfileCreate.as_view(),  
name="profile_create"),
```

[Post Create - class based view](#)

```
path("post_create/", views.PostCreate.as_view(), name="post_create"),
```

Post Update - class based view [Post id provided as parameter]

```
path("post/<int:pk>/edit/", views.PostUpdate.as_view(), name="post_update"),
```

Post Delete - class based view [Post id provided as parameter]

```
path("post/<int:pk>/delete/", views.PostDelete.as_view(),  
name="post_delete"),
```

Friends App:

Friend Section/Landing page - class based view

```
path("friends/", views.FriendHome.as_view(), name="friends"),
```

Friend Landing - class based view [Friend id provided as parameter]

```
path("friend_add/<int:pk>/", views.FriendCreate.as_view(),  
name="friend_add"),
```

Friend Accept - class based view [Friend id provided as parameter]

```
path("friend_accept/<int:pk>/", views.FriendAccept.as_view(),  
name="friend_accept"),
```

Friend Accept - class based view [Friend id provided as parameter]

```
path("friend_delete/<int:pk>/", views.FriendDelete.as_view(),  
name="friend_delete"),
```

Chat App:

Chat Section/Landing page - class based view

```
path("chat/", views.ChatHome.as_view(), name="chat_home"),
```

Space/Room - class based view [slug is provided as parameter]

```
path("chat/room/<str:slug>/", views.ChatRoom.as_view(), name="chat"),
```

Space/Room create - class based view

```
path("create/", views.SpaceCreate.as_view(), name="space_create"),
```

Space/Room join - class based view

```
path("join/", views.SpaceJoin.as_view(), name="space_join"),
```

API:

User List - class based List API View

```
path("api/users/", api.UserList.as_view(), name="users_api"),
```

User Create - class based Create API View

```
path("api/create_user/", api.CreateUser.as_view(), name="create_user_api"),
```

Profile Create - class based Create API View

```
path("api/create_profile/", api.CreateProfile.as_view(),  
name="create_profile_api"),
```

Profile Read - class based Retrieve API View

```
path("api/read_profile/<str:user__username>/", api.ReadProfile.as_view(),  
name="read_profile_api"),
```

User Posts - class based List API View

```
path("api/user_posts/<str:user__username>/", api.UserPosts.as_view(),
name="user_posts_api"),
```

Models

Each application have its own model, having separate application ensures modularity of application.

Social App:

This model is used to store the profile-related information for the user. Django builtin user module is used for authentication, therefore, it is used as foreign key in profile model. Path to store images is also provided in the model. Moreover, validator is also provided at model level for birthdate to ensure having date before current date.

```
class Profile(models.Model):
    # There is one to one relationship between profile and user
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    country = CountryField(blank_label="(select country)")
    birthdate = models.DateField(null=False, blank=False,
    validators=[MaxValueValidator(limit_value=date.today)])
    image = models.ImageField(upload_to="images/", null=True,
    default="images/undraw_cabin.jpg")
    # It allows to follow someone only without following back
    friends = models.ManyToManyField("self", related_name="followed_by",
    symmetrical=False, blank=True)
```

This model is for user posts with the foreign key of user and text, image fields. There is also limit provided to the text field similar to other social media apps such as twitter, facebook etc.

```
class Post(models.Model):
    owner = models.ForeignKey(User, on_delete=models.CASCADE)
    text = models.TextField(max_length=140)
    image = models.ImageField(upload_to="post_images/", null=True,
    blank=True)
    likers = models.ManyToManyField(User, blank=True, related_name="likes")
```

Chat App:

This model provide the chat name and slug for the space to make it unique because, there could be possibility that users might create chat spaces for same name multiple times. Users will have access to all spaces available and they can interact with each other.

```
class ChatSpace(models.Model):
    name = models.CharField(max_length=128)
```

```
slug = models.SlugField(unique=True)
users = models.ManyToManyField(User)
```

Chat messages are stored in this model. This was required to display the interaction between users once someone rejoin the space. Date of message is also provided, which is automatically selected by the servers.

```
class ChatMessage(models.Model):
    space = models.ForeignKey(ChatSpace, on_delete=models.CASCADE)
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    message = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
```

Chat App:

This model maintain the status of friend request between users. Once it is accepted date is provided and status is changed accordingly. The request creation date is also provided to keep track of each request. User is taken as foreign key for both users sending request and receiving requests.

```
class FriendRequest(models.Model):
    from_user = models.ForeignKey(User, related_name="friend_requests_sent",
    on_delete=models.CASCADE)
    to_user = models.ForeignKey(User,
related_name="friend_requests_received",
    on_delete=models.CASCADE)
    created_at = models.DateTimeField(auto_now_add=True)
    accepted = models.BooleanField(default=False)
    STATUS_CHOICES = (
        ("pending", "Pending"),
        ("accepted", "Accepted"),
        ("rejected", "Rejected"),
    )
    status = models.CharField(choices=STATUS_CHOICES, default="pending",
max_length=20)
```

Authentication

The authentication system that comes with Django manages user accounts, groups, permissions, and cookie-based user sessions. You can use this system to verify usernames and passwords and specify the actions that each user is permitted.

Django automatically provides four default permissions when you create a model: add, change, delete, and view. Users are able to add, delete or alter instances of the model, accordingly, using these permissions.

I have implemented the user signup view, login and logout views using the built-in authentication. Different parts of the application is restricted based on authentication. There are two types of authentication templates are created one [named "base_auth"] take care for navigation between pages tagged as authenticated and other one [named "base_normal"] takes the navigation between unauthenticated pages.

The bifurcation of pages between authenticated users and unauthenticated users is as follows:

Unauthenticated View - index page, login page, signup page.

Authenticated view - Home page, profile page, create post, update post, delete post, created profile, update profile, friends, search friends, accept friend, delete friend, chat home, chat room, space create, space, join

Forms

Django comes with powerful form feature that handle the user input from the web application. It have builtin functionality of defining fields, rules and validation on submitted data from the frontend. It help in receiving the consistent and cleaned data for database. Having predefined rules on forms also provide consistency. These form can be used as HTML form tag and could be render from the backend. The list of forms defined in the application and its use is provided in the table below:

Social App:

```
class UserSignupForm(forms.ModelForm):
```

- Form to register a new user in the data base
- HTMX for dynamic navigation from login to user home page without loading Page

```
self.helper.attrs = {  
    # Redirecting on the save to signup page again  
    "hx-post": reverse_lazy("signup"),  
    # To replace/swap the form with the information returned by django  
    "hx-target": "#signup-form",  
    # Ajax swap to replace the outer HTML (Avoiding placing html inside  
    the target - form inside form)
```

```
- "hx-swap": "innerHTML",
- }
```

- CSS class is added to the button to be displayed on form

```
self.helper.add_input(
    Submit(
        "submit",
        "Submit",
        # Tailwind CSS class reference for button
        css_class="bg-transparent hover:bg-blue-500 text-blue-700
font-semibold hover:text-white py-2 px-4 border border-blue-500
hover:border-transparent rounded",
    )
)
```

- Password fields are defined to check that user password and repeat passwords are matching with each other and error messages are raised.

```
Function to validate username before saving to database
"""

def clean_username(self):
    username = self.cleaned_data["username"]
    if len(username) <= 3:
        raise forms.ValidationError("Username is too short")
    return username

def clean_password2(self):
    password1 = self.cleaned_data.get("password1")
    password2 = self.cleaned_data.get("password2")

    if password1 and password2 and password1 != password2:
        raise forms.ValidationError("Passwords does not match")
    return password2
```

- Password is hashed in this form before saving to the database

```
class UserLoginForm(forms.Form):
```

- Form to login into the social web application.
- Errors are identified using the form "Validation Error" class.
- HTMX for dynamic navigation from login to user home page without loading page

```
class UserProfileForm(forms.ModelForm):
```

- Form to login show user profile.
- Errors are identified using the form "Validation Error" class.
- Function is provided to validate birth date is on/before today date

```
    def clean_birthdate(self):
        birthdate = self.cleaned_data["birthdate"]
        if birthdate >= datetime.date.today():
            raise forms.ValidationError("Please enter date before today's
date")
        return birthdate
```

```
class PostForm(forms.ModelForm):
```

- Form for submitting the post
- Validation provided to accept only images as input.

```
    image = forms.ImageField(
        required=False,
        error_messages={"invalid": "Image files only"},
        widget=forms.FileInput,
    )
```

- HTMX for dynamic web after submitting the post

```
self.helper = FormHelper(self)
self.helper.form_id = "post-form"
# Dictionary to add the extra attributes to the crispy form
self.helper.attrs = {
    # Redirecting on the save to signup page again
    "hx-post": reverse_lazy("home"),
    # To replace/swap the form with the information returned by
django
    "hx-target": "#post-form",
    # Ajax swap to replace the outer HTML (Avoiding placing html
inside the target - form inside form)
    "hx-swap": "outerHTML",
}
```

Friends App:

There are no form created in friend app because this is all handled using the request fields defined in view and it only require a connection between users without submitting any additional information.

Chat App:

There are no backend django forms are used in the chat application instead frontend forms are used with HMTX integration for communication between the users.

```
<div hx-ws="connect:/chat/{{ slug }}/">
  <form hx-ws="send:submit">
    <div >
      <input name='message' id='message' rows="1" placeholder="Your
        message..."></input>
      <button type="submit">
        <span class="sr-only">Send message</span>
      </button>
    </div>
  </form>
</div>
```

Please note that CSS and SVG classes are removed from the code above to make it more presentable.

hx-ws is an attribute in htmx that allows you to work with WebSockets directly from HTML. The value of the attribute can be one or more of the following, separated by commas: connect:<url> or connect:<prefix>:<url> - A URL to establish a WebSocket connection [Source: <https://htmx.org/attributes/hx-ws/>]

Crispy Forms:

I have installed django crispy form in the application to further management django forms. One of the benefits of using the crispy form is to amend the properties of django form such as submit button, and button ss classes without rewriting these in the template file. Moreover, tailwind-crispy is also installed to use the tailwind classes for styling crispy form components.

Example - Form Validations

Invalid username or password

Username*

test

Password*

Login

First name

Ahmad

Last name

Ahmad

Username*

Ahmad

A user with that username already exists.

Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password*

Enter the password

Repeat Password*

Enter the password

Passwords does not match

Submit

Already have account? [Login](#)

Chat Logic/Functionality:

Chat bot is created using a ASGI server, following technologies as briefed before in the project are used to implement the realtime (ASGI) chat in the application:

- Django-channels
- daphne
- HTMX

Please note that dphane should be defined as first app in the setting.py.

```
INSTALLED_APPS = [  
    # Web server gateway interface for asynchronous web applications.  
    "daphne",  
    "django.contrib.admin",
```

```

        "Django.contrib.auth",
        ..... ,
        ..... ,
        # for Chat app
        "channels",
    ]

```

```

ASGI_APPLICATION = "socialweb.asgi.application"

```

```

from chat import routing

application = ProtocolTypeRouter(
    {
        "http": get_asgi_application(),
        "websocket":
AuthMiddlewareStack(URLRouter(routing.websocket_urlpatterns)),
    }
)

```

Channels, a wrapper around Django ASGI that enables us to incorporate additional asynchronous protocols for web sockets, Objective is to establish long-running connections.

Hence, in order to use the WebSocket protocol in the chat application, I have opted to use Django channels module. WebSocket is a communication protocol (set of guidelines and requirements to be fulfilled) that enables message sending and receiving between the client and server (communication).

There are options to use redis, in-memory, however, since awesome social web is not large-scale or production-scale app, therefore, i have decided to use daphne. There is also other kinds of backend services available such as RabbitMQ, KAFKA, google cloud pub/sub etc.

Django official documents are extensively referred to write routers and webconsumers for chat app, There is chat consumer created, which from the AsyncWebsocketConsumer provided by the channels.generic.websocket module.

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

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

        # Adding the users on connecting the rooms
        await self.add_user(self.room_name, self.user)

        await self.accept()

    # To remove the users when they exit the chat
    async def disconnect(self, close_code):
        await self.remove_user(self.room_name, self.user)
        await self.channel_layer.group_discard(self.room_group_name,
self.channel_name)

    async def receive(self, text_data):
        text_data_json = json.loads(text_data)
        message = text_data_json["message"]
        user = self.user
        username = user.username
        room = self.room_name

        # Save conversation between the users
        await self.save_message(room, user, message)

        await self.channel_layer.group_send(
            self.room_group_name,
            {
                "type": "chat_message",
                "message": message,
```

```

        "username": username,
    },
)

async def chat_message(self, event):
    message = event["message"]
    username = event["username"]

message_html = f"<div hx-swap-oob='beforeend:#messages'><p><b>{username}</b>:
{message}</p></div>"
    await self.send(
        text_data=json.dumps(
            {
                "message": message_html,
                "username": username,
            }
        )
    )
)

```

This consumer class has let me define the following functionalities:

- Connecting with the required space name after accepting
- Messages sent to and received from the group/space.
- Removing the user from the room or group and terminating the WebSocket session

When the client creates a websocket session, the function connect is invoked. When that occurs, the function stores the string value for room name and obtains the space slug from the client's URL. By adding the chat_ string with the space name, it produces a new variable called room group name. It also retrieves the user who is presently logged in from the request. The channel name is subsequently added to the room group name group. The connection's or consumer's specific identification in the channel is channel name. The consumer can publish the text message to all of the channels in the group by providing the channel name. After the function acknowledges the connection, a web connection is created on both sides, a connection request is made from the client, and the connection is now complete.

The disconnect method, which essentially removes the channel name from the group, i.e. the space name or whatever the string it may be, is triggered when the client submits a terminate connection query to the server. So, since

the websocket has been closed on both ends, it effectively eliminates the client from the broadcast channel and prevents it from receiving or sending text messages through it.

Installed Apps

The list of additional applications in django project in addition to the pre available applications is as follows:

```
INSTALLED_APPS = [  
    # Web server gateway interface for asynchronous web applications.  
    "daphne",  
    "django.contrib.admin",  
    "django.contrib.auth",  
    "django.contrib.contenttypes",  
    "django.contrib.sessions",  
    "django.contrib.messages",  
    "django.contrib.staticfiles",  
    # "django_browser_reload",  
    # Crispy Forms  
    "crispy_forms",  
    "crispy_tailwind",  
    # Django Extension:  
    "django_extensions",  
    # Rest Framework  
    "rest_framework",  
    # Django Countries for model  
    "django_countries",  
    # Social App within Social web project  
    "socialapp",  
    # Friends App within Social web project  
    "friends",  
    # chat App within Social web project  
    "chat",  
    # to clear the data already inserted in tables without destroying  
    # structure  
    "django_truncate",  
    # for Chat app
```

```
"channels",  
# for Apis  
"api",  
]
```

Note: Django browser reload is commented out because it started creating issues on installing the depene ASGI server for chat app. Socket connection was getting disconnected because of frequent reload calls.

User - Precreated / Test Database

The list of users already created in the database and their purpose are:

User Name	User id	Password	Purpose
admin	admin	admin	Its a super user created to control other user and other functions
Ahmad	Ahmad	Ahmad	Normal user created for testing purpose
UOL1	UOL1	UOL1	Created to submit posts and send friend request amount the user in application and also to use the chat between users
UOL3	UOL3	UOL3	Created to submit posts and send friend request amount the user in application and also to use the chat between users

admin may be used to login to Django admin via URL: <http://localhost:8000/admin> to view database content and to make or incorporate any modifications.

Tests

The test created for teh application are defined in “tests.py” file within API app. There are mix of different test defined to check the functionality of application and parameters for model data acceptance.

```

class ProfileModelTestCase(TestCase):
    def setUp(self):
        self.user = User.objects.create_user(username="testuser",
        email="testuser@example.com", password="testpass")
        self.profile = Profile.objects.create(
            user=self.user, country="US", birthdate=timezone.now().date(),
        image="images/test.jpg"
        )

    def test_profile_creation(self):
        """Test that a profile can be created"""
        self.assertIsInstance(self.profile, Profile)

    def test_profile_str(self):
        """Test that the profile string representation is correct"""
        expected = str(self.user.username)
        self.assertEqual(str(self.profile), expected)

    def test_profile_friends(self):
        """Test that friends can be added to a profile"""
        friend = Profile.objects.create(
            user=User.objects.create_user(username="frienduser",
        email="frienduser@example.com", password="friendpass"),
            country="CA",
            birthdate=timezone.now().date(),
            image="images/friend.jpg",
        )
        self.profile.friends.add(friend)
        self.assertIn(friend, self.profile.friends.all())

    def test_profile_friends_symmetry(self):
        """Test that the friendship relationship is not symmetrical"""
        friend = Profile.objects.create(
            user=User.objects.create_user(username="frienduser",
        email="frienduser@example.com", password="friendpass"),
            country="CA",
            birthdate=timezone.now().date(),
            image="images/friend.jpg",

```

```

    )
    self.profile.friends.add(friend)
    self.assertNotIn(self.profile, friend.friends.all())

class PostModelTest(TestCase):
    @classmethod
    def setUpTestData(cls):
        # Set up non-modified objects used by all test methods
        test_user = User.objects.create_user(username="testuser",
password="testpass")
        Post.objects.create(owner=test_user, text="This is a test post")

    def test_owner_label(self):
        """Test that a owner label can be created"""
        post = Post.objects.get(id=1)
        field_label = post._meta.get_field("owner").verbose_name
        self.assertEqual(field_label, "owner")

    def test_text_max_length(self):
        """Test the length of maximum number of characters"""
        post = Post.objects.get(id=1)
        max_length = post._meta.get_field("text").max_length
        self.assertEqual(max_length, 140)

```

Git/ Version Control

Although I haven't yet released any versions, I utilized Git to record all of the codebase's modifications. There are five branches created (Users, Posts, Friends, Chat and Clean up) and these are later merged with the main branch. Run git log in the terminal to get a list of my changes.

Dependencies management

I have used poetry for dependency management. Poetry is a Python dependency management tool that helps manage dependencies, packages, and libraries in your Python project. It simplifies your project by resolving dependency

complexities and managing installation and updates for you [source: [Using Poetry Dependency Management tool in Python - GeeksforGeeks](#)].

Though, requirements.txt file is also kept in the folder for someone who is not familiar with poetry and wants to run the application in the conventional way using pip.

Application Start / Dependencies Installion

1. Go to the project directory

Terminal Command: `cd socialweb`

2. Install dependencies

Terminal Command: `poetry install`

3. Start the server

Terminal Command: `poetry run python manage.py runserver`

Database file is also submitted along with the project to keep it simple testing, however, new user/data can be created.

Note: It is assumed that poetry dependency management tool is already installed in the system

Linting Tool

I have used `black` and `isort` for linting in the vscode, the line length for the code in black was defined as 120 characters.

Project Completion Checklist

R1: The application contains the functionality required

- a) ☒ Users can create accounts
- b) ☒ Users can log in and log out
- c) ☒ Users can search for other users
- d) ☒ Users can add other users as friends

- e) ☒ Users can chat in realtime with friends
- f) ☒ Users can add status updates to their home page
- g) ☒ Users can add media (such as images to their account and these are accessible via their home page
- h) ☒ correct use of models and migrations
- i) ☒ correct use of form, validators and serialisation
- j) ☒ correct use of django-rest-framework
- k) ☒ correct use of URL routing
- l) ☒ appropriate use of unit testing
- m) ☒ An appropriate method for storing and displaying media files is given

R2: ☒ Implements and appropriate database model to model accounts, the stored data and the relationships between account

R3: ☒ Implementation of appropriate code for a REST interface that allows users to access their data

R4: ☒ Implementation appropriate tests for the server side code

C1: ☒ Code is clearly organised into appropriate files (i.e. view code is placed in an appropriate view.py or api.py file, models are placed in an appropriate models.py file)

C2: ☒ Appropriate comments are included to ensure the code is clear and Readable (Files are thoroughly commented both front and back end)

C3: ☒ Code is laid out clearly with consistent indenting, ideally following python pep8 standard (Linting tool used to ensure pep 8 standards)

C4: ☒ Code is organised into appropriate functions with clear, limited Purpose (Modularity is ensure through front end and back end with proper file structure)

C5: ☒ Functions, classes and variables have meaningful names, with a consistent naming style (Consistent naming is used in classes also in front end)

C6: ☒ Appropriate tests to cover the API functionality are provided

Word Count: 4866 (Meets the requirements of having word count between 4000-6000)

Youtube Video Link: <https://www.youtube.com/watch?v=7oJNT5yFGK0>