

CM3035 Advanced Web Development
Coursework 2 Report

LIM WEE KIAT

Table of Contents

Introduction	Page 4
R1: The application contains the functionality requires:	Page 5 - Page 27
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 accounts.	Page 28 - Page 30
R3: Implementation of appropriate code for a REST interface that allows users to access their data.	Page 31 - Page 34
R4: Implementation appropriate tests for the server side code.	Page 35

Introduction

In this assignment, I need to develop a Social Network web application. The term social networking refers to the use of internet-based social media sites to stay connected with friends, family, colleagues, customers, or clients. In this case, I will develop a social network web application with some functions which are communicate with each other, register account, login account, search function, add friends, post status and so on.

To successfully implement the application, I have used the Django to develop the application. Django is a high-level Python web framework for the rapid development of secure and maintainable websites. Django handles most of the hassle of web development, so I can focus on writing applications without reinventing the wheel. It is open-source software with a thriving, active community, excellent documentation, and many free and paid support options.

Besides, I have used the Bootstrap to create user interface in web applications. Bootstrap is a free front-end framework for faster and easier web development. Bootstrap includes HTML and CSS based typography, forms, buttons, tables, navigation, modals, image carousels and many other design templates, with optional JavaScript plugins. Bootstrap also enables you to easily create responsive designs.

Furthermore, communication between client and server is managed using WebSockets (via Django Channels), and every time a user is authenticated, an event is broadcast to all other connected users. Each user's screen changes automatically without reloading the browser.

R1: The application contains the functionality requires

Before doing anything else, I need to create a virtual environment and install Django. After that, activate the virtual environment and start a new Django project called 'socialnetwork'. After creating the Django project, create two new Django app called 'social' and 'landing'. For the 'landing' app, it have all guest pages for an unregistered user showing the site. For the 'social' app, which will hold all of our actual network views and urls. I need to register the 'social' and 'landing' app to INSTALLED_APPS in [socialnetwork/settings.py]. Besides, we also need to create a super user who can login to the admin site.

(a) Users can create accounts

First and foremost, I use allauth Django as my user authentication. Django-allauth is a reusable Django application that allows registration with social accounts (i.e. Facebook, Google, Twitter, Instagram). Basically, Django-allauth will allow you to quickly handle and create registration flows instead of using Django's built-in UserCreationForm. Besides, Django-crispy-forms is an application to help manage Django forms. To use the allauth Django, we need install django-allauth by using the command 'pip install django-allauth'. Then add the django-allauth and crispy_forms to the INSTALLED_APPS in settings.py and include multiple authentication backends. We add a redirect URL to specify where the user will be redirected after logging in with social authentication as shown in figure 1.0.1.

```
33 AUTHENTICATION_BACKENDS = [  
34     # Needed to login by username in Django admin, regardless of `allauth`  
35     'django.contrib.auth.backends.ModelBackend',  
36  
37     # `allauth` specific authentication methods, such as login by e-mail  
38     'allauth.account.auth_backends.AuthenticationBackend',  
39 ]  
40  
41 INSTALLED_APPS = [  
42     'social',  
43     'landing',  
44  
45     'crispy_forms',  
46     'allauth',  
47     'allauth.account',  
48     'allauth.socialaccount',  
49  
50     'django.contrib.admin',  
51     'django.contrib.auth',  
52     'django.contrib.contenttypes',  
53     'django.contrib.sessions',  
54     'django.contrib.messages',  
55     'django.contrib.staticfiles',  
56     'django.contrib.sites',  
57 ]  
58  
59 SITE_ID = 1  
60  
61 LOGIN_REDIRECT_URL = 'post-list'
```

Figure 1.0.1 authentication_backend (settings.py)

Besides, add the allauth.urls in socialnetwork/urls.py and migrate the app as shown in figure 1.0.2.

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('landing.urls')),
    path('accounts/', include('allauth.urls')),
    path('social/', include('social.urls')),
]
```

Figure 1.0.2 allauth.urls(socialnetwork/urls.py)

For the design of the signup page, we downloaded the templates folder from Github website and edit them by self. Now we modify our settings file so we need to tell Django that's where our templates are at as shown in figure 1.0.3.

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    ],
]
```

Figure 1.0.3 Edit templates default(settings.py)

In this case, we use the signup page provided in the templates folder as the registration page for our socialnetwork application. We add the link into our index.html so users can click the register button then it will redirect to the sign up page as shown in figure 1.0.4.

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

{% block content %}
<div class="container">
  <div class="row justify-content-center mt-5">
    <div class="col-md-10 col-sm-12 text-center">
      <h1 class="display-2">Connect With Your Friends</h1>
      <p class="mt-3 lead">Follow people who interest you, stay up to date on the latest news and join
    <div class="d-flex justify-content-center mt-5">
      <a href="{% url 'account_login' %}" class="btn btn-light mr-2">Log In</a>
      <a href="{% url 'account_signup' %}" class="btn btn-dark">Register</a>
    </div>
    </div>
  </div>
</div>
{% endblock content %}
```

Figure 1.0.4 Index.html

Now, the socialnetwork application can let user to create new accounts as shown in figure 1.0.5.

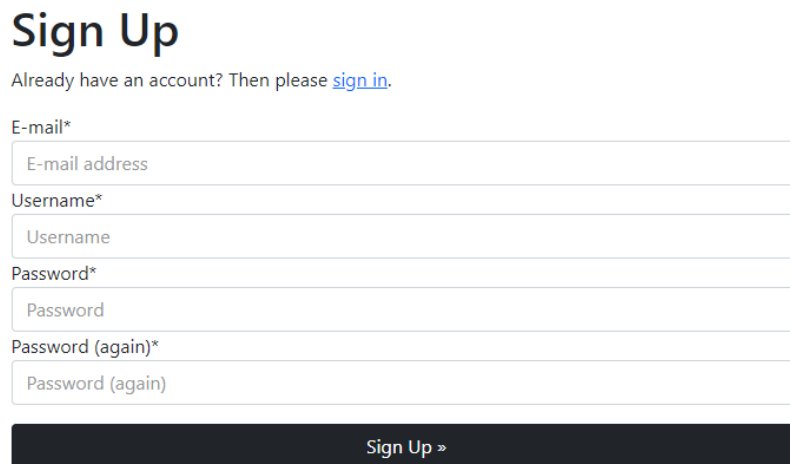
The image shows a 'Sign Up' form. At the top, the title 'Sign Up' is in a large, bold, dark blue font. Below it, a smaller line of text says 'Already have an account? Then please [sign in.](#)'. The form consists of five input fields, each with a label to its left: 'E-mail*' (with 'E-mail address' as placeholder text), 'Username*' (with 'Username' as placeholder text), 'Password*' (with 'Password' as placeholder text), and 'Password (again)*' (with 'Password (again)' as placeholder text). The last field is empty. At the bottom of the form is a dark blue button with the text 'Sign Up »' in white.

Figure 1.0.5 SignUp page

(b) Users can log in and log out

For the log in and log out function, we also use the allauth Django to do it. Django comes with a user authentication system. It handles user accounts, groups, permissions, and cookie-based user sessions. The auth system consists of:

- Users
- Permissions: Binary (yes/no) flags designating whether a user may perform a certain task.
- Groups: A generic way of applying labels and permissions to more than one user.
- Messages: A simple way to queue messages for given users.

Besides, we use the login.html and logout.html that in the templates folder. For the login.html, user can login with their username and password. If the user haven't create any account, there have a link that can redirect to the sign up page. If the user has forgotten their password, there will also be a forgot password link as shown in figure 1.0.6.

Sign In

If you have not created an account yet, then please [sign up](#) first.

Username*

Password*

☐ Remember Me

[Forgot Password?](#)

Sign In

Figure 1.0.6 Login page

For the log out page, we add the url of the log out page in our navbar.html to allow the user to logout successfully. A confirmation notification is sent to the user when they attempt to log out of their account. After the user click the sign out, it will logout the account and redirect to the index.html as shown in figure 1.0.7.

Successfully signed in as admin.

Sign Out

Are you sure you want to sign out?

Sign Out

Figure 1.0.7 Logout confirm notification

(c) Users can search for other users

We make the user search in our navbar functional. First and foremost, we add a name and a value on the input field in navbar.html. We put in a name called 'query' and give it a value and pass in "{{ request.GET.query }}" . We are doing here is we are setting the whatever we type into this field, we are sending it the name to equal query. So in our url it will say query equals whatever we typed into this field and the value attribute will just take if there is a query as you enter url, it will take that value and pre-populate the input field with that value.

Besides, we need to add an actual search view that will be used to render a template that will list out all the profile that are found in the search. In views.py, we created a class called 'UserSearch' and inherit from our generic view class and then we will add a get method. In the get method, we declare a variable called 'query' to get the query that we search for. We also need to filter out by the profiles with declare a variable called 'profile_list' and equal to 'UserProfile.objects.filter'. Now we want to pass in the query object. So Django has this useful tool which called Q and put some parentheses to figure out some sort of filter parameter. In this case, we want to find if the username on the profile matches what was searched, we can type 'user__username__icontains=query'. So if anything in the query contains something that matches this username then we want to return it inside this profile list. Besides, we need to create a context variable and pass our profile list and render that all to a template as shown in figure 1.0.8.

```
class UserSearch(View):
    def get(self,request,*args,**kwargs):
        query = self.request.GET.get('query')
        profile_list = UserProfile.objects.filter(
            Q(user__username__icontains=query)
        )

        context = {
            'profile_list':profile_list,
        }

        return render(request,'social/search.html',context)
```

Figure 1.0.8 UserSearch View(views.py)

Besides, we create a url pattern for the UserSearch view by creating a path and passing our url which this case will be 'search/'. After that, we created a 'search.html' and design the layout. We use a for loop that will looping all the profiles that we found that match the query and list those all out as shown in figure 1.0.9.

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

{% block content %}
<div class="container">

  <div class="row mt-5">
    <div class="col-md-5 col-sm-6">
      <a href="{% url 'post-list' %}" class="btn btn-dark mt-3">Back To Feed</a>
    </div>
  </div>

  {% for profile in profile_list %}
    <div class="row justify-content-center mt-3">
      <div class="col-md-5 col-sm-12 border-bottom position-relative">
        <div>
          <a href="{% url 'profile' profile.pk %}">
            
          </a>

          <p style="padding-top: 0.5rem;">
            <a style="text-decoration:none" class="text-primary" href="{% url 'profile' profile.pk %}">@{{ profile.user }}</a>
          </p>
        </div>

        {% if profile.location %}
          <p>{{ profile.location }}</p>
        {% endif %}
        <p>Friends: {{ profile.friends.all.count }}</p>
      </div>
    </div>
  {% endfor %}
</div>
</div>
```

Figure 1.0.9 Search.html

Lastly, now we can search the user as shown in figure 1.1.0.

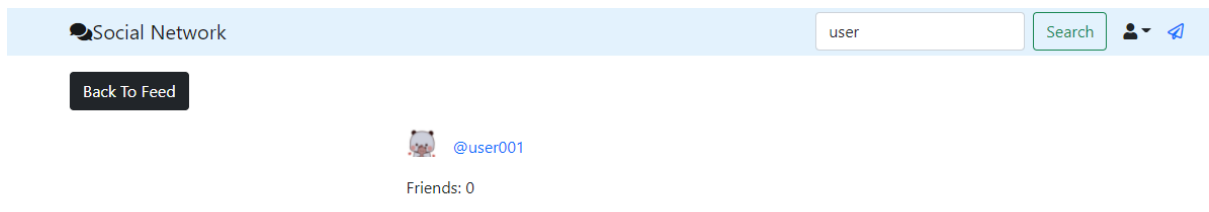


Figure 1.1.0 Search user result

(d) Users can add other users as friends

First and foremost, we created our UserProfile model and add a field called 'friends' where we can hold friends and kind of store them on the field. After adding the field, we need to make migrations and migrate the changes to the database. Besides, we put another link on our username that will go to the profile in order to add or remove friends in our post_list.html. We created 'AddFriend' class in our views.py and we pass in 'LoginRequiredMixin' and use our generic view class for this class so we can add a post methods to handle http request. In the post method, we created a variable called 'profile' to get the profile that we are on. To add friends' many to many field, Django makes it very easy with just an add method so we can use friends.add to add to the friends field. We also use redirect to go back to that profile view that we were just on as shown in figure 1.1.1. We also created a RemoveFriend class and this will be almost exactly the same with the AddFriend class, just change the add function to remove function only as shown in figure 1.1.2.

```
class AddFriend(LoginRequiredMixin,View):
    def post(self,request,pk,*args,**kwargs):
        profile = UserProfile.objects.get(pk=pk)
        profile.friends.add(request.user)

        return redirect('profile',pk=profile.pk)
```

Figure 1.1.1 AddFriend view(views.py)

```
class RemoveFriend(LoginRequiredMixin,View):
    def post(self,request,pk,*args,**kwargs):
        profile = UserProfile.objects.get(pk=pk)
        profile.friends.remove(request.user)

        return redirect('profile',pk=profile.pk)
```

Figure 1.1.2 RemoveFriend view(views.py)

Furthermore, we add the url pattern for the AddFriend and RemoveFriend view in our urls.py as shown in figure 1.1.3.

```
path('profile/<int:pk>/friends/add',AddFriend.as_view(),name='add-friends'),
path('profile/<int:pk>/friends/remove',RemoveFriend.as_view(),name='remove-friends'),
```

Figure 1.1.3 Url pattern (urls.py)

To actually show the number of friends on the profile, we need to add some logic in our ProfileView to check how many friends there are. We use all method to get every object that is in this many to many field. Now we have all our friends here, we can count how many there are and we can use the len method to get length of this list. Besides, we want to have some sort of boolean value that shows whether or not this user is currently adding the user on this profile, if it is, we want to go ahead and pass in it is adding boolean to the context as well and we will use that to show either a add

or remove button based on if they are currently adding it or not as shown in figure 1.1.4.

```
class ProfileView(View):
    def get(self, request, pk, *args, **kwargs):
        profile = UserProfile.objects.get(pk=pk)
        user = profile.user
        posts = Post.objects.filter(author=user).order_by('-created_on')

        friends = profile.friends.all()

        if len(friends) == 0:
            is_adding=False

        for friend in friends:
            if friend == request.user:
                is_adding = True
                break
            else:
                is_adding = False

        number_of_friends = len(friends)

        context = {
            'user':user,
            'profile':profile,
            'posts':posts,
            'number_of_friends': number_of_friends,
            'is_adding':is_adding,
        }
```

Figure 1.1.4 Profile view(views.py)

After that, we update the profile templates to show the button in the profile. Now user can add other friends as shown in figure 1.1.5.

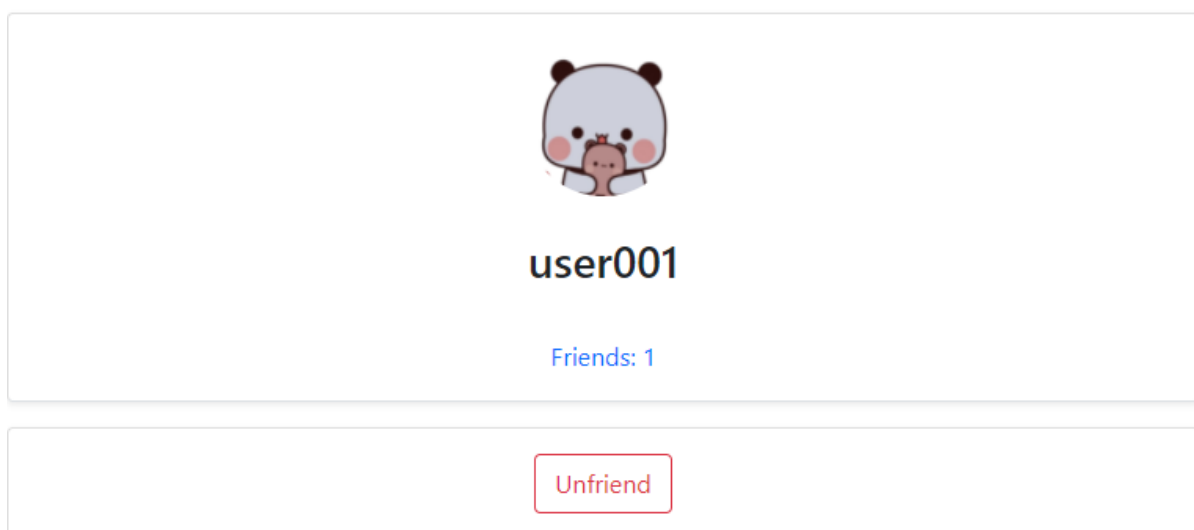


Figure 1.1.5 Add other user as friend

(e) Users can chat in realtime with friends

Firstly, we need two new models, we need a ThreadModel to hold all the messages between the two users. Then we will have a MessageModel to hold the data for a single message. On the threading model, we need to save the user, the receiving user, and a boolean to determine if there are any unread messages. For the message model, we'll save the thread it's attached to, the sending user, the receiving user, the body, the image, the date sent, and a boolean value of whether it was read as shown in figure 1.1.6.

```
class ThreadModel(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE, related_name='+')
    receiver = models.ForeignKey(User, on_delete=models.CASCADE, related_name='+')

class MessageModel(models.Model):
    thread = models.ForeignKey('ThreadModel', related_name='+', on_delete=models.CASCADE, blank=True, null=True)
    sender_user = models.ForeignKey(User, on_delete=models.CASCADE, related_name='+')
    receiver_user = models.ForeignKey(User, on_delete=models.CASCADE, related_name='+')
    body = models.CharField(max_length=1000)
    image = models.ImageField(upload_to='uploads/message_photos', blank=True, null=True)
    date = models.DateTimeField(default=timezone.now)
    is_read = models.BooleanField(default=False)
```

Figure 1.1.6 Thread Model and Message Model

Besides, we need to create two forms, one will create a thread where we need to input a username of the person we want to talk to, and another one is message thread that will have a text box for the message as shown in figure 1.1.7.

```
class ThreadForm(forms.Form):
    username = forms.CharField(label='', max_length=100)

class MessageForm(forms.ModelForm):
    body = forms.CharField(label='', max_length=1000)
    image = forms.ImageField(required=False)

    class Meta:
        model = MessageModel
        fields = ['body', 'image']
```

Figure 1.1.7 Thread Form and Message Form

In addition, we need to create four views which are CreateThread view, List Thread view, CreateMessage view, and a thread view. These will handle all the basic functionality of direct messages. There are two methods on the CreateThread view, one is the get method, which displays the form to enter the username. There will also be a post method to handle creating the thread. To create a thread, we want to take what was entered in the form and see if there is a user that matches that username. If not, we'll stop here. If there are users, we will check if a thread already exists between those users, if there is, we will not create one, we will open the thread. And if there is no thread, we will create one and redirect it to that new thread as shown in figure 1.1.8.

```

class CreateThread(View):
    def get(self, request, *args, **kwargs):
        form = ThreadForm()

        context = {
            'form': form
        }

        return render(request, 'social/create_thread.html', context)

    def post(self, request, *args, **kwargs):
        form = ThreadForm(request.POST)

        username = request.POST.get('username')
        try:
            receiver = User.objects.get(username=username)
            if ThreadModel.objects.filter(user=request.user, receiver=receiver).exists():
                thread = ThreadModel.objects.filter(user=request.user, receiver=receiver)[0]
                return redirect('thread', pk=thread.pk)
            elif ThreadModel.objects.filter(user=receiver, receiver=request.user).exists():
                thread = ThreadModel.objects.filter(user=receiver, receiver=request.user)[0]
                return redirect('thread', pk=thread.pk)

            if form.is_valid():
                thread = ThreadModel(
                    user=request.user,
                    receiver=receiver
                )
                thread.save()

                return redirect('thread', pk=thread.pk)
        except:
            messages.error(request, 'Invalid username')
            return redirect('create-thread')

```

Figure 1.1.8 Create Thread View(views.py)

For the ListThread view, this will act as an inbox where we can see all our conversations. All we need is a get method that will get all the threads where the logged-in user is the sending user or the receiving user as shown in figure 1.1.9.

```

class ListThreads(View):
    def get(self, request, *args, **kwargs):
        threads = ThreadModel.objects.filter(Q(user=request.user) | Q(receiver=request.user))

        context = {
            'threads': threads
        }

        return render(request, 'social/inbox.html', context)

```

Figure 1.1.9 ListThread View(views.py)

For the CreateMessage view, this will require a post method that creates the message. We don't need the get method because our form will be displayed in a ThreadView which will display the dialog. We just need to send a post request to this view to create the message and then redirect back to the ThreadView as shown in figure 1.2.0.

```
class CreateMessage(View):
    def post(self,request,pk,*args,**kwargs):
        form = MessageForm(request.POST,request.FILES)
        thread = ThreadModel.objects.get(pk=pk)
        if thread.receiver == request.user:
            receiver = thread.user
        else:
            receiver = thread.receiver

        if form.is_valid():
            message = form.save(commit=False)
            message.thread = thread
            message.sender_user=request.user
            message.receiver_user = receiver
            message.save()

            # message = MessageModel(
            #     thread = thread,
            #     sender_user = request.user,
            #     receiver_user = receiver,
            #     body = request.POST.get('message')
            # )

            #message.save()
            return redirect('thread',pk=pk)
```

Figure 1.2.0 CreateMessage View(views.py)

For the ThreadView view, it will show all messages in a thread and it will display a form at the bottom to send a new message as shown in figure 1.2.1.

```
class ThreadView(View):
    def get(self,request,pk,*args,**kwargs):
        form = MessageForm()
        thread = ThreadModel.objects.get(pk=pk)
        message_list = MessageModel.objects.filter(thread__pk__contains=pk)
        context = {
            'thread':thread,
            'form':form,
            'message_list':message_list
        }

        return render(request,'social/thread.html',context)
```

Figure 1.2.1 ThreadView View(views.py)

After created the views, we need to create our html templates for each of them. In this case, we need the create thread with the form which we will call 'create_thread.html'. We also created a 'inbox.html ' for the ListThread template. We don't need a template for the CreateMessage view, so the last view we need to create a template for is ThreadView. We need to place the message on the left or right side of the screen depending on the sender, if the logged in user is the user who sent the message, the message will be displayed on the left, otherwise it will be displayed on the right. Then at the bottom, we will display our 's create message form. After that, we need to add url paths for each of our views as shown in figure 1.2.2.

```
path('inbox/',ListThreads.as_view(),name='inbox'),  
path('inbox/create-thread/',CreateThread.as_view(),name='create-thread'),  
path('inbox/<int:pk>/',ThreadView.as_view(),name='thread'),  
path('inbox/<int:pk>/create-message/',CreateMessage.as_view(),name='create-message')
```

Figure 1.2.2 url path(urls.py)

Lastly, we add the the inbox link to our navbar and now users can chat with friends as shown in figure 1.2.3.

@ user001

Hi

hi



nice to meet you

Figure 1.2.3 Example chat room

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

In this case, we will add a social feed where posts can be viewed and a form where new posts can be added. First and foremost, we need to create our Post model in our social/models.py as shown in figure 1.2.4.

```
class Post(models.Model):
    body = models.TextField()
    image = models.ManyToManyField('Image', blank=True)
    created_on = models.DateTimeField(default=timezone.now)
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    likes = models.ManyToManyField(User, blank=True, related_name='likes')
    dislikes = models.ManyToManyField(User, blank=True, related_name='dislikes')
```

Figure 1.2.4 Post model

We are just creating a simple data model here, where we have a body of text, the date of the post was created, the user who wrote the post, likes and dislikes. After created the Post Model, now we can create a view for our social feed. This view will first list all posts when sending a get request, and also have a form at the top for submitting new posts as shown in figure 1.2.5.

```
class PostListView(LoginRequiredMixin, View):
    def get(self, request, *args, **kwargs):
        # logged_in_user = request.user
        # posts = Post.objects.filter(
        #     author__profile__followers__in=[logged_in_user]
        # ).order_by('-created_on')
        posts = Post.objects.all().order_by('-created_on')
        form = PostForm()

        context = {
            'post_list': posts,
            'form': form,
        }
        return render(request, 'social/post_list.html', context)
```

Figure 1.2.5 PostListView (views.py)

Now we need to pass the request.POST data to our form and we can check if the form is valid. If it is, we can set the author as the currently logged in user and save the post. Then we can render the exact same template and it will show the new post as shown in figure 1.2.6.

```

def post(self,request,*args,**kwargs):
    posts = Post.objects.all().order_by('-created_on')
    form = PostForm(request.POST,request.FILES)
    files=request.FILES.getlist('image')

    if form.is_valid():
        new_post = form.save(commit=False)
        new_post.author = request.user
        new_post.save()

        for f in files:
            img = Image(image=f)
            img.save()
            new_post.image.add(img)

        new_post.save()

    context = {
        'post_list':posts,
        'form':form,
    }
    return render(request,'social/post_list.html',context)

```

Figure 1.2.6 PostListView (views.py)

In PostListView, we get all posts and sort by newest first. Besides, we created a forms.py file in the social app to create the PostForm that will be the form to create a new post as shown in figure 1.2.7.

```

class PostForm(forms.ModelForm):
    body = forms.CharField(
        label='',
        widget=forms.Textarea(attrs={
            'rows':'3',
            'placeholder': 'Say something...'
        }))

    image = forms.ImageField(
        required=False,
        widget=forms.ClearableFileInput(attrs={
            'multiple': True
        }))

    class Meta:
        model = Post
        fields = ['body']

```

Figure 1.2.7 PostForm(forms.py)

We use the `ModelForm` object for our `PostForm` because the `ModelForm` will allow us to create forms based on models. In this case, we will create it based on our `Post` model. In the `Meta` subclass, we can define the information we need, which will be what model we want this form to be used for, and what fields. We only want the body text to be a field in the form, so in our field variable, we create a list and just put "body" in it. This will create a text area input that defaults to 3 lines and some placeholder text.

After that, we need to create an html template for our `PostForm` view called 'post_list.html'. We can reuse the landing base template. At the top, we are putting in the form and using the `crispy forms` to make it look nice. We can use a `for` loop to iterate through the posts and display each one. Lastly, we created a url route for our `PostListView` view in our `social/urls.py`. Now users can add status updates to their home page as shown in figure 1.2.8.

Add a post!


Say something...

Image


Choose Files


No file chosen

POST

 @admin March 4, 2022, 6:58 a.m.

Hi, Nice to meet you all



 0


 0

Figure 1.2.8 Output of add status updates

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

In this case, we will add a button to select a file, and we will display it below the text of the post. Firstly, we need to add another model to the image. Then we can add a ManyToMany field. This will allow us to store multiple images as shown in figure 1.2.9.

```
class Image(models.Model):
    image = models.ImageField(upload_to='uploads/post_photos',blank=True,null=True)

class Post(models.Model):
    body = models.TextField()
    image = models.ManyToManyField('Image',blank=True)
    created_on = models.DateTimeField(default=timezone.now)
    author = models.ForeignKey(User,on_delete=models.CASCADE)
    likes = models.ManyToManyField(User,blank=True,related_name='likes')
    dislikes = models.ManyToManyField(User,blank=True,related_name='dislikes')
```

Figure 1.2.9 Image model and Post model

Next, we need to update our post form and we will update the image widget. We can add multiple properties to allow the user to select multiple images. We also want to remove the image field from the Meta subclass. We do this to prevent the form from automatically saving the image field. We want to make sure this field is also not required so users are not forced to upload images as shown in figure 1.3.0.

```
class PostForm(forms.ModelForm):
    body = forms.CharField(
        label='',
        widget=forms.Textarea(attrs={
            'rows':'3',
            'placeholder': 'Say something...'
        }))

    image = forms.ImageField(
        required=False,
        widget=forms.ClearableFileInput(attrs={
            'multiple': True
        }))

    class Meta:
        model = Post
        fields = ['body']
```

Figure 1.3.0 PostForm(forms.py)

To save the images and add them to the Post object, we will need to get the images, then loop through them and save them as Image objects respectively. Once we have saved an image object, we can add them to the ManyToMany field of the Post object. At last, we need to make sure to call the save function to save everything to the database as shown in figure 1.3.1.

```
def post(self,request,*args,**kwargs):
    posts = Post.objects.all().order_by('-created_on')
    form = PostForm(request.POST,request.FILES)
    files=request.FILES.getlist('image')

    if form.is_valid():
        new_post = form.save(commit=False)
        new_post.author = request.user
        new_post.save()

        for f in files:
            img = Image(image=f)
            img.save()
            new_post.image.add(img)

        new_post.save()

    context = {
        'post_list':posts,
        'form':form,
    }
    return render(request,'social/post_list.html',context)
```


Figure 1.3.1 PostList view(views.py)

Now we need to update our HTML templates to show multiple images. We can check if there is a post by checking if the length is greater than 0, then we can use `post.image.all` to get an iterable list. Then we can loop through each image and display it on the page. Finally, users can add media such as images to their account as shown in figure 1.3.2.


Add a post!

Image

No file chosen


@admin March 11, 2022, 6:35 a.m.

Test user can add media



0 0

Figure 1.3.2 Output of add media

(h) correct use of models and migrations

Models are the single, unambiguous source of information about your data. It contains the basic fields and behavior of the data you store. Typically, each model maps to a database table. So that, the models.py is to let us build a database structure for the application.

First and foremost, I had created five models in my models.py. For the Post model, it can store all the post details such as body, image, created_on, author, likes and dislikes. For the comment model, it will store all the comment details such as comment, created_on, author, and the post. For the UserProfile model, it will store all the user information details such as user, name, bio, birth_date, location, picture and friends. For the Image model, it will store all the image and save it in the folder. For the ThreadModel and MessageModel, it will store all the chat details as shown in figure 1.3.3.

```
class Post(models.Model):
    body = models.TextField()
    image = models.ManyToManyField('Image', blank=True)
    created_on = models.DateTimeField(default=timezone.now)
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    likes = models.ManyToManyField(User, blank=True, related_name='likes')
    dislikes = models.ManyToManyField(User, blank=True, related_name='dislikes')

class Comment(models.Model):
    comment = models.TextField()
    created_on = models.DateTimeField(default=timezone.now)
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    post = models.ForeignKey('Post', on_delete=models.CASCADE)

class UserProfile(models.Model):
    user = models.OneToOneField(User, primary_key=True, verbose_name='user', related_name='profile', on_delete=models.CASCADE)
    name = models.CharField(max_length=50, blank=True, null=True)
    bio = models.TextField(max_length=256, blank=True, null=True)
    birth_date = models.DateField(null=True, blank=True)
    location = models.CharField(max_length=256, blank=True, null=True)
    picture = models.ImageField(upload_to='uploads/profile_pictures', default='uploads/profile_pictures/default.png', blank=True)
    friends = models.ManyToManyField(User, blank=True, related_name='friends')

@receiver(post_save, sender=User)
def create_user_profile(sender, instance, created, **kwargs):
    if created:
        UserProfile.objects.create(user=instance)

@receiver(post_save, sender=User)
def save_user_profile(sender, instance, **kwargs):
    instance.profile.save()

class Image(models.Model):
    image = models.ImageField(upload_to='uploads/post_photos', blank=True, null=True)

class ThreadModel(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE, related_name='+')
    receiver = models.ForeignKey(User, on_delete=models.CASCADE, related_name='+')

class MessageModel(models.Model):
    thread = models.ForeignKey('ThreadModel', related_name='+', on_delete=models.CASCADE, blank=True, null=True)
    sender_user = models.ForeignKey(User, on_delete=models.CASCADE, related_name='+')
    receiver_user = models.ForeignKey(User, on_delete=models.CASCADE, related_name='+')
    body = models.CharField(max_length=1000)
    image = models.ImageField(upload_to='uploads/message_photos', blank=True, null=True)
    date = models.DateTimeField(default=timezone.now)
    is_read = models.BooleanField(default=False)
```

Figure 1.3.3 models.py

After I had done the model creation, I need to run the migrations command to create the table in the database. First and foremost, I use the 'python manage.py makemigrations' command to create new migrations based on what I had made for my models. After that, I use 'python manage.py migrate' to apply the migrations as shown in figure 1.3.4.

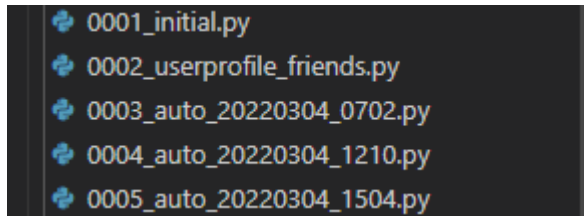


Figure 1.3.4 migrations

i) correct use of form, validators and serialisation

Django provides a Form class for creating HTML forms. It describes a form and how it works and appears. It is similar to the ModelForm class and uses a Model to create a form, but it does not require a Model.

In our socialnetwork application, we have created 4 forms which are PostForm, CommentForm, ThreadForm and MessageForm. For the PostForm, it contains a field of CharField type and a field of ImageField type. For the CommentForm and ThreadForm, it contains a field of CharField type. For the MessageForm, it contains a field of CharField type and a field of ImageField type as shown in figure 1.3.5.

```
class PostForm(forms.ModelForm):
    body = forms.CharField(
        label='',
        widget=forms.Textarea(attrs={
            'rows': '3',
            'placeholder': 'Say something...'
        }))

    image = forms.ImageField(
        required=False,
        widget=forms.ClearableFileInput(attrs={
            'multiple': True
        }))

    class Meta:
        model = Post
        fields = ['body']
```

```

class CommentForm(forms.ModelForm):
    comment = forms.CharField(
        label='',
        widget=forms.Textarea(attrs={
            'rows': '3',
            'placeholder': 'Say something...'
        }))

    class Meta:
        model = Comment
        fields = ['comment']

class ThreadForm(forms.Form):
    username = forms.CharField(label='', max_length=100)

class MessageForm(forms.ModelForm):
    body = forms.CharField(label='', max_length=1000)
    image = forms.ImageField(required=False)

    class Meta:
        model = MessageModel
        fields = ['body', 'image']

```

Figure 1.3.5 forms.py

j) correct use of django-rest-framework

Views are callables that accept requests and return responses. It's not just a function, Django provides some examples of classes that can be used as views. These allow you to build views and reuse code by leveraging inheritance and mixins. In this case, we total created 17 views in our social network application. For example, I created a view called 'PostListView' and pass in LoginRequiredMixin and View. Inside the PostListView, we created a get method which will get all the post information from oldest to newest and a post method which will add a post in their home page as shown in figure 1.3.6.

```
class PostListView(LoginRequiredMixin,View):
    def get(self,request,*args,**kwargs):
        # logged_in_user = request.user
        # posts = Post.objects.filter(
        #     author__profile_followers__in=[logged_in_user]
        # ).order_by('-created_on')
        posts = Post.objects.all().order_by('-created_on')
        form = PostForm()

        context = {
            'post_list':posts,
            'form':form,
        }
        return render(request,'social/post_list.html',context)

    def post(self,request,*args,**kwargs):
        posts = Post.objects.all().order_by('-created_on')
        form = PostForm(request.POST,request.FILES)
        files=request.FILES.getlist('image')

        if form.is_valid():
            new_post = form.save(commit=False)
            new_post.author = request.user
            new_post.save()

            for f in files:
                img = Image(image=f)
                img.save()
                new_post.image.add(img)

            new_post.save()

        context = {
            'post_list':posts,
            'form':form,
        }
        return render(request,'social/post_list.html',context)
```

Figure 1.3.6 PostList view(views.py)

k) correct use of URL routing

When a user requests a page on your web application, the Django controller takes over and looks through the url.py file for the appropriate view, returning either an HTML response or a 404 not found error if not found. In url.py, the most important is the "urlpatterns" tuple. where you define the mapping between URLs and views. For example, the landing/urls.py has one path which will return to the index view as shown in figure 1.3.7.

```
urlpatterns = [  
    path('', Index.as_view(), name='index'),  
]
```

Figure 1.3.7 urls.py(landing app)

```
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('', include('landing.urls')),  
    path('accounts/', include('allauth.urls')),  
    path('social/', include('social.urls')),  
]
```

Figure 1.3.8 urls.py(socialnetwork project)

```
urlpatterns = [  
    path('', PostListView.as_view(), name='post-list'),  
    path('post/<int:pk>', PostDetailView.as_view(), name='post-detail'),  
    path('post/edit/<int:pk>', PostEditView.as_view(), name='post-edit'),  
    path('post/delete/<int:pk>', PostDeleteView.as_view(), name='post-delete'),  
    path('post/<int:post_pk>/comment/delete/<int:pk>', CommentDeleteView.as_view(), name='comment-delete'),  
    path('post/<int:pk>/like', AddLike.as_view(), name='like'),  
    path('post/<int:pk>/dislike', Dislike.as_view(), name='dislike'),  
    path('profile/<int:pk>', ProfileView.as_view(), name='profile'),  
    path('profile/edit/<int:pk>', ProfileEditView.as_view(), name='profile-edit'),  
    path('profile/<int:pk>/friends/add', AddFriend.as_view(), name='add-friends'),  
    path('profile/<int:pk>/friends/remove', RemoveFriend.as_view(), name='remove-friends'),  
    path('profile/<int:pk>/friends/', ListFriends.as_view(), name='list-friends'),  
    path('search/', UserSearch.as_view(), name="profile-search"),  
    path('inbox/', ListThreads.as_view(), name='inbox'),  
    path('inbox/create-thread/', CreateThread.as_view(), name='create-thread'),  
    path('inbox/<int:pk>', ThreadView.as_view(), name='thread'),  
    path('inbox/<int:pk>/create-message/', CreateMessage.as_view(), name='create-message')  
]
```

Figure 1.3.9 urls.py(social app)

l) appropriate use of unit testing

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

First and foremost, we set up the media in order to store our profile pictures, post photos, and message photos. Now we need to add some changes to our settings.py to set up our media paths as shown in figure 1.4.0.

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'
```

Figure 1.4.0 settings.py

Besides, we need make a directory that matches our path set in the UserProfile model, Image model, and Message model with media added on the front of it. For example, it will look like this: media/uploads/profile_pictures/ as shown in figure 1.4.1.

```
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root = settings.MEDIA_ROOT)
```

Figure 1.4.1 settings.py(store media)

```
class Image(models.Model):
    image = models.ImageField(upload_to='uploads/post_photos', blank=True, null=True)
```

Figure 1.4.2 Example of store photos of post

Finally, we can properly store our media files as shown in figure 1.4.3.

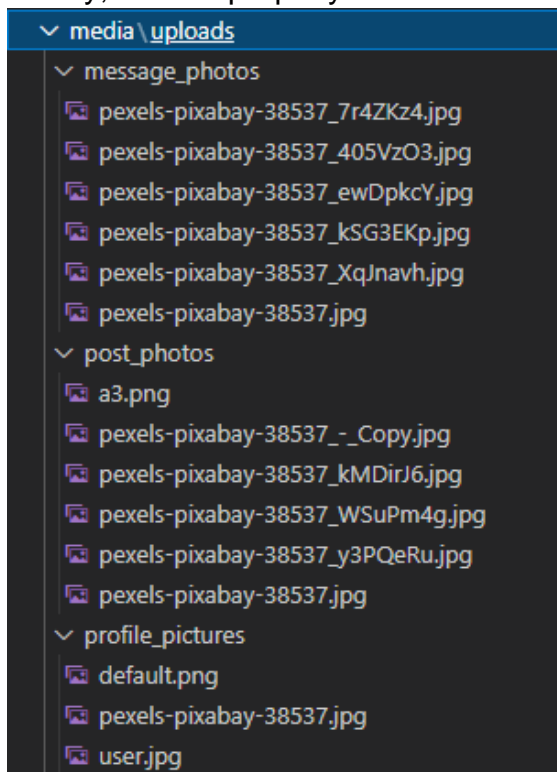


Figure 1.4.3 Store media files in media folder

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

For the Post model, we created 6 fields which are:

- body = save the content of the post.
- image = it is a many to many field that allow use save multiple images and related to Image model.
- created_on = store the created date of the post.
- author = it is a foreign key that related to the User model.
- likes = it is a many to many field that allow user to like the post.
- dislikes = it is a many to many field that allow user to dislike the post.

```
class Post(models.Model):
    body = models.TextField()
    image = models.ManyToManyField('Image', blank=True)
    created_on = models.DateTimeField(default=timezone.now)
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    likes = models.ManyToManyField(User, blank=True, related_name='likes')
    dislikes = models.ManyToManyField(User, blank=True, related_name='dislikes')
```

Figure 1.4.4 Post model

For the Comment model, we created 4 fields which are:

- comment = save the comment of the specific post.
- created_on = store the created date of the comment.
- author = it is a foreign key that related to the User model.
- post = it is a foreign key that related to the Post model.

```
class Comment(models.Model):
    comment = models.TextField()
    created_on = models.DateTimeField(default=timezone.now)
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    post = models.ForeignKey('Post', on_delete=models.CASCADE)
```

Figure 1.4.5 Comment model

For the Image model, we created one fields which is:

```
class Image(models.Model):
    image = models.ImageField(upload_to='uploads/post_photos', blank=True, null=True)
```

Figure 1.4.6 Image model

For the UserProfile model, we created 7 fields which are:

- user = It is a primary for the model and related to User model.
- name = It is a char field to store username.
- bio = It is a text field to store user's bio.
- birth_date = It store user's birth date.
- location = It store user's location.
- picture = It store user's profile picture and save each user's profile picture in the media folder.
- friends = It is a many to many field that related to User model.

```
class UserProfile(models.Model):
    user = models.OneToOneField(User, primary_key=True, verbose_name='user', related_name='profile', on_delete = models.CASCADE)
    name = models.CharField(max_length=50, blank=True, null=True)
    bio = models.TextField(max_length=256, blank=True, null=True)
    birth_date = models.DateField(null=True, blank=True)
    location = models.CharField(max_length=256, blank=True, null=True)
    picture = models.ImageField(upload_to = 'uploads/profile_pictures', default='uploads/profile_pictures/default.png', blank =True)
    friends = models.ManyToManyField(User, blank=True, related_name='friends')

    @receiver(post_save, sender=User)
    def create_user_profile(sender, instance, created, **kwargs):
        if created:
            UserProfile.objects.create(user=instance)

    @receiver(post_save, sender=User)
    def save_user_profile(sender, instance, **kwargs):
        instance.profile.save()
```

Figure 1.4.7 UserProfile model

For the ThreadModel, we created 2 fields which are:

- user=It is a foreign key that related to User model.
- receiver = It is a foreign key that related to User model.

```
class ThreadModel(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE, related_name='+')
    receiver = models.ForeignKey(User, on_delete=models.CASCADE, related_name='+')
```

Figure 1.4.8 ThreadModel model

For the MessageModel, we created 7 fields which are:

- thread = It is a foreign key that related to ThreadModel.
- sender_user = It is a foreign key that related to User model.
- receiver_user = It is a foreign key that related to User model.
- body = It store the content of each message.
- image = it is a Image field that allow user upload the image and store them in the media folder.
- date = it will store the date for each message.
- is_read = It is a boolean field.

```
class MessageModel(models.Model):
    thread = models.ForeignKey('ThreadModel', related_name='+', on_delete=models.CASCADE, blank=True, null=True)
    sender_user = models.ForeignKey(User, on_delete=models.CASCADE, related_name='+')
    receiver_user = models.ForeignKey(User, on_delete=models.CASCADE, related_name='+')
    body = models.CharField(max_length=1000)
    image = models.ImageField(upload_to='uploads/message_photos', blank=True, null=True)
    date = models.DateTimeField(default=timezone.now)
    is_read = models.BooleanField(default=False)
```

Figure 1.4.9 Message Model

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

For the REST interface of my social network web application, I had created a folder called templates in landing app and created a sub-folder called 'landing' inside the templates folder. In the landing folder, I had created index.html to inherit from the base template; base.html to hold all of our base style; and a navbar.html to hold all of our navbar style. Furthermore, I grab the bootstrap starter template from bootstrap5 website and paste in my base.html. I also create a kit from fontawesome.com to use some font awesome icons and paste the link in the base.html.

Before every user login or register their account, the default page will show each user with a heading, paragraph and two button which are login button and register button as shown in figure 1.5.0

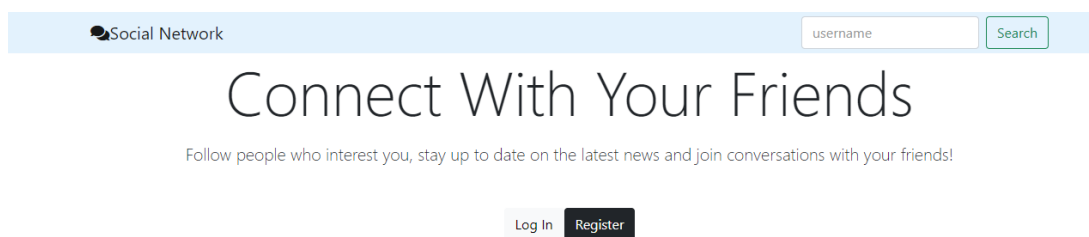


Figure 1.5.0 Index.html

Sign In

If you have not created an account yet, then please [sign up](#) first.

Username*

Password*

☐ Remember Me

[Forgot Password?](#)

Sign In

Figure 1.5.1 login.html

Sign Up

Already have an account? Then please [sign in](#).

E-mail*

Username*

Password*

Password (again)*

Sign Up »

Figure 1.5.2 register.html

After user login their account, it will redirect to their home page as shown in figure 1.5.3. In the home page, user can add the post and it will display in the below. Besides, there have a navbar include a search bar, user profile, and direct message with friends.

The screenshot shows the 'Social Network' home page. At the top is a light blue navigation bar with the site name, a search bar with the placeholder 'username', and icons for user profile and direct messages. Below the navbar is a section titled 'Add a post!'. It contains a text input field with the placeholder 'Say something...', an 'Image' section with a 'Choose Files' button and the text 'No file chosen', and a green 'POST' button. Below the form is a user post from '@admin' dated 'March 4, 2022, 6:58 a.m.' with the text 'Hi, Nice to meet you all' and a photo of a path through a forest. The post has 0 likes and 0 comments.

Figure 1.5.3 Home page

Furthermore, user can click the post to view the post detail as shown in figure 1.5.4. In the post detail page, there have a button that can back to home page, and user can edit or delete their post. User also can like or dislike the post.

The screenshot shows the post detail page. On the left is a dark button labeled 'Back To Feed'. The main content is a post from '@admin' dated 'March 4, 2022, 6:58 a.m.' with the text 'Hi, Nice to meet you all' and a photo of a path through a forest. Below the photo are 0 likes and 0 comments. At the bottom is a text input field with the placeholder 'Say something...' and a green 'Send' button.

Figure 1.5.4 Post detail page

User can view their profile page at the navbar. In the profile page, user can edit their information with the icon and all the user information will display in the center as shown in figure 1.5.5. There also have a button that can let user back to their home page.

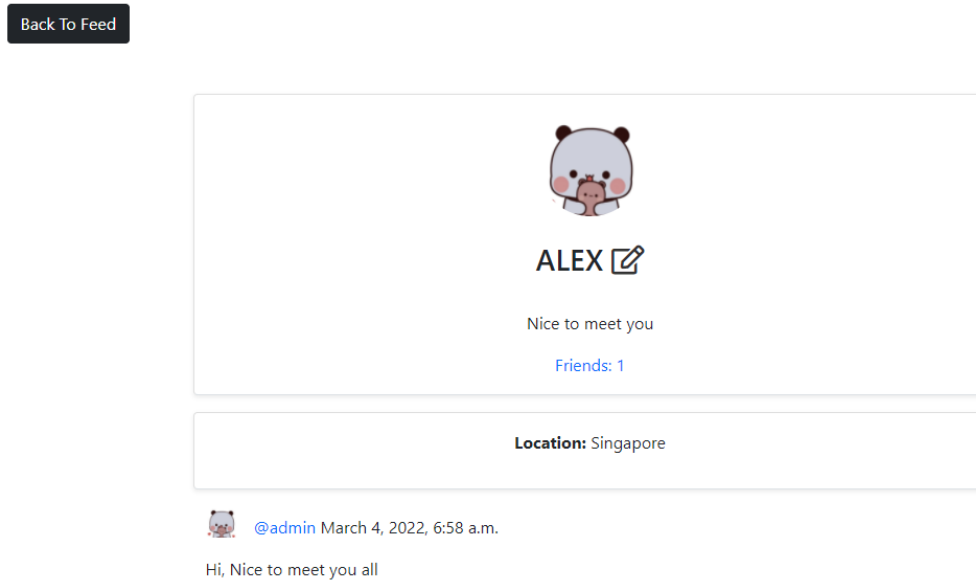


Figure 1.5.5 User profile page

To start communicate with friends, user can click the icon at the navbar. User can write message and upload some pictures to friends as shown in figure 1.5.6.

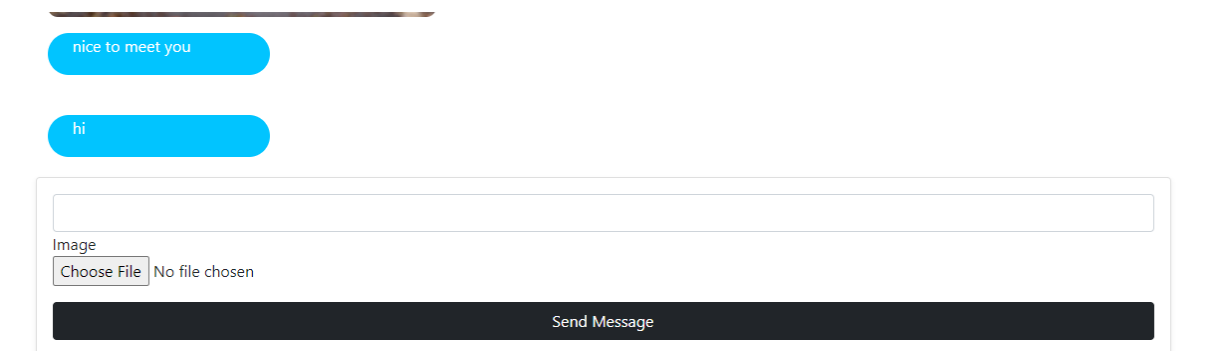


Figure 1.5.6 Inbox page

R4: Implementation appropriate tests for the server side code

In conclusion, the social network application has fulfilled all the requirements and added some extra features in order to make our application more functional. For example, we add the edit and delete post features to let users can make the change with their posts. I think the home page of the application could be better. This is because now the home page just has some basic functions. If I attempted the project again, I will change the home page and add a lot of features such as notifications, replies to comments, and so on. It is a good experience for doing this project. It let me learn a lot of knowledge about Django and Python.