1. **Created the new Django project:** I created a new Django project by running the following command:

    - **django-admin startproject Django-based backend system for a social media platform social media**

    - This command will create a new directory named ‘social media’, which will contain the basic structure of a Django project.

2. **Created new apps:** A Django project can have multiple apps, each containing a specific set of functionalities. I created a new app by running the following command:

    - **python manage.py startapp user**
    - **python manage.py startapp user_profile**
    - **python manage.py startapp posts**
    - **python manage.py startapp likes**
    - **python manage.py startapp comments**
    - **Python manage.py startapp search**
    - **python manage.py startapp api**

    - These commands will create a new directory, which will contain the basic structure of a Django app.

3. **Defined models:** In Django, models define the structure of your database tables. I define the models in the 'models.py' file of the app.

    - For this project, i needed to define two models:

        a. **Profile Model:** This model will have many fields 'dob', ‘gender’, ‘phone’, ‘works_at’, ‘studies_at’, ‘profile_image’ which will store the data of the user.

```
class Profile(models.Model):
    owner = models.OneToOneField(User, on_delete=models.CASCADE,
related_name='profile_data')
    gender = models.CharField(
        max_length=20,
        choices=options,
        default='male',
        null=False,
        blank=False
    )
    dob = models.DateField(null=True, blank=True, default=None)
    phone = models.CharField(max_length=20, null=True, blank=True)
    works_at = models.CharField(max_length=200, null=True, blank=True)
```

```
    studies_at = models.CharField(max_length=20, null=True, blank=True)
    profile_image = models.ImageField(upload_to="profile_image", null=True, blank=True)
```

b. **Post model:** This model will have four fields: 'owner' (a foreign key to the user register), 'content', 'links', 'post_image', and 'post_data' (auto_now).

```
from django.db import models

# Create your models here.
class Post(models.Model):
    owner=models.ForeignKey('auth.User',related_name='posts',on_delete=models.CASCADE)
    content=models.CharField(max_length=200)
    links=models.URLField(null=True,blank=True)
    post_image=models.ImageField(upload_to="post_image",null=True,blank=True)
    post_date= models.DateField(auto_now_add=True)
    def __str__(self):
        return self.content
```

c. **Like model:** This model will have two fields: ‘post’ (a foreign key to the Post model), ‘like’.

```
from django.db import models
from posts.models import Post

# Create your models here.

class Like_Model(models.Model):
    post=models.ForeignKey(Post,related_name='likes',on_delete=models.CASCADE)
    like=models.ForeignKey('auth.User',related_name='like_user',on_delete=models.CASCADE,
default=None, blank=True, null=True)
    def __str__(self):
        return self.post.content
```

d. **Comment model:** This model will have four fields: ‘owner’, ‘post’ (a foreign key to the Post model), ‘comment’, comment_date (auto_now-add).

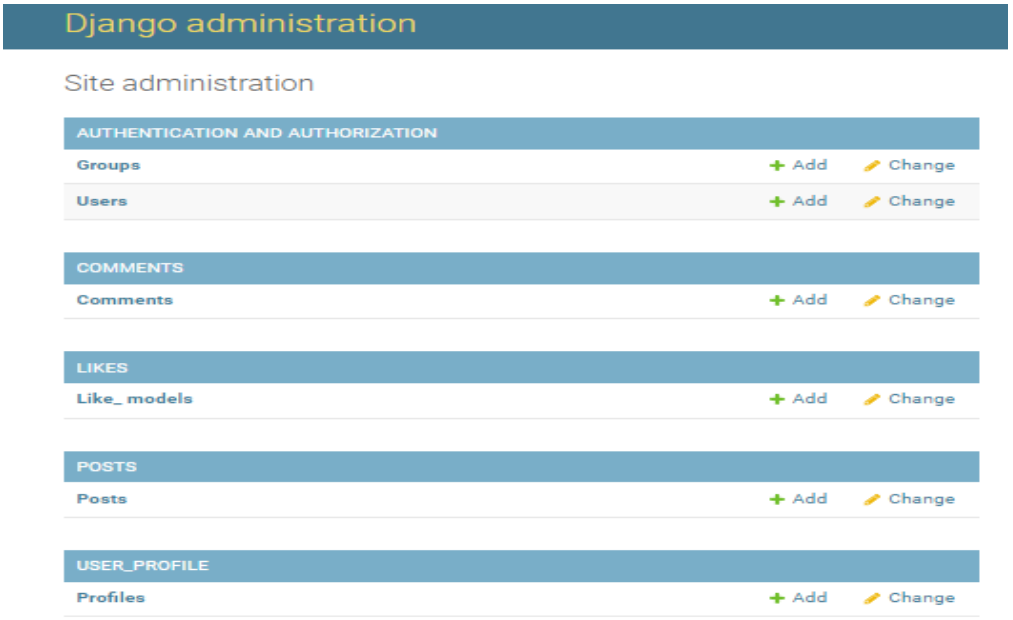
```
from django.db import models
from posts.models import Post

# Create your models here.
```

```
class Comment(models.Model):
   owner = models.ForeignKey('auth.User', on_delete=models.CASCADE)
   post=models.ForeignKey(Post,related_name='comments',on_delete=models.CASCADE)
   comment=models.CharField(max_length=2000)
   comment_date=models.DateField(auto_now_add=True)
   def __str__(self):
      return self.comment
```

4. **Created database tables:** After defining the models, I needed to create the corresponding database tables. I did this by running the following command:

    - **python manage.py migrate**

    - This command will create the necessary database tables for my apps.



5. **Serializers :** Serializers play a crucial role in the serialization and deserialization of data in DRF, which is essential for building RESTful APIs. For this project, I needed to define many serializers -

    - UserSerializers

```
from rest_framework import serializers
from django.contrib.auth.models import User
from user_profile.serializers import ProfileSerializer
```

```python
class UserSerializers(serializers.ModelSerializer):
    profile_data=ProfileSerializer(read_only=True)
    class Meta:
        model=User
        fields = ('id','username', 'email','password','is_active', 'profile_data')

extra_kwargs={'email':{'required':True,'write_only':True},'password':{'write_only':True}}

    def create(self,validate_data):
        user=User(
            email=validate_data['email'],
            username=validate_data['username']
        )

        user.set_password(validate_data['password'])
        user.save()
        return user
```

The code you provided is a Django REST Framework (DRF) serializer for the built-in Django User model. Let me explain what this code does.

First, the necessary imports are made: **serializers** from **rest_framework** and **User** model from **django.contrib.auth.models**. Additionally, the serializer for the related Profile model is imported from the **user_profile** app.

The **UserSerializers** class inherits from **serializers.ModelSerializer**, which is a shortcut for creating serializers that deal with model instances. The **profile_data** field is defined as an instance of **ProfileSerializer**, which is a nested serializer used to serialize the related profile data of the user.

The **Meta** class is used to specify the model and fields to be serialized. In this case, the model is **User** and the fields to be serialized are **id, username, email, password, is_active**, and **profile_data**. The **extra_kwargs** dictionary is used to set additional options for the **email** and **password** fields.

The **create** method is used to create a new user instance from the validated data received from a POST request. It creates a new **User** instance with the provided email and username, sets the password using the **set_password()** method, saves the user to the database, and returns the newly created user instance.

Overall, this serializer allows for the serialization and deserialization of **User** instances, including related profile data, and provides additional validation and creation functionality for creating new user instances through POST requests.

- PostSerializers: There are following fields in post serializers- 'id', 'content', 'post_image', 'links', 'post_date', 'comments', 'likes'.

```
from rest_framework import serializers

from .models import Post

from comments.serializers import CommentSerializers

from likes.serializers import LikeSerializers

class PostSerializer(serializers.ModelSerializer):

    comments=CommentSerializers(many=True, read_only=True)

    likes=LikeSerializers(many=True, read_only=True)

    class Meta:

        model = Post

        fields =
('id','content','post_image','links','post_date','comments','likes')
```

- LikeSerializers: There are following fields in like serializers- 'id', 'post', 'like_by'.

```
from rest_framework import serializers

from .models import Like_Model

class LikeSerializers(serializers.ModelSerializer):

    like_by=serializers.ReadOnlyField(source='like_by.username')

    class Meta:

        model=Like_Model

        fields=('id','post','like_by')
```

- CommentSerializers: There are following fields in comment serializers- 'id','post','like_by'

```
from rest_framework import serializers

from .models import Like_Model

class LikeSerializers(serializers.ModelSerializer):

    like_by=serializers.ReadOnlyField(source='like_by.username')

    class Meta:

        model=Like_Model

        fields=('id','post','like_by')
```

6. **Created views:** Views are the Python functions that handle requests and generate responses in Django. I defined the views in the 'views.py' file of the apps. For this project, I needed to define many views in many apps:

    - **user_profile view:** This view will check that some permissions like IsAuthenticatedOrReadOnly, IsOwnerOrReadOnly.

```
from user_profile.serializers import ProfileSerializer
from user_profile.models import Profile
from rest_framework import viewsets, permissions
from .permissions import IsOwnerOrReadOnly
# Create your views here.

class ProfileViewSet(viewsets.ModelViewSet):
    queryset=Profile.objects.all()
    serializer_class=ProfileSerializer

permission_classes=[permissions.IsAuthenticatedOrReadOnly,Is
OwnerOrReadOnly]

    def perform_create(self, seralizer):
        seralizer.save(owner=self.request.user)
```

- **posts view:** This view will check some permissions like IsAuthenticatedOrReadOnly, IsOwnerOrReadOnly and also User can search the post by content.

```
from django.shortcuts import render
from rest_framework import viewsets, status, permissions
from .models import Post
from user_profile.permissions import IsOwnerOrReadOnly
from .serializers import PostSerializer
from django_filters.rest_framework import
DjangoFilterBackend

# Create your views here.

class PostViewSet(viewsets.ModelViewSet):
    queryset=Post.objects.all()
    serializer_class=PostSerializer

permission_classes=[permissions.IsAuthenticatedOrReadOnly,
IsOwnerOrReadOnly]
    filter_backends = [DjangoFilterBackend]
    filterset_fields = ['content']
    def perform_create(self, serializer):
        serializer.save(owner=self.request.user)
```

- **Like Views:** In views.py file, This is the logical code for the user to like the post.

```
from django.shortcuts import render, get_object_or_404
from posts.models import Post
from likes.permissions import hasSelfLikeOrReadOnly
from rest_framework import viewsets, status, permissions, serializers
from .models import Like_Model
from .serializers import LikeSerializers

# Create your views here.
class LikeViewSet(viewsets.ModelViewSet):
    queryset=Like_Model.objects.all()
    serializer_class=LikeSerializers

permission_classes=[permissions.IsAuthenticatedOrReadOnly,hasSelfLikeOr
ReadOnly]
```

```
    def perform_create(self, serializer):
        post_instance = get_object_or_404(Post,
pk=self.request.data['post'])
        like = self.request.data.get('like')  # use get method to avoid
MultiValueDictKeyError
        if like:
            already_like =
Like_Model.objects.filter(post=post_instance,
like_by=self.request.user).exists()
            if already_like:
                raise serializers.ValidationError({"message":"you have
already liked this post"})
            else:
                serializer.save(like_by=self.request.user,
post=post_instance)
```

- **Comment Views:** In views.py file, This is the logical code for the user to make a comment on the post.

```
from django.shortcuts import render
from rest_framework import viewsets, status, permissions
from .models import Comment
from .serializers import CommentSerializers
from user_profile.permissions import IsOwnerOrReadOnly

# Create your views here.

class CommentViewSet(viewsets.ModelViewSet):
    queryset=Comment.objects.all()
    serializer_class=CommentSerializers

permission_classes=[permissions.IsAuthenticatedOrReadOnly,IsOwnerOrRead
Only]
    def perform_create(self, serializer):
        serializer.save(owner=self.request.user)
```

- **Search Views:** In views.py file, This is the code for the user to search the other user.

```
from rest_framework import viewsets, status, permissions
from posts.models import Post
from user_profile.permissions import IsOwnerOrReadOnly
```

```
# from posts.serializers import PostSerializer
from django_filters.rest_framework import DjangoFilterBackend

from user_profile.serializers import ProfileSerializer
from user_profile.models import Profile



class SearchViewSet(viewsets.ModelViewSet):
    queryset=Profile.objects.all()
    serializer_class=ProfileSerializer

permission_classes=[permissions.IsAuthenticatedOrReadOnly,IsOwnerOrRead
Only]
    filter_backends = [DjangoFilterBackend]
    filterset_fields = ['owner']
```

8. Admin.py : The admin.py file is used to register models and customize the admin interface for those models.

```
from django.contrib import admin
from posts.models import Post
# Register your models here.
admin.site.register(Post)
```

9. Permissions

- **User_profile.permissions.py :** In this file we create a permissions that the login user edit only his/her profile and only view the other profile.

```
from rest_framework import permissions

class IsOwnerOrReadOnly(permissions.BasePermission):
    def has_object_permission(self, request, view, obj):
        if request.method in permissions.SAFE_METHODS:
            return True
        return obj.owner==request.user
```

- **Likes.permissions.py :** In this file I define a permissions that the owner of the post is also like their post.

```
from rest_framework import permissions
class hasSelfLikeOrReadOnly(permissions.BasePermission):
```

```
    def has_object_permission(self, request, view, obj):
        if request.method in permissions.SAFE_METHODS:
            return True
        return obj.like_by==request.user
```

10. Social media file : This is the inner project folder.

Settings.py file : This is the main file of the whole project. All the register apps are present in settings.py file and database connectivity, permissions, rest_framework is also defined in this file.
Database connectivity

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'social',
        'USER': 'root',
        'PASSWORD': 'Admin@123',
        'HOST': '127.0.0.1',
        'PORT': '3306',
        'OPTIONS': {
            'init_command': "SET sql_mode='STRICT_TRANS_TABLES'"
        }
    }
}
```

7. **Defined URLs:** URLs map requests to views in Django. You can define your URLs in the 'urls.py' file of your app. For this project, you need to define the following URLs:

```
from django.contrib import admin
from django.urls import path
from django.urls.conf import include
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/',include('api.urls')),
    path('api-auth/',include('rest_framework.urls')),
]
```

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

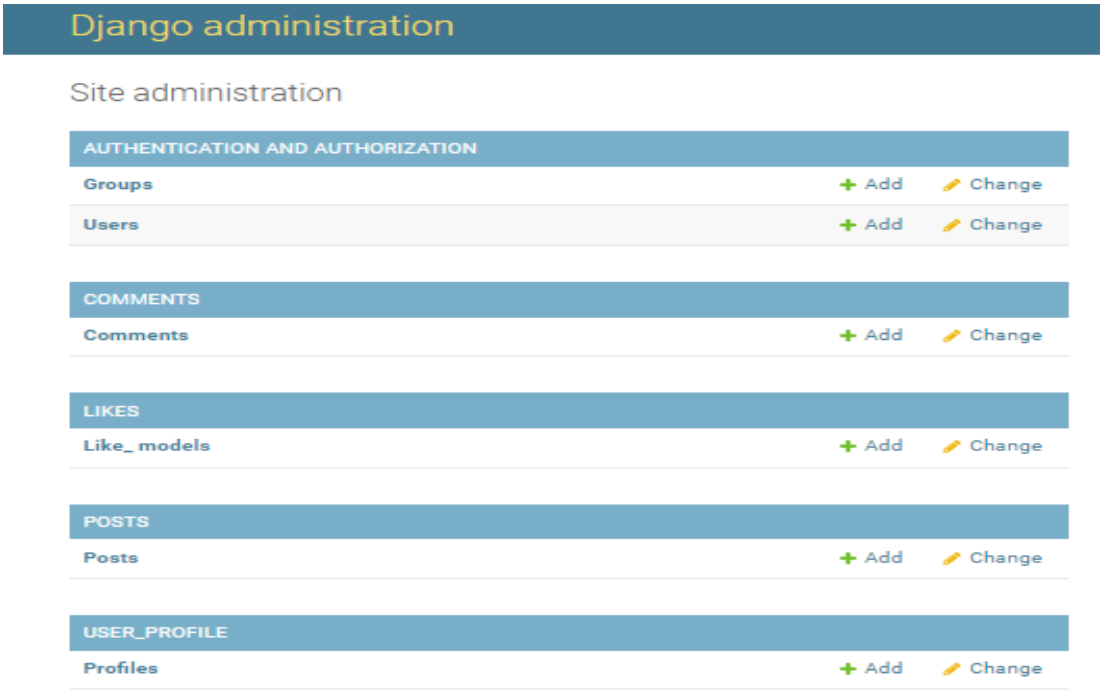
- **‘admin/’:** This URL will map requests to view the Admin panel.
- **api/** : This includes the api.urls file.

Api.urls file : Overall, this code sets up the URL routing for a DRF project using a DefaultRouter, which simplifies the process of defining URL patterns for view sets.

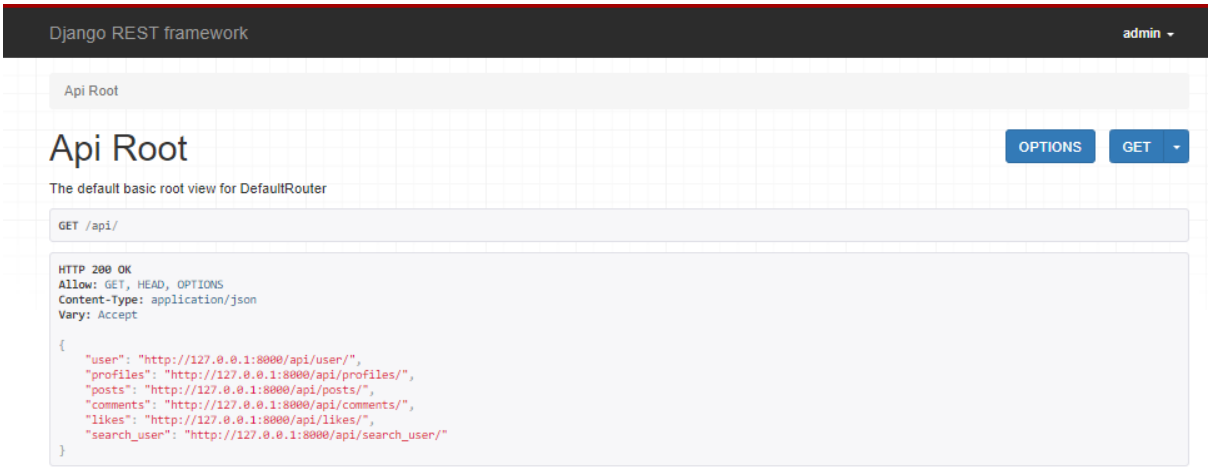
```
from rest_framework.routers import DefaultRouter
from users.views import UserViewSet
from user_profile.views import ProfileViewSet
from posts.views import PostViewSet
from comments.views import CommentViewSet
from likes.views import LikeViewSet
from searchs.views import SearchViewSet

router=DefaultRouter()

router.register(r'user',UserViewSet,basename='user')
router.register(r'profiles',ProfileViewSet,basename='profiles')
router.register(r'posts',PostViewSet, basename='posts')
router.register(r'comments',CommentViewSet, basename='comments')
router.register(r'likes',LikeViewSet,basename='likes')
router.register(r'search_user',SearchViewSet,basename='searchs')

urlpatterns=router.urls
```

Outcome:

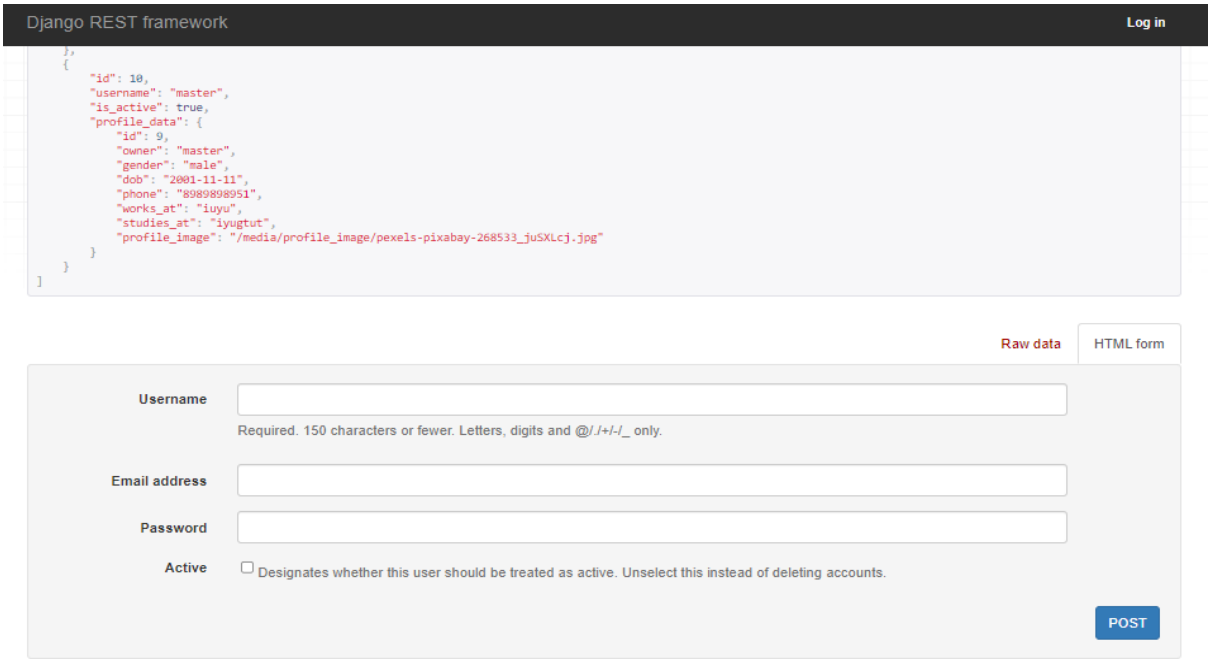
For admin urls is - http://127.0.0.1:8000/admin/



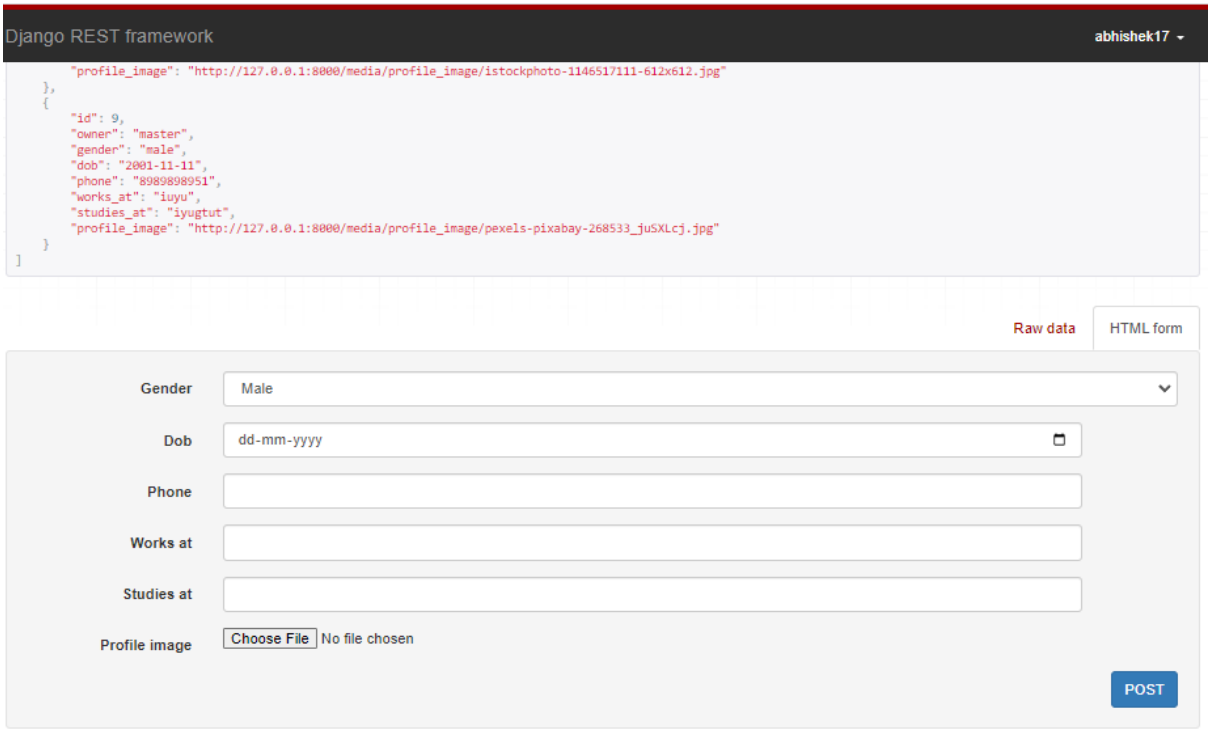
My api roots is http://127.0.0.1:8000/api/



For user register/login : http://127.0.0.1:8000/api/user/

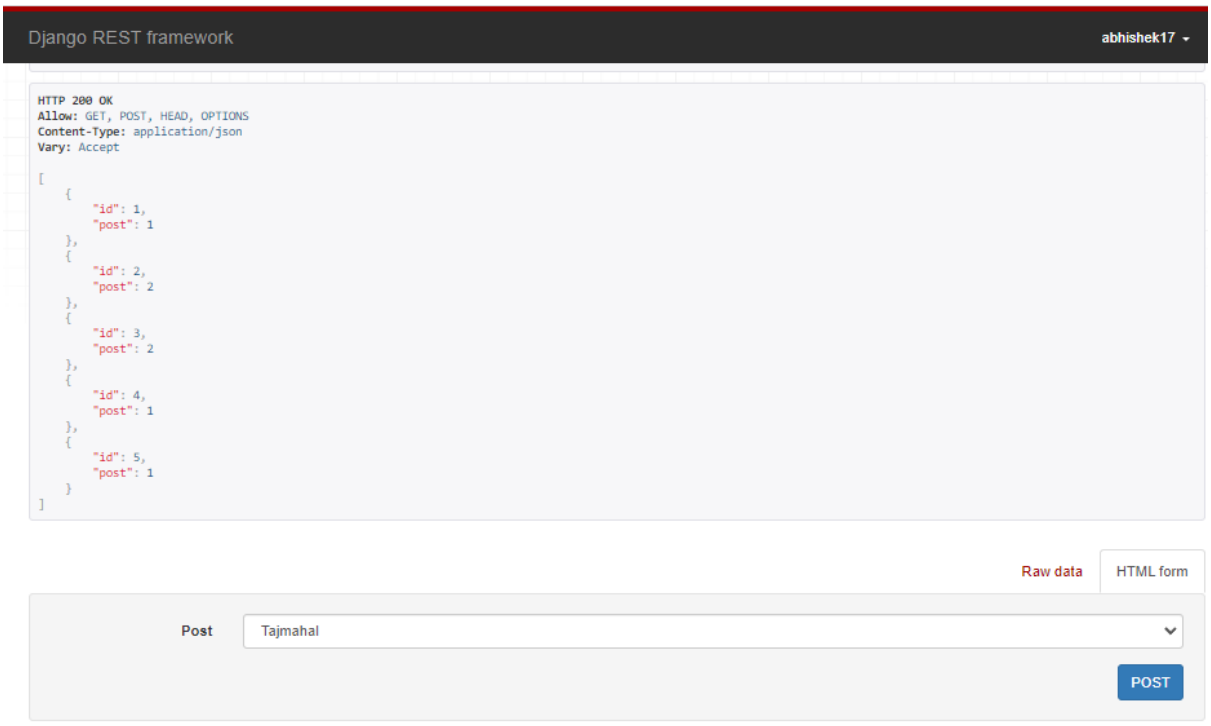


For user profile setting :http://127.0.0.1:8000/api/profiles/

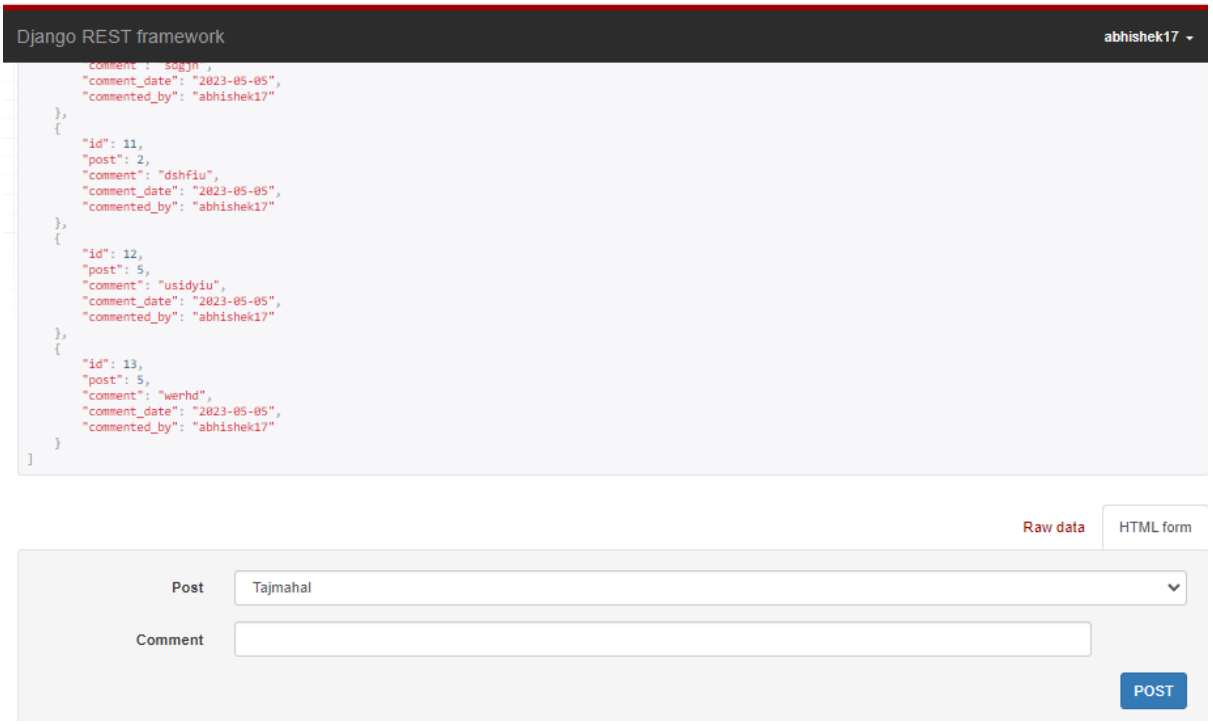


For post : http://127.0.0.1:8000/api/posts/

For like : http://127.0.0.1:8000/api/likes/



For comment :http://127.0.0.1:8000/api/comments/



For user search :