FULL STACK TWEET APP COPY:

Installed a virtual environment using:

python -m venv .venv

This creates a new file named .venv which will help us activate the virtual environment using the command:

.venv\Scripts\activate

This activates the virtual environment and we will install django in that virtual environment using the command:

pip install django

After that to check we can run the command:

django-admin –version

After that we start the project using the command:

django-admin startproject projecthq

Here the ‘projecthq’ becomes the name of our project and we can start working on it.
After that we can run the server using:

python manage.py runserver

Here the manage.py file is present in our project and it helps us run servers and also make a lot of changes, it gets created when we start our project.
A little trick to find out if we are in the right directory is that i can write the starting two letters of a certain file name and then press **TAB** if the file name completes itself with the correct file name that means we have entered the correct directory.

Now when we run the server it runs peacefully but it shows errors of migrations in the termina, they appear because we have some user models which need to go to admin and they havent been migrated yet so we can correct those error using two commands:

python manage.py makemigrations
python manage.py migrate

When we clear these migrations we can easily access admin and we should also create a superuser for that admin panel. We do that with the help of the command:

python manage.py createsuperuser

After that we give a username and a password and confirm it, giving email is also an option but it is optional so we can skip it.
In here the info i gave was
Username: vansh
Email: vansh@gmail.com
Password: Chai@123

After this we can go to the admin url and login using the credentials we gave and created.
Now most of out work will be done in projecthq file where we create all the apps and urls and models etc
Then we go to settings.py in projecthq to import os using:

import os

The os module can help you when working with images in your project by allowing you to manage file paths and directories in a system-independent way.
We installed os because we will be importing images in our project and since we will be playing with images we also need to install pillow. But before that we go to out settings.py and add configuration of images by adding:

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

We add these two lines at the end of the settings.py so that the images can get easily configured.basically it helps in the configuration of how media files (like images, videos, documents, etc.) are handled in your Django project.

1. `MEDIA_URL`

- This defines the **URL endpoint** where your media files can be accessed in the browser. For example, if you upload an image, the `MEDIA_URL` specifies the base URL for accessing that image.

  Example:

  `MEDIA_URL = '/media/'`

- Here, `/media/` is the base URL where the media files will be accessible. If you upload a file like `my_image.jpg`, it would be accessible at `http://yourdomain.com/media/my_image.jpg`.

2. `MEDIA_ROOT`

- This defines the **absolute path** on your server or local system where the uploaded media files are stored. It tells Django where to save files when they are uploaded.

  Example:

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

- This means that all uploaded media files will be stored in the `media/` folder located in the project's root directory (`BASE_DIR`).

3. How Does This Help with Media Configuration?

"Configuration" in this context refers to **setting up how Django handles media files**. By adding `MEDIA_URL` and `MEDIA_ROOT`, you are telling Django:

- Where to **store** media files on the server or local machine (`MEDIA_ROOT`).
- How these media files can be **accessed** via a URL in the browser (`MEDIA_URL`).

This is crucial when you're dealing with **user-uploaded content** like profile pictures, documents, etc. Django needs to know where to place these files and how to serve them when users or the site need to access them.

And as for static files:

```
STATIC_URL = '/static/'

STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static'),]
```

Static file configuration refers to how Django manages, locates, and serves static assets (CSS, JavaScript, images) in your project. This configuration tells Django **where** to find static files and **how** to make them available to users, ensuring that the front-end loads correctly.

1. `STATIC_URL = '/static/'`

- **Purpose**: This defines the **URL path** through which static files are accessed. It tells Django and the browser where static files (CSS, JS, images) can be found.
- **Example**: If you have a CSS file called `style.css`, this setting makes it accessible at `http://yourdomain.com/static/style.css`.
- **Importance**: This ensures that static files can be **served correctly to the browser**. When you load a web page, it will look for CSS, JS, or images at the `/static/` URL path.

2. `STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static')]`

- **Purpose**: This tells Django where to **look for static files** within your project. It specifies the directories that contain your CSS, JS, images, and other static assets.
- **`os.path.join(BASE_DIR, 'static')`**: This creates a path to the `static/` folder located in the root directory of your project (next to `settings.py`). This is where Django will search for static files during development.
- **Importance**: It helps Django **find static assets** within your project, ensuring that the necessary styles, scripts, and images are available to your web pages.

How This Configuration Works:

- When you run your project and load a web page, Django serves static files like CSS or JavaScript from the `static/` directory. These files are accessible through the `/static/` URL, as defined by `STATIC_URL`.

  Example:

  ```
  <link rel="stylesheet" href="{% static 'css/style.css' %}">
  ```

- This template tag `{% static 'css/style.css' %}` generates the correct URL based on `STATIC_URL` and `STATICFILES_DIRS`. Django will serve the file `style.css` from the `static/css/` directory at the URL `/static/css/style.css`.

We basically create a folder named static and keep css and js files in it to serve on the front end.

But we cant just add static media like this we have to add it to urls.py as well because that is where we will access urls and we will get to see which url gives which static file and we do that by going to urls.py and:

```
urlpatterns = [

    path('admin/', admin.site.urls),

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

Defining URL Patterns

```
urlpatterns = [

    path('admin/', admin.site.urls),

]
```

- **`urlpatterns`**: This is a list that defines the URL patterns for your Django application. Each pattern corresponds to a specific view or functionality within your app.
- **`path('admin/', admin.site.urls)`**: This line adds a URL pattern for the Django admin site. When you visit `http://yourdomain.com/admin/`, the Django admin interface will be displayed.

Serving Media Files

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

- **`+ static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)`**: This line appends the media URL pattern to your `urlpatterns`.
    - **`settings.MEDIA_URL`**: This is the URL prefix used to access media files (e.g., `/media/`). It defines how the media files will be accessed via the web browser.
    - **`document_root=settings.MEDIA_ROOT`**: This specifies the filesystem path where uploaded media files are stored. The `document_root` is the directory from which the media files will be served.

Using this we add the media urls to the file and also we cant just use settings like that without importing it so we also do add these two commands in them:

```
from django.conf import settings

from django.conf.urls.static import static
```

1. Import Statements

```
from django.conf import settings

from django.conf.urls.static import static
```

- **`from django.conf import settings`**: This imports the `settings` module, which contains all your project's configuration settings (like `MEDIA_URL`, `MEDIA_ROOT`, `STATIC_URL`, and `STATIC_ROOT`). You can access any settings defined in your `settings.py` file using this module.
- **`from django.conf.urls.static import static`**: This imports the `static` function, which is specifically designed to serve static and media files during development. This function helps generate URL patterns that point to the locations of your static and media files based on the settings defined in `settings.py`.

With all of this done we have done most of the basic requirements for us to start an app.We do start the app using a command in the terminal and we keep the app in the project file, that command is:

python manage.py startapp tweet

This command creates and app named 'tweet' in our project files and will have a lot of functionality. We dont find any urls.py and forms.py in this start app file so we create it.

We create urls.py and since we dont know what will be put in it firstly we put all the data from our main projects urls.py into this urls,py file

We then create a small view in our urls.py file in the app and through that view we create a template called index html which helps us test our application properly. A view in Django contains the logic to process a web request and return a response. It acts as a bridge between the web user interface and the backend logic of your application. To do all that we go to the views.py file and create views.

```python
from django.shortcuts import render



# Create your views here.

def index(request):

    return render(request, 'index.html')
```

This is how i wrote a single view it may change later when we add more views

Defining the View Function

```
def index(request):
```

- **`def index(request):`**: This line defines a view function named `index`. In Django, view functions are Python functions that take an HTTP request as an argument and return an HTTP response. The `request` parameter is an instance of `HttpRequest`, which contains metadata about the request (like GET or POST data).

Rendering the Template

```
return render(request, 'index.html')
```

- **`return render(request, 'index.html')`**: This line does the following:
  - **`render(...)`**: Calls the `render` function.
  - **`request`**: Passes the `request` object so that Django knows which request is being processed.
  - **`'index.html'`**: Specifies the template to be used. In this case, it's looking for a file named `index.html` in the template directory of your Django app.

How It Works Together

- When a user navigates to the URL associated with this view (typically defined in `urls.py`), Django calls the `index` function.
- Inside the `index` function, the `render` function combines the specified `index.html` template with the request context (if any) and returns the rendered HTML as an `HttpResponse` object.
- The rendered HTML is then sent back to the user's browser, allowing them to see the content of `index.html`.

Then we go back to the urls.py file to import that view path and show it to the user.

But also we have to tell our main settings.py in our main project that we have created a new app which helps our main project to recognize the app.

So we go to out settings.py and go to the INSTALLED_APPS section and add the name of our app also we configure the templates which helps django where should we pick our html templates from. When we configure templates django ensures that we can find and render the templates when needed. We do that by going to settings.py nad searching for TEMPLATES and in there there is a DIRS and inside that we add it like

```
TEMPLATES = [

    {

        'DIRS': [os.path.join(BASE_DIR, 'templates')],            ],

        },

    },

]
```

It simply means “ os.path.join karna hai base directory and templates se”

With this now i can put a folder named templates in each app which will be hunted through this.

After all that we also tell the urls.py file that we have a new app. We do that by importing an import with the path from the django.urls and then we create a new path like this:

```
path('tweet/', include('tweet.urls')),
```

This simply means that whenever we go to the tweet include a file named urls from tweet and do the rest included in that file.we use include because we have already told our installedapps that we have created an app.

After we run the server and go to the specified tweet page it will first access the urls.py of our main project which will redirect it to the urls.py of our tweet app which will redirect it to views.py and finally that views.py will redirect to a templates folder which will contain the index.html file which will be represented

It will all work like this:

We first visit the tweet page then:

```
from django.contrib import admin

from django.urls import path, include

from django.conf import settings

from django.conf.urls.static import static



urlpatterns = [

    path('admin/', admin.site.urls)

    path('tweet/', include('tweet.urls')),

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

This code runs and redirect us to the urls.py of our tweet app, which will have the code:

```
from django.urls import path

from . import views



urlpatterns = [

    path('', views.index, name='index'),

]
```

Then this file will redirect us to the views.py which will be:

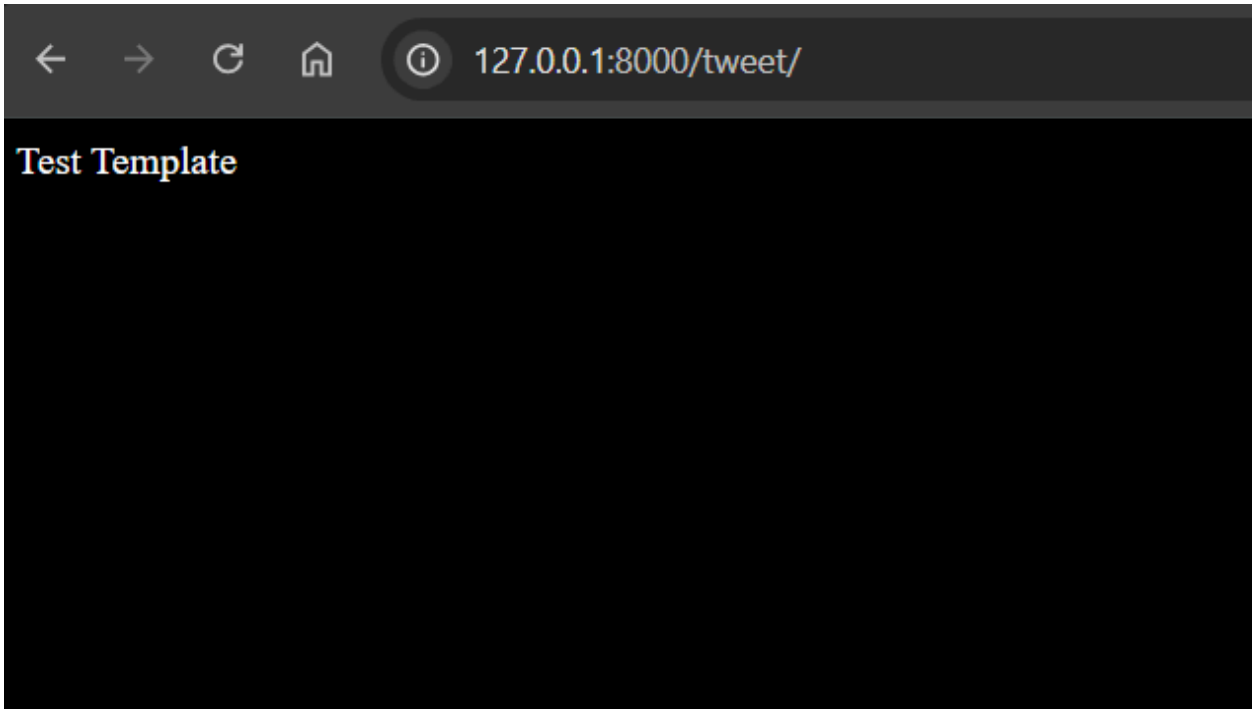
```
from django.shortcuts import render



# Create your views here.

def index(request):

    return render(request, 'index.html')
```

And finally this views will redirect us to the templates and index.html inside it:

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

    <style>
```

```
        body{
            background-color: #000;
            color: #FFF;
        }
    </style>
</head>
<body>
    Test Template
</body>
</html>
```

And this will be respresented on the tweet url:



We also create a template file in the project file because we have to structure our project in the same way and the same kind of style.so we create a folder named template in the project file and create a file named layout.html in it which will have common template which will be followed in the entire project.

When we create the file in layout.html we also apply a load static therewhich looks like this in total:

```
{% load static %}

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>

        {% block title %}

        project from django

        {% endblock %}

    </title>

</head>

<body>

    <div class="container">

        {% block content %}

        {% endblock %}

    </div>

</body>

</html>
```

What Does `{% load static %}` Do?

- **Purpose**: This tag tells Django that you want to use the static files handling functionality in your template. It allows you to access the static files (like CSS, JavaScript, images, etc.) you have in your static directories.

Static URL Generation: After loading the static tag library, you can use the `{% static 'path/to/your/static/file' %}` syntax to generate the correct URL for your static files.

For example:

```
<link rel="stylesheet" href="{% static 'css/styles.css' %}">
```

- `<script src="{% static 'js/scripts.js' %}"></script>`

1. The `{% block title %}`

html

Copy code

```
<title>
    {% block title %}
    project from django
    {% endblock %}
</title>
```

- **Purpose**: This block defines a section for the page title that can be customized in child templates. When you extend this base template in another template, you can provide a different title.
- **Default Content**: The default content within the block is "project from django." If a child template does not override this block, this title will be used.

Customization: In a child template, you could override this block like this:

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


{% block title %}

My Custom Title

{% endblock %}
```

- This would change the title of the page to "My Custom Title" instead of the default.

2. The `{% block content %}`

html

Copy code

```
<div class="container">

    {% block content %}

    {% endblock %}

</div>
```

- **Purpose**: This block is for the main content of the page. It allows you to insert different content for each child template that extends the base template.
- **Empty by Default**: The block is empty by default. In a child template, you can fill this block with any HTML content you want to display on that particular page.

Customization: In a child template, you might do something like this:

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


{% block content %}

<h1>Welcome to My Project</h1>
```

```
<p>This is the home page of my Django project.</p>

{% endblock %}
```

- This would replace the empty content block in the base template with the specified HTML.

Now we install bootstrap so that we can design the UI of the app a little bit. We simply copy the link line from the bootstrap file online and copy it into the layout.html file . Even though it becomes a little bloated it still makes our work easy by making our typing part less.
After installing bootstrap in our layout file we also install the navbar component so that we can use it in very page.

Now we have totally done the layout configuration properly now we can head to models which help us in backend and we model data and forms.

We create a file named models in our tweet app.The `models.py` file is crucial in Django or similar web frameworks, as it defines the structure and behavior of the data that your application will work with.

1. Defining the Database Schema

- **Purpose**: The `models.py` file contains Python classes that map to your database tables. Each class represents a model or database table, and the class attributes correspond to the columns of the table.

Example:

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=100)
    published_date = models.DateField()
    isbn = models.CharField(max_length=13)
```

- Here, the `Book` class defines a table with columns for the title, author, published date, and ISBN.

2. ORM (Object-Relational Mapping)

- **Purpose**: Django (and other frameworks) uses an ORM to automatically translate Python objects into database queries. By defining models in `models.py`, you avoid writing SQL manually. The ORM handles querying, updating, and deleting rows in your database.

Example: Instead of writing SQL queries, you can interact with the database using Python code:

```
# Create a new book
new_book = Book(title="1984", author="George Orwell",
published_date="1949-06-08", isbn="1234567890123")
new_book.save()

# Query books by author
```

- `books = Book.objects.filter(author="George Orwell")`

3. Validation and Business Logic

- **Purpose**: Models can include validation rules and methods to enforce data integrity and implement business logic. You can define constraints on fields (e.g., field length, required values) and even custom methods.

Example:

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=100)

    def __str__(self):
        return self.title

    def is_long_book(self):
```

- `        return len(self.title) > 50`

The `models.py` file is crucial in Django or similar web frameworks, as it defines the structure and behavior of the data that your application will work with. Here's why it's needed:

1. Defining the Database Schema

- **Purpose**: The `models.py` file contains Python classes that map to your database tables. Each class represents a model or database table, and the class attributes correspond to the columns of the table.

Example:
python
Copy code

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=100)
    published_date = models.DateField()
    isbn = models.CharField(max_length=13)
```

- Here, the `Book` class defines a table with columns for the title, author, published date, and ISBN.

2. ORM (Object-Relational Mapping)

- **Purpose**: Django (and other frameworks) uses an ORM to automatically translate Python objects into database queries. By defining models in `models.py`, you avoid writing SQL manually. The ORM handles querying, updating, and deleting rows in your database.

Example: Instead of writing SQL queries, you can interact with the database using Python code:
python
Copy code

```
# Create a new book
new_book = Book(title="1984", author="George Orwell",
published_date="1949-06-08", isbn="1234567890123")
new_book.save()

# Query books by author
books = Book.objects.filter(author="George Orwell")
```

3. Validation and Business Logic

- **Purpose**: Models can include validation rules and methods to enforce data integrity and implement business logic. You can define constraints on fields (e.g., field length, required values) and even custom methods.

Example:
python
Copy code

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=100)

    def __str__(self):
        return self.title

    def is_long_book(self):
        return len(self.title) > 50
```

4. Migration Management

- **Purpose**: Whenever you create or update models in `models.py`, Django tracks these changes and generates database migrations. These migrations are used to apply changes to your database schema without manually altering tables.

Example: After updating `models.py`, you run:

```
python manage.py makemigrations
python manage.py migrate
```

- This updates your database schema according to the changes in your models.

Then we import users into models.py using the import statement:

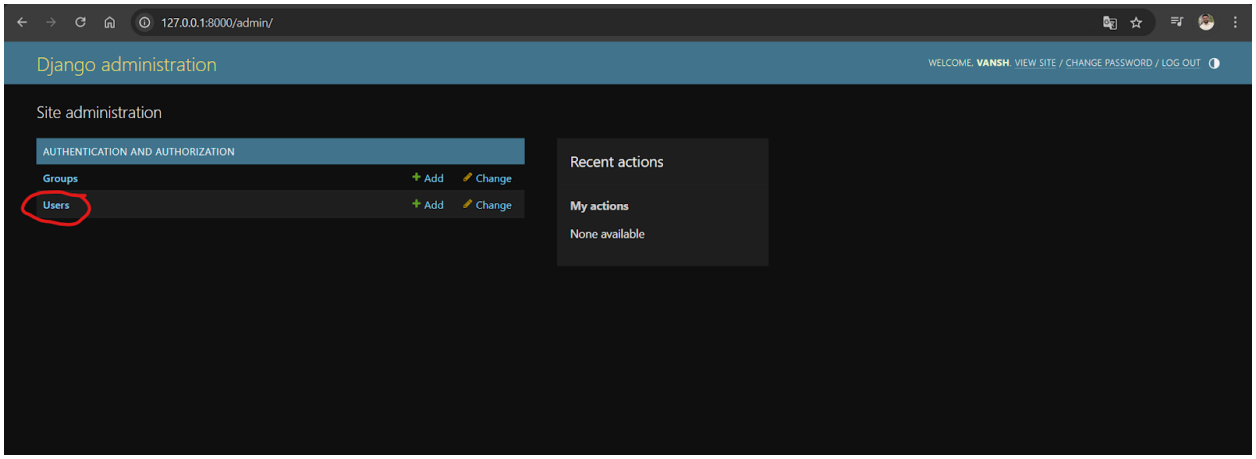
```
from django.contrib.auth.models import User
```

The line `from django.contrib.auth.models import User` imports the built-in **User** model provided by Django, which is part of the `django.contrib.auth` framework. This User model is used to handle user authentication and management in Django applications.

Breaking it down:

1. `django.contrib.auth.models`:
    - This is a module within Django's authentication framework (`auth`). The `auth` module handles common tasks such as login, logout, and user permissions. The `models` submodule provides pre-built models like `User` for managing users in the application.
2. `User`:
    - This is a pre-defined model that represents the users of your application. It comes with essential fields such as `username`, `email`, `password`, `first_name`, `last_name`, and more, and handles the core functionality required to manage users, including authentication and permissions.
    - The `User` model can be extended or customized if needed, but it provides a solid base for most applications.

Basically it imports the user from when we login to our admin url which looks like this:



We now work on models.py and then we create a class and some variables inside it.

In Django, when you create a class in `models.py`, you're defining a **model** that represents a table in the database. The variables you create inside the class are **fields** that define the columns of the table and the type of data those columns will store.

Key Points About These Variables (Fields):

- **Class variables** inside the model represent **database fields**.
- Each variable is assigned a specific field type (e.g., `CharField`, `IntegerField`, etc.) that defines the kind of data the field will store.

- These fields also include metadata, such as field size, default values, nullability, uniqueness, etc.

This is the model we defined for a single tweet that we are going to do:

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

# Create your models here.

class Tweet(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    text = models.TextField(max_length=300)
    photo = models.ImageField(upload_to='photos/', blank=True, null=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return f'{self.user.username} - {self.text[:10]}'
```

This Django model defines a `Tweet` class that represents tweets (like posts on Twitter). Here's a breakdown of each part of the code:

Imports:

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

- **`from django.db import models`**: This imports Django's model framework, allowing you to define database models.
- **`from django.contrib.auth.models import User`**: This imports Django's built-in `User` model, which represents users in your application. The `User` model handles authentication, username, password, and other user-related fields.

Class Definition:

`class Tweet(models.Model):`

- **`class Tweet(models.Model)`**: This defines the `Tweet` model, which represents a tweet (post) in the database. It inherits from `models.Model`, meaning it is a Django model and will map to a database table.

Fields (Attributes):

1. **`user = models.ForeignKey(User, on_delete=models.CASCADE)`**:
    - **Purpose**: This field creates a relationship between the `Tweet` and the `User`. Each tweet is associated with a specific user (the person who posted it).
    - **`ForeignKey`**: It's a many-to-one relationship, meaning one user can have many tweets, but each tweet belongs to only one user.
    - **`on_delete=models.CASCADE`**: This means if the user is deleted, all their tweets will also be deleted.
2. **`text = models.TextField(max_length=300)`**:
    - **Purpose**: This field stores the content of the tweet (the text). It uses `TextField`, which is designed for long text entries.
    - **`max_length=300`**: Limits the tweet's text to 300 characters.
3. **`photo = models.ImageField(upload_to='photos/', blank=True, null=True)`**:
    - **Purpose**: This field allows the user to upload a photo with the tweet.
    - **`ImageField`**: Stores an image file.
    - **`upload_to='photos/'`**: Specifies that uploaded images will be stored in the `photos/` directory within your media folder.
    - **`blank=True, null=True`**: This means the field is optional—tweets can have no photo. `blank=True` allows it to be left blank in forms, and `null=True` allows it to store a `NULL` value in the database.
4. **`created_at = models.DateTimeField(auto_now_add=True)`**:
    - **Purpose**: This field stores the date and time when the tweet was created.
    - **`auto_now_add=True`**: Automatically sets this field to the current timestamp when the tweet is created. It doesn't change afterward.
5. **`updated_at = models.DateTimeField(auto_now=True)`**:
    - **Purpose**: This field stores the date and time when the tweet was last updated.
    - **`auto_now=True`**: Automatically updates this field to the current timestamp whenever the tweet is modified.

String Representation:

```
def __str__(self):
    return f'{self.user.username} - {self.text[:10]}'
```

- **Purpose**: This defines how the object will be represented as a string (especially useful in the Django admin or the shell).
- **`f'{self.user.username} - {self.text[:10]}'`**: Returns the username of the person who posted the tweet, followed by the first 10 characters of the tweet text. This gives a brief representation of each tweet.

Summary of Fields:

- **`user`**: Links the tweet to a specific user (foreign key relationship).
- **`text`**: Contains the tweet's text content (up to 300 characters).
- **`photo`**: Allows an optional image to be uploaded with the tweet.
- **`created_at`**: Automatically tracks when the tweet is created.
- **`updated_at`**: Automatically tracks when the tweet is updated.

How It Works:

- Whenever a user creates a new tweet, a record will be inserted into the `Tweet` table in the database. The fields `created_at` and `updated_at` will be automatically managed by Django. If a user updates their tweet, only `updated_at` will be changed, and if the user is deleted, all their tweets will also be removed due to the `on_delete=models.CASCADE` rule.

Also when we run this we need to install pillow which will help us in the image uploading and storing scene. We can install pillow using the command:

python -m pip install Pillow

In Django, **migrations** are a way to apply changes to the database schema (structure). After defining or modifying models in your `models.py` file, you need to create migrations to synchronize those changes with the database.

We do that using the same command:

python manage.py makemigrations

python manage.py migrate

After all this we also register this in the admin.py file.We do it simply by importing command:

```
from .models import Tweet
```

This helps us import our Tweet from models to admin.py file.Then we register it.

```
admin.site.register(Tweet)
```

Using this saying “ admin ki site pe register krdo tweet ko”.

Now we can register and make tweets by going on to the admin panel.

So now we create a forms file named forms.py

Now we import forms and models to our forms file like:

```
from django import forms

from .models import Tweet
```

The `forms` module in Django helps you create forms that allow users to input data and submit it. Django forms automatically handle form fields, validation, and rendering in templates, making it easier to process user input.

We import `models` when we are creating forms that need to interact with the database, especially when the form is tied to a specific model. These forms are called **ModelForms**.

Then we create classes and form elements like:

```
class tweetfor(forms.ModelForm):

    class Meta:

        model = Tweet

        fields = ['text', 'photo']
```

In your `forms.py` file, you've created a **Django ModelForm** class named `tweetfor`. This class is tied to the `Tweet` model, allowing you to automatically generate a form that users can fill out to create or edit a `Tweet` object. Here's a brief explanation of what each part of your code does:

1. `class tweetfor(forms.ModelForm)`:

- You are creating a form class (`tweetfor`) that **inherits from `forms.ModelForm`**. This means the form will be automatically linked to a model, in this case, the `Tweet` model.
- This inheritance allows you to avoid manually creating form fields for every attribute in the model and rely on Django to handle it for you.

2. `class Meta:`:

- The `Meta` class inside `tweetfor` is where you specify the configuration for the form, including which model the form is based on and which fields should be included in the form.

3. `model = Tweet`:

- This line tells Django that this form is tied to the **`Tweet` model**. The form will be automatically built based on the fields in the `Tweet` model.

4. `fields = ['text', 'photo']`:

- This specifies that the form should only include the `text` and `photo` fields from the `Tweet` model.
- When you render this form, it will display input fields for:
    - **`text`**: The text content of the tweet (which is a `TextField` in the `Tweet` model).
    - **`photo`**: An optional image that the user can upload (which is an `ImageField` in the model).
- Any other fields (like `user`, `created_at`, `updated_at`) are excluded from the form, so they won't be visible or editable through this form.

Purpose of `tweetfor`:

The `tweetfor` class is designed to allow users to input or update data for the `text` and `photo` fields of a `Tweet` instance. You can use this form in views where users can submit or edit a tweet. Django will automatically handle the creation of HTML form fields for these model attributes, and when the form is submitted, it will validate and save the data to the database.

Example of Usage:

In your views, you might use this form like this:

```
def create_tweet(request):

    if request.method == 'POST':

        form = tweetfor(request.POST, request.FILES)

        if form.is_valid():

            form.save()

    else:

        form = tweetfor()


    return render(request, 'create_tweet.html', {'form': form})
```

This view will display the form, process the user's input, and save it to the database when it's valid.

Then we get back to views.py and make a lot of views.

We go to the views.py file in our app and import statements.

```
from .models import Tweet

from .forms import tweetform

from django.shortcuts import get_object_or_404
```

- You import `get_object_or_404` to **safely retrieve a model instance** (in this case, a `Tweet`) from the database.
- **Usage**:
    - This is commonly used when you want to fetch a specific tweet, but you're not sure if it exists in the database. If the tweet is found, it’s returned; if not, Django

raises a `404 Not Found` error, which is useful for handling cases where a user might try to access or update a tweet that doesn't exist.
    - It prevents your app from crashing if the object isn’t found and ensures a proper `404` response is shown to the user.

Example:

```
tweet = get_object_or_404(Tweet, pk=tweet_id)
```

After importing al this we write all the views.

```
def tweet_list(request):

    tweets = Tweet.objects.all().order_by('-created_at')

    return render(request, 'tweet_list.html', {'tweets': tweets})
```

This will help us show all the tweets

The `tweet_list` function is a **Django view** designed to display a list of tweets. Here's a brief explanation of each part:

1. `def tweet_list(request):`

- This defines a view function named `tweet_list` that takes an **HTTP request** (`request`) as its argument. This is the starting point for processing the user's request to view the list of tweets.

2. `tweets = Tweet.objects.all().order_by('-created_at')`

- **`Tweet.objects.all()`**: This retrieves all the tweet objects from the **`Tweet`** model (i.e., all the records in the `Tweet` table in the database).
- **`.order_by('-created_at')`**: The tweets are then **ordered by the `created_at` field in descending order**. The `'-created_at'` means the most recently created tweets will appear first.

This line gathers all the tweets and ensures they are sorted by their creation date, showing the newest tweets at the top of the list.

3. `return render(request, 'tweet_list.html', {'tweets': tweets})`

- **`render(request, 'tweet_list.html', {'tweets': tweets})`**:
  - The `render` function is used to generate an **HTML response** by combining the template (HTML file) with the context data (in this case, the list of `tweets`).
  - **`'tweet_list.html'`**: This is the template file (HTML) where the tweets will be displayed. It's rendered and sent back to the user's browser.
  - **`{'tweets': tweets}`**: This is the **context data**. The key `'tweets'` will be available in the template, and it contains the list of tweets retrieved earlier. The template will use this data to display the tweets on the webpage.

Now we create view functions to create, edit and delete tweets:

```
def tweet_list(request):

    tweets = Tweet.objects.all().order_by('-created_at')

    return render(request, 'tweet_list.html', {'tweets': tweets})



def tweet_create(request):

    if request.method == "POST":

        form = tweetform(request.POST, request.FILES)

        if form.is_valid():

            tweet = form.save(commit=False)

            tweet.user = request.user

            tweet.save()

            return redirect('tweet_list')

    else:

        form = tweetform()
```

```
    return render(request, 'tweet_form.html', {'form': form})


def tweet_edit(request, tweet_id):
    tweet = get_object_or_404(Tweet, pk = tweet_id, user = request.user)
    if request.method == "POST":
        form = tweetform(request.POST, request.FILES, instance=tweet)
        if form.is_valid():
            tweet = form.save(commit=False)
            tweet.user = request.user
            tweet.save()
            return redirect('tweet_list')
    else:
        form = tweetform(instance=tweet)
    return render(request, 'tweet_form.html', {'form': form})


def tweet_delete(request, tweet_id):
    tweet = get_object_or_404(Tweet, pk = tweet_id, user = request.user)
    if request.method == "POST":
        tweet.delete()
        return redirect('tweet_list')
    return render(request, 'tweet_confirm_delete.html', {'tweet': tweet})
```

1. `tweet_list` Function

```
def tweet_list(request):

    tweets = Tweet.objects.all().order_by('-created_at')

    return render(request, 'tweet_list.html', {'tweets': tweets})
```

Purpose: This view function retrieves and displays a list of all tweets.

- **`request`**: This parameter is an HttpRequest object that contains metadata about the request, including any submitted data.
- **`tweets = Tweet.objects.all().order_by('-created_at')`**:
  - This line queries the `Tweet` model to fetch all tweet entries from the database.
  - The `.order_by('-created_at')` method orders the tweets by the `created_at` timestamp in descending order (most recent tweets first).
- **`return render(...)`**:
  - This line renders the `tweet_list.html` template and passes the `tweets` context variable to it. This means the template will have access to the list of tweets, which it can display.

2. `tweet_create` Function

```
def tweet_create(request):

    if request.method == "POST":

        form = tweetform(request.POST, request.FILES)

        if form.is_valid():

            tweet = form.save(commit=False)

            tweet.user = request.user

            tweet.save()

            return redirect('tweet_list')

    else:
```

```
        form = tweetform()

    return render(request, 'tweet_form.html', {'form': form})
```

Purpose: This function handles the creation of new tweets.

- **`if request.method == "POST":`**:
  - Checks if the request is a POST request, indicating that the user submitted the form.
- **`form = tweetform(request.POST, request.FILES)`**:
  - Initializes a `tweetform` instance with the submitted data (`request.POST` for text fields and `request.FILES` for uploaded files).
- **`if form.is_valid():`**:
  - Validates the form data to ensure it meets the defined criteria (e.g., required fields are filled).
- **`tweet = form.save(commit=False)`**:
  - Creates a new `Tweet` instance but doesn't save it to the database yet (`commit=False`).
- **`tweet.user = request.user`**:
  - Sets the `user` field of the tweet to the currently logged-in user.
- **`tweet.save()`**:
  - Saves the tweet to the database.
- **`return redirect('tweet_list')`**:
  - Redirects the user to the tweet list page after successfully creating the tweet.
- **`else:`**:
  - If the request is not POST (i.e., it's a GET request), it initializes an empty `tweetform` instance.
- **`return render(...)`**:
  - Renders the `tweet_form.html` template with the form (empty or filled with errors).

3. `tweet_edit` Function

```
def tweet_edit(request, tweet_id):

    tweet = get_object_or_404(Tweet, pk=tweet_id, user=request.user)
```

```
    if request.method == "POST":
        form = tweetform(request.POST, request.FILES, instance=tweet)
        if form.is_valid():
            tweet = form.save(commit=False)
            tweet.user = request.user
            tweet.save()
            return redirect('tweet_list')
    else:
        form = tweetform(instance=tweet)
    return render(request, 'tweet_form.html', {'form': form})
```

Purpose: This function allows a user to edit an existing tweet.

- **`tweet = get_object_or_404(Tweet, pk=tweet_id, user=request.user)`**:
  - Retrieves the tweet with the given `tweet_id` that belongs to the currently logged-in user. If it doesn't exist, it returns a 404 error.
- **`if request.method == "POST":`**:
  - Checks if the request is a POST request, indicating that the user submitted the form.
- **`form = tweetform(request.POST, request.FILES, instance=tweet)`**:
  - Initializes the form with the submitted data and pre-fills it with the existing tweet instance.
- **`if form.is_valid():`**:
  - Validates the submitted form data.
- **`tweet = form.save(commit=False)`**:
  - Updates the existing `Tweet` instance but doesn't save it to the database yet.
- **`tweet.user = request.user`**:
  - Ensures the `user` field remains set to the current user (although this is redundant for editing).
- **`tweet.save()`**:
  - Saves the updated tweet to the database.
- **`return redirect('tweet_list')`**:

    - Redirects to the tweet list page after successfully editing the tweet.
- **`else:`**:
    - If the request is not POST, it initializes the form with the existing tweet data.
- **`return render(...)`**:
    - Renders the `tweet_form.html` template with the form, allowing the user to see and edit the existing tweet.

4. `tweet_delete` Function

```
def tweet_delete(request, tweet_id):

    tweet = get_object_or_404(Tweet, pk=tweet_id, user=request.user)

    if request.method == "POST":

        tweet.delete()

        return redirect('tweet_list')

    return render(request, 'tweet_confirm_delete.html', {'tweet':
tweet})
```

Purpose: This function handles the deletion of a tweet.

- **`tweet = get_object_or_404(Tweet, pk=tweet_id, user=request.user)`**:
    - Retrieves the tweet to be deleted based on the provided `tweet_id` and ensures it belongs to the currently logged-in user. If not found, it returns a 404 error.
- **`if request.method == "POST":`**:
    - Checks if the request is a POST request, indicating that the user has confirmed the deletion.
- **`tweet.delete()`**:
    - Deletes the tweet from the database.
- **`return redirect('tweet_list')`**:
    - Redirects the user to the tweet list page after deletion.
- **`return render(...)`**:
    - If the request is not POST, it renders the `tweet_confirm_delete.html` template, passing the tweet object for confirmation before deletion. This allows the user to see which tweet they are about to delete.

Summary

- **`tweet_list`**: Retrieves and displays a list of all tweets.
- **`tweet_create`**: Handles creating a new tweet, displaying a form, and saving the tweet.
- **`tweet_edit`**: Allows editing of an existing tweet, pre-filling a form with the current tweet data.
- **`tweet_delete`**: Manages the deletion of a tweet, confirming before actually deleting it.

These view functions work together to enable users to manage tweets in your application, providing a complete CRUD (Create, Read, Update, Delete) interface for tweets.

We have created all the views now then we will create all the templates that we will present on accessing different different views we will add and design jinja templates.

According to code i wrote and explained above in views.py we need to create three template files which are tweet_list.html, tweet_form.html & tweet_confirm_delete.html

First of all we will create out home page which we will create on tweet_list because it will have options of create delete edit and will also showcase all the tweets that were created.

We will also need to modify urls to showcase the tweets and all the urls we go to on clicking them.

tweet_list will look like:

```
{% extends "layout.html" %}

 {% block title %}

 Project or Tweet

 {% endblock %}



 {% block content %}

 <h1 class='text-center text-white mt-4'>Welcome to my First full stack
Project</h1>

 <a class="btn btn-primary mb-4" href="{% url 'tweet_create' %}">Create a
New Tweet</a>
```

```
<div class="container row gap-3">

   {% for tweet in tweets %}

   <div class="card" style="width: 18rem;">

       <img src="{{tweet.photo.url}}" class="card-img-top" alt="...">

       <div class="card-body">

         <h5 class="card-title">{{tweet.user.username}}</h5>

         <p class="card-text">{{tweet.text}}</p>

         <a href="{% url 'tweet_edit' tweet.id %}" class="btn
btn-primary">Edit Tweet</a>

         <a href="{% url 'tweet_delete' tweet.id %}" class="btn
btn-danger">Delete Tweet</a>

       </div>

     </div>

   {% endfor %}

</dv>

{% endblock %}
```

1. Content Block

```
{% block content %}

 <h1 class='text-center text-white mt-4'>Welcome to my First full
stack Project</h1>
```

- **`{% block content %} ... {% endblock %}`**: This is where the main content of the page goes. You are overriding the `content` block defined in `layout.html` to insert your own content.

Heading:

```
<h1 class='text-center text-white mt-4'>Welcome to my First full stack Project</h1>
```

-
    - Displays a heading in the center of the page, styled with Bootstrap classes (`text-center`, `text-white`, and `mt-4` for margin-top). This heading welcomes users to the project.

2. Create Tweet Button

```
<a class="btn btn-primary mb-4" href="{% url 'tweet_create' %}">Create a New Tweet</a>
```

- **`<a class="btn btn-primary mb-4" href="{% url 'tweet_create' %}">Create a New Tweet</a>`**:
    - This is a button that allows users to create a new tweet. It's styled as a Bootstrap button (`btn btn-primary`), and it links to the URL for creating a new tweet, which is dynamically generated using `{% url 'tweet_create' %}`. The `tweet_create` refers to the name of the view that handles tweet creation.

3. Tweet List Display

```
<dv class="container row gap-3">

    {% for tweet in tweets %}

    <div class="card" style="width: 18rem;">

        <img src="{{tweet.photo.url}}" class="card-img-top" alt="...">

        <div class="card-body">

          <h5 class="card-title">{{tweet.user.username}}</h5>

          <p class="card-text">{{tweet.text}}</p>

          <a href="{% url 'tweet_edit' tweet.id %}" class="btn btn-primary">Edit Tweet</a>
```

```
          <a href="{% url 'tweet_delete' tweet.id %}" class="btn
btn-danger">Delete Tweet</a>

        </div>

      </div>

    {% endfor %}

</dv>
```

- **`<dv class="container row gap-3">`**: This is a typo—it should be `<div>`. This `<div>` element contains a row of tweet cards, using Bootstrap's `container` and `row` classes to arrange the cards in a grid with some gap between them.
- **`{% for tweet in tweets %}`**: This is a Django template tag that loops through the list of tweets passed to the template (from the `tweet_list` view function). For each `tweet` in the `tweets` list, it generates a card.

Inside the Loop:

Each tweet is displayed in a Bootstrap card component:

- **`<div class="card" style="width: 18rem;">`**: This creates a Bootstrap card for each tweet, with a fixed width of 18rem.
- **`<img src="{{tweet.photo.url}}" class="card-img-top" alt="...">`**: Displays the tweet's image (photo) if available. The image is pulled from the `tweet.photo.url`, which is the URL of the uploaded photo.
- **`<h5 class="card-title">{{tweet.user.username}}</h5>`**: Displays the username of the tweet's author. `tweet.user.username` accesses the `User` model (linked via `ForeignKey`) and fetches the `username` of the user who posted the tweet.
- **`<p class="card-text">{{tweet.text}}</p>`**: Displays the tweet's text. The `{{tweet.text}}` is dynamically inserted, showing the content of the tweet.
- **Edit and Delete Buttons**:

Edit:
`<a href="{% url 'tweet_edit' tweet.id %}" class="btn btn-primary">Edit Tweet</a>`

    -
        - This button links to the tweet editing page. The URL is dynamically generated using `{% url 'tweet_edit' tweet.id %}`, where

`tweet.id` is the unique identifier for the tweet. It leads to the view that allows users to edit the specific tweet.

Delete:

```
<a href="{% url 'tweet_delete' tweet.id %}" class="btn
btn-danger">Delete Tweet</a>
```

- This button links to the tweet deletion page. It uses `{% url 'tweet_delete' tweet.id %}`, where `tweet.id` is the unique ID for the tweet. It leads to the view that allows users to delete the tweet.

4. End of Blocks

```
{% endblock %}
```

Also we edit urls.py like:

```
from django.urls import path

from . import views



urlpatterns = [

    path('', views.tweet_list, name='tweet_list'),

    path('create/', views.tweet_create, name='tweet_create'),

    path('<int:tweet_id>/delete/', views.tweet_delete,
name='tweet_delete'),

    path('<int:tweet_id>/edit/', views.tweet_edit, name='tweet_edit'),

]
```

1. Tweet List View

```
path('', views.tweet_list, name='tweet_list'),
```

- `''`: Represents the root URL of the app (e.g., `/tweets/`).
- **`views.tweet_list`**: Maps the URL to the `tweet_list` view function, which will display the list of all tweets.
- **`name='tweet_list'`**: Assigns a name to this URL pattern, which makes it easy to reference in templates and other parts of the code.

2. Create Tweet View

```
path('create/', views.tweet_create, name='tweet_create'),
```

- **`'create/'`**: The URL pattern for creating a new tweet (e.g., `/tweets/create/`).
- **`views.tweet_create`**: Links to the `tweet_create` view function, which will handle the tweet creation process.
- **`name='tweet_create'`**: Name given to this URL pattern for easy referencing in templates.

3. Delete Tweet View

```
path('<int:tweet_id>/delete/', views.tweet_delete,
name='tweet_delete'),
```

- **`'<int:tweet_id>/delete/'`**: URL pattern for deleting a specific tweet (e.g., `/tweets/5/delete/`).
    - `<int:tweet_id>`: This part captures the tweet's unique `id` as an integer from the URL and passes it to the view function.
- **`views.tweet_delete`**: Maps to the `tweet_delete` view function, which will delete the tweet.
- **`name='tweet_delete'`**: The name given to this pattern for referencing it elsewhere (e.g., in templates for the delete button).

4. Edit Tweet View

```
path('<int:tweet_id>/edit/', views.tweet_edit, name='tweet_edit'),
```

- **`'<int:tweet_id>/edit/'`**: URL pattern for editing a specific tweet (e.g., `/tweets/5/edit/`).
    - `<int:tweet_id>`: Captures the tweet's `id` as an integer from the URL and passes it to the `tweet_edit` view.
- **`views.tweet_edit`**: Maps to the `tweet_edit` view function, which handles the tweet editing process.
- **`name='tweet_edit'`**: The name of this URL pattern for referencing in templates (e.g., the edit button).

With this our urls.py is now complete and also our tweet_list is complete now we work on the rest of the two layouts.

Our create and edit tweet html template looks like this:

```
{% extends "layout.html" %}

 {% block title %}

 Project or Tweet

 {% endblock %}



 {% block content %}

 <h1 class='text-center text-white mt-4'>Welcome to my First full stack
Project</h1>

 <h2> {%if form.instance.pk %}Edit Tweet {% else %} Create Tweet{% endif
%}

 <form method="post" enctype="multipart/form-data" class="form">

    {% csrf_token %}

    {{ form.as_p }}
```

```
    <button class="btn btn-warning" type="submit">Submit</button>

 </form>

 <a href="{% url 'tweet_list' %}">Back to tweet page</a>

</h2>

 {% endblock %}
```

Content Block

`{% block content %}`

- This defines the content that will be injected into the content area of the `layout.html` file.

Heading

`<h1 class='text-center text-white mt-4'>Welcome to my First full stack Project</h1>`

- Displays a centered heading welcoming users to the project.

Conditional Form Heading

`<h2> {%if form.instance.pk %}Edit Tweet {% else %} Create Tweet{% endif %}`

- Checks if the form corresponds to an existing tweet (`form.instance.pk`):
  - If it exists, it shows "Edit Tweet."
  - If it's a new form (no instance), it shows "Create Tweet."

Form for Creating/Editing a Tweet

```
<form method="post" enctype="multipart/form-data" class="form">

   {% csrf_token %}

   {{ form.as_p }}
```

```
   <button class="btn btn-warning" type="submit">Submit</button>

</form>
```

- **`<form method="post" enctype="multipart/form-data">`**: A form is being created that allows file uploads (for the photo).
- **`{% csrf_token %}`**: Adds a security token to prevent Cross-Site Request Forgery (CSRF) attacks.
- **`{{ form.as_p }}`**: Renders the form fields (text, photo) as paragraph elements.
- **`<button>`**: A submit button styled with Bootstrap to allow the user to submit the form.

Link to Tweet List

```
<a href="{% url 'tweet_list' %}">Back to tweet page</a>
```

- Provides a link to go back to the tweet list page (`tweet_list` view) when clicked.

Now we edit our deletion page.

Our tweet_confirm_delete file looks like this:

```
{% extends "layout.html" %}

 {% block title %}

 Project or Tweet

 {% endblock %}



 {% block content %}

 <h1 class='text-center text-white mt-4'>Are you sure that you want to
delete the tweet?</h1>

 <form method="post">

    {% csrf_token %}
```

```
    <button class="btn btn-danger">Delete</button>

    <a class="btn btn-success" href="{% url 'tweet_list' %}">Cancel</a>

</form>

{% endblock %}
```

Content Block Explanation:

Heading

```
<h1 class='text-center text-white mt-4'>Are you sure that you want to delete the tweet?</h1>
```

- Displays a prominent heading that asks the user if they are sure about deleting the tweet. The text is centered, white, and has a margin at the top for spacing (`mt-4`), using Bootstrap classes.

Form for Deleting the Tweet

```
<form method="post">

    {% csrf_token %}

    <button class="btn btn-danger">Delete</button>

    <a class="btn btn-success" href="{% url 'tweet_list' %}">Cancel</a>

</form>
```

- **Form**: The form is used to handle the deletion of the tweet. It uses the POST method to securely send data to the server.
- **`{% csrf_token %}`**: Adds a Cross-Site Request Forgery token to the form for security purposes.
- **Delete Button**: A red ("danger") button is provided for confirming the deletion. It submits the form and triggers the tweet deletion in the backend.
- **Cancel Button**: A green ("success") button is included as a link to the tweet list page. If the user clicks it, they are redirected back to the main list of tweets without deleting anything.

Now we try to make our website more secure and also secure some paths like only authenticated or the person who has log in can delete and edit the tweets on the person posted and not any other persons tweets.

We keep the tweet_lsit path as it is because we want people to view all the tweets without login or registration but we will secure other paths so that we can make user authentication and also add user login and register.

We bring out decorators to give auth and give login required using:

```
from django.contrib.auth.decorators import login_required
```

The line `from django.contrib.auth.decorators import login_required` is used to import the `login_required` decorator from Django's authentication system. The `login_required` decorator is a feature provided by Django that ensures a user must be logged in to access a specific view. Here's a detailed breakdown:

Explanation:

1. **`from django.contrib.auth.decorators`**:
    - This is the module from which `login_required` is imported. It contains decorators related to user authentication.
    - **`django.contrib.auth`** is Django's built-in authentication framework, which handles tasks such as user login, registration, password management, etc.
2. **`login_required` decorator**:
    - **Decorator**: A decorator is a Python feature that modifies the behavior of a function. It is placed just above a function definition with the `@` symbol and alters the behavior of that function when called.
    - **Purpose**: `login_required` is used to ensure that only authenticated users (users who have logged in) can access a particular view in your application.
    - If an unauthenticated user (i.e., a user who is not logged in) tries to access the view decorated with `@login_required`, they will be redirected to the login page.

Then we put

```
@login_required
```

How to Use It:

To restrict access to certain views only for logged-in users, you apply the `login_required` decorator like this:

```
from django.contrib.auth.decorators import login_required


@login_required
def tweet_create(request):
    # Logic to create a tweet
    ...
```

In this example:

- **`@login_required`** ensures that only users who are logged in can access the `tweet_create` view.
- If a user is not logged in, they will automatically be redirected to the login page (by default, `/accounts/login/`).

Now we add user registration and logins.

First of all we add a registration function in views.py like this:

```
def register(request):
    if request.method == "POST":
        form = UserRegistrationForm(request.POST)
        if form.is_valid():
```

```
            user = form.save(commit=False)

            user.set_password(form.cleaned_data['password1'])

            user.save()

            login(request, user)

            return redirect('tweet_list')

    else:

        form = UserRegistrationForm()

    return render(request, 'registration/register.html', {'form': form})
```

This `register` function is used to handle user registration in a Django web application. It handles both the form display and the form submission for registering a new user. Here's a breakdown of how it works:

Function Overview:

- **Purpose**: To register a new user when they submit a registration form. If the form data is valid, it creates a new user, logs them in, and redirects them to another page. If the form isn't submitted (i.e., a GET request is made), it simply displays the registration form.

Key Components:

1. **request**:
    - The `request` object is the HTTP request sent by the browser. It contains information like whether the request is a `POST` (form submission) or a `GET` (form display).
2. **if request.method == "POST"**:
    - This checks if the request is a `POST`. A `POST` request happens when the user submits the form (i.e., sends their registration data).
    - If it's a `POST` request, it means the user has filled out the registration form and submitted it.
3. **form = UserRegistrationForm(request.POST)**:
    - This creates an instance of the `UserRegistrationForm` form class, passing in the submitted form data (`request.POST`).
    - `UserRegistrationForm` is a form designed to handle user registration details, including fields like username, password, and others.

4. **if form.is_valid()**:
    - This checks if the form data is valid. Django will validate the form based on the rules defined in `UserRegistrationForm`. For example, it will check if the passwords match, if the username is unique, etc.
5. **user = form.save(commit=False)**:
    - If the form is valid, this line creates a `User` object from the form data but does **not** save it to the database yet (`commit=False` delays saving).
    - The `form.save()` method creates a new instance of the `User` model, using the cleaned data from the form.
6. **user.set_password(form.cleaned_data['password1'])**:
    - Instead of directly saving the password as plain text, this line hashes the password (using Django's built-in password hashing method) for security.
    - The password is set using the cleaned data from the form (where `password1` is the first password field in the form).
7. **user.save()**:
    - After setting the hashed password, the user object is saved to the database.
    - Now the new user is fully registered in the database.
8. **login(request, user)**:
    - Once the user is successfully saved, this function logs in the newly registered user. This means that after registration, the user does not need to log in again manually.
    - The `login` function is provided by Django to handle user sessions.
9. **return redirect('tweet_list')**:
    - After logging the user in, the function redirects them to another page, in this case, the page with the URL name `'tweet_list'`.
    - This prevents the form from being submitted again if the user refreshes the page.
10. **else**:
    - If the request is **not** a `POST` (i.e., the user hasn't submitted the form yet, such as when they first visit the registration page), the function creates an empty `UserRegistrationForm` object.
11. **return render(request, 'registration/register.html', {'form': form})**:
    - The function renders the registration page (template `'registration/register.html'`) and passes the `form` to the template.
    - If it's the first time the user is visiting the page (a `GET` request), an empty form is rendered.
    - If the form has errors (i.e., invalid submission), the form will retain the entered data and display validation errors.

Summary:

- If the request method is `POST`, the form data is validated, and if valid, a new user is created, logged in, and redirected to another page.

- If the request method is not `POST` (i.e., it's a `GET` request), a blank registration form is rendered.
- The `UserRegistrationForm` handles the user input, and the password is securely hashed before saving the user to the database.

We also modify the import statements:

```
from django.shortcuts import render

from .models import Tweet

from .forms import tweetform, UserRegistrationForm

from django.shortcuts import get_object_or_404, redirect

from django.contrib.auth.decorators import login_required

from django.contrib.auth import login
```

Then we go to urls.py to register this view function:

```
path('register/', views.register, name='register'),
```

We also create a forms in forms.py like this:

```
class UserRegistrationForm(UserCreationForm):

    email = forms.EmailField

    class Meta:

        model = User

        fields = ('username', 'email', 'password1', 'password2')
```

This code defines a custom user registration form in Django by extending Django's built-in `UserCreationForm`. This form is designed to handle user registration, collecting information such as username, email, and passwords. Here's a detailed explanation:

Form Breakdown:

```
class UserRegistrationForm(UserCreationForm):

    email = forms.EmailField

    class Meta:

        model = User

        fields = ('username', 'email', 'password1', 'password2')
```

Components:

1. **`class UserRegistrationForm(UserCreationForm):`**:
    - This defines a new form class called `UserRegistrationForm`, which inherits from `UserCreationForm`.
    - `UserCreationForm` is a built-in Django form used specifically for creating new users. It includes fields for the username and password management (i.e., `password1`, `password2`).
2. **`email = forms.EmailField`**:
    - This adds an email field to the form, making it a required field for user registration.
    - `forms.EmailField` is a Django form field that ensures the user inputs a valid email address.
    - Note: This line should call the field as `forms.EmailField()` (with parentheses), or it won't function correctly.

Corrected version:
```
email = forms.EmailField()
```

3.
4. **`class Meta:`**:
    - The `Meta` class defines metadata for the form, specifically the model the form is based on and the fields it should include.
5. **`model = User`**:

- This specifies that the form is linked to the `User` model. The `User` model is Django's default user model, used for authentication.
- When the form is processed, it will create or modify a `User` object in the database.

6. **`fields = ('username', 'email', 'password1', 'password2')`**:
   - This defines which fields from the `User` model and form should be included in the form.
   - The form will contain the following fields:
     - `username`: The username of the user (a field that comes from the `User` model).
     - `email`: The email address of the user (a field added explicitly in the form).
     - `password1`: The first password input (handled by the `UserCreationForm`).
     - `password2`: The confirmation password input (handled by the `UserCreationForm`).

Summary:

- This `UserRegistrationForm` is a custom form for registering new users. It extends Django's built-in `UserCreationForm` to include the email field.
- The form is linked to Django's `User` model and includes fields for the username, email, and two password fields (`password1` for the password and `password2` for confirming the password).
- By inheriting from `UserCreationForm`, this form automatically gets password validation and management functionality (like ensuring the two password fields match).

Then we go to our main projects settings.py and add the redirected urls to the project:

```
LOGIN_URL = '/accounts/login'



LOGIN_REDIRECT_URL = '/tweet/'

LOGOUT_REDIRECT_URL = '/tweet/'
```

In Django, these settings control the behavior of the login and logout processes, specifically where the user is redirected after they log in or log out. Here's a detailed explanation of what each of these lines does and why you added them:

1. `LOGIN_URL = '/accounts/login'`

- **What it does**: This setting specifies the URL where Django will redirect users if they try to access a restricted view (a view that requires login) without being logged in.
- **Why you added it**:
    - If a user attempts to access a page that requires authentication (e.g., viewing tweets) but is not logged in, they will be redirected to this login URL.
    - In this case, the login page is located at `/accounts/login`, so if users are not authenticated, Django will redirect them to this URL.

2. `LOGIN_REDIRECT_URL = '/tweet/'`

- **What it does**: This setting determines where a user is redirected after they log in successfully.
- **Why you added it**:
    - After a successful login, you want users to be taken to the page that lists tweets (in this case, `/tweet/`).
    - This improves user experience by taking them directly to the main part of the app after logging in, instead of keeping them on the login page or some other default location.

3. `LOGOUT_REDIRECT_URL = '/tweet/'`

- **What it does**: This setting defines where users are redirected after they log out.
- **Why you added it**:
    - After a user logs out, they will be redirected to the tweet list page (`/tweet/`).
    - This ensures that after logging out, the user is still directed to a meaningful page, possibly one that can still be viewed by non-authenticated users.

Why These Are Important:

- **User Navigation**: These settings are critical for controlling user flow and providing a smooth experience during login and logout actions.
- **Login**: Users need to be redirected to a login page if they're not authenticated, which is why `LOGIN_URL` is set.
- **Post-login Redirection**: After successfully logging in, users are typically redirected to a relevant page (like the tweet list) instead of staying on the login form.

- **Post-logout Redirection**: After logging out, you want users to land on a meaningful page instead of an empty or confusing screen, so you redirect them to a non-restricted page like the tweet list.

Summary:

- `LOGIN_URL`: Redirects unauthenticated users to the login page when they try to access protected views.
- `LOGIN_REDIRECT_URL`: Redirects users to the tweet list after a successful login.
- `LOGOUT_REDIRECT_URL`: Redirects users to the tweet list after logging out.

These settings ensure smooth navigation in your app during authentication processes.

After all this we go to our main urls .py and add

```
path('accounts/', include('django.contrib.auth.urls')),
```

The line:

`path('accounts/', include('django.contrib.auth.urls')),`

in your `urls.py` is used to include Django's built-in authentication system's URL patterns (such as login, logout, password reset, etc.) under the `/accounts/` URL path. Here's a detailed breakdown of what this does and why you added it:

Breakdown:

1. **`path('accounts/', ...)`**:
   - **What it does**: This sets the base URL path as `/accounts/`. Any authentication-related URL will be accessed under this path.
   - For example:
     - The login page will be at `/accounts/login/`.
     - The logout page will be at `/accounts/logout/`.
     - Password reset will be at `/accounts/password_reset/`, and so on.
2. **`include('django.contrib.auth.urls')`**:
   - **What it does**: This includes all the default authentication-related URL patterns provided by Django's `auth` app. Django provides a set of built-in views for user authentication, such as:

        - **Login**: `/accounts/login/`
        - **Logout**: `/accounts/logout/`
        - **Password Change**: `/accounts/password_change/`
        - **Password Reset**: `/accounts/password_reset/`
    - By including this, you are telling Django to use its built-in views and URL patterns for handling these authentication tasks. You don't need to create these views or URLs manually; Django handles it for you.

Why You Added This:

1. **Pre-built Authentication Views**:
    - Django provides ready-to-use views and templates for common authentication tasks like login, logout, password change, and password reset. By adding this line, you're including all these default functionalities in your project without needing to write custom views or URL patterns for these actions.
2. **Convenience**:
    - Instead of manually coding the login/logout/password reset views and URL patterns, Django gives you these tools out of the box. This allows you to quickly set up user authentication.
3. **URL Structure**:
    - By placing it under the `/accounts/` path, you're giving all authentication-related views a consistent URL structure. For example:
        - `/accounts/login/` for logging in.
        - `/accounts/logout/` for logging out.
        - `/accounts/password_change/` for changing passwords.
    - This is a clean and organized way to handle user authentication URLs.

Included URLs:

When you add `include('django.contrib.auth.urls')`, the following default URLs are included:

- `login/`: Displays the login form.
- `logout/`: Logs out the user.
- `password_change/`: Displays the form to change the user's password.
- `password_change/done/`: Confirms that the password was successfully changed.
- `password_reset/`: Displays the form to reset the password via email.
- `password_reset/done/`: Confirms that a password reset email was sent.
- `reset/<uidb64>/<token>/`: A URL for the link sent in the password reset email.
- `reset/done/`: Confirms that the password was successfully reset.

Summary:

- The line `path('accounts/', include('django.contrib.auth.urls'))` automatically includes Django's built-in authentication system URLs (like login, logout, password reset) under the `/accounts/` URL path.
- You added this to quickly provide user authentication functionalities without writing custom code for these actions.

Which helps us create a logical path for the login and registration path.

Now even after all this we will still see a blank page that is because we havent made anything on our login.html page.

Now we create all three login logout and register files:

Loginfile:

```
{% extends "layout.html" %}



{% block content %}

<h2>login form</h2>

<form method="POST">

    {% csrf_token %}

    {{form.as_p}}

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

</form>

<p>Dont have an Account?<a href="{% url 'register' %}">Register
Here</a></p>

{% endblock %}
```

Code Breakdown:

```
{% extends "layout.html" %}
```

- **What it does**: This line tells Django to extend (inherit from) a base template called `layout.html`. The `layout.html` file likely contains common structure elements like the header, footer, and navigation, so you don't have to rewrite them in every template.

```
{% block content %}
```

- **What it does**: This defines a block of content that will be inserted into the corresponding block in the `layout.html` file. The content of this block is specific to this page (the login form).

```
<h2>login form</h2>

<form method="POST">

    {% csrf_token %}

    {{form.as_p}}

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

</form>
```

- **`<h2>login form</h2>`**: This is a heading for the login form.
- **`<form method="POST">`**: This starts an HTML form with the `POST` method, which is typically used to submit data (in this case, login information) securely to the server.
- **`{% csrf_token %}`**: This adds a CSRF (Cross-Site Request Forgery) token for security purposes. It's required in Django forms to protect against CSRF attacks.
- **`{{form.as_p}}`**: This renders the form passed in the context (likely a Django form object) with each field wrapped in `<p>` tags for simple display. The form will likely include fields for the username and password.

- **`<button class="btn btn-primary" type="submit">Submit</button>`**: This adds a submit button styled with Bootstrap's "primary" button style (if Bootstrap is included in your project).

```
<p>Dont have an Account?<a href="{% url 'register' %}">Register Here</a></p>
```

- **What it does**: This paragraph provides a link for users who don't have an account to register.
- **`{% url 'register' %}`**: This uses Django's `{% url %}` template tag to generate the URL for the registration page. The `'register'` refers to the name of the URL pattern defined in your `urls.py`.

```
{% endblock %}
```

- **What it does**: This closes the `content` block. Everything between `{% block content %}` and `{% endblock %}` will be inserted into the corresponding block in the `layout.html` file.

Summary:

- The template extends a base layout and defines a login form.
- The form includes a CSRF token for security and renders the login fields dynamically with `{{form.as_p}}`.
- There is a link directing users to the registration page if they don't have an account.
- The form uses the `POST` method, and the submit button is styled using Bootstrap classes.

Logout:

```
{% extends "layout.html" %}



{% block content %}

<h2>logged out form</h2>



<p>You have been logged out<a href="{% url 'login' %}">login again</a></p>

{% endblock %}
```

Code Breakdown:

`{% extends "layout.html" %}`

- **What it does**: This line tells Django that this template inherits from `layout.html`, which is the base template. The base template likely contains shared elements like headers, footers, and navigation menus.

`{% block content %}`

- **What it does**: This opens a block of content that will be placed into the `content` block defined in `layout.html`. This block will contain the logout confirmation message.

`<h2>logged out form</h2>`

- **What it does**: This is a heading for the page, simply indicating that the user is viewing a "logged out" confirmation page.

```
<p>You have been logged out<a href="{% url 'login' %}">login
again</a></p>
```

- **What it does**:
  - This displays a message saying, "You have been logged out."
  - The text "login again" is a clickable link that directs the user to the login page if they want to log in again.
  - **`{% url 'login' %}`**: The `{% url %}` template tag dynamically generates the URL for the login page. `'login'` refers to the URL pattern for the login view (which should be defined in `urls.py`).

```
{% endblock %}
```

- **What it does**: This closes the `content` block. Everything within the `{% block content %}` and `{% endblock %}` tags will be inserted into the base template (`layout.html`) where the `content` block is defined.

Summary:

- This template extends a base layout (`layout.html`) and informs the user that they have successfully logged out.
- It displays a message and provides a link for users to log in again using Django's built-in URL pattern for the login page.
- The template maintains a simple and clean structure for the logout confirmation page.

Register:

```
{% extends "layout.html" %}



{% block content %}

<h2>register form</h2>

<form method="POST">

    {% csrf_token %}

    {{form.as_p}}

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

</form>

{% endblock %}
```

Code Breakdown:

`{% extends "layout.html" %}`

- **What it does**: This line tells Django that this template extends (inherits from) a base template called `layout.html`. The base template likely contains shared elements like the header, footer, or navigation.

`{% block content %}`

- **What it does**: This begins a block of content that will be inserted into the `content` block in the `layout.html` template. Everything between `{% block content %}` and `{% endblock %}` is specific to this registration page.

```
<h2>register form</h2>
```

- **What it does**: This is a heading for the page that indicates the user is on the registration form page.

```
<form method="POST">
    {% csrf_token %}
    {{form.as_p}}
    <button class="btn btn-primary" type="submit">Register</button>
</form>
```

- **`<form method="POST">`**: This opens an HTML form that uses the `POST` method, which is typically used for submitting data securely to the server.
- **`{% csrf_token %}`**: This tag adds a CSRF (Cross-Site Request Forgery) token for security. Django requires this token in all forms that modify data to prevent CSRF attacks.
- **`{{form.as_p}}`**: This renders the registration form (which is passed in the template context) with each field wrapped in a `<p>` tag, making the form fields automatically generated and displayed properly. The form likely contains fields like username, email, password, etc.
- **`<button class="btn btn-primary" type="submit">Register</button>`**: This adds a submit button for the form, styled using Bootstrap's "primary" button class, making it appear as a blue button.

```
{% endblock %}
```

- **What it does**: This closes the `content` block. Everything within this block will be inserted into the corresponding block in the `layout.html` template.

Summary:

- This template creates a user registration page, allowing users to register by filling out the form.
- The form includes a CSRF token for security and uses `{{form.as_p}}` to render the form fields.
- A styled "Register" button is provided for submitting the form.
- It inherits common elements from the `layout.html` base template.

With this we get all the three pages and i made a new account using these credentials:

Username:newuser

Password:myfirsttime

email:newuser@gmail.com

Now we edit the deletion part for non users

We simply inject a lot so that we can access and not access the things.first of all we go to our main tweet_list.html file and add:

```
{% if tweet.user == user %}

          <a href="{% url 'tweet_edit' tweet.id %}" class="btn
btn-primary">Edit Tweet</a>

          <a href="{% url 'tweet_delete' tweet.id %}" class="btn
btn-danger">Delete Tweet</a>

          {% endif %}
```

Above our edit and delete buttons they ensure that only the logged in users can edit and delete and no other user can delete other users tweet.
Similarly we want the logged out users to see login option and logged in users to see the logout option and we do that by going to out main layout.html which is giving our main layout and then:

```
<a href="{% url 'tweet_list' %}" class="btn btn-primary mx-2">Tweet
Home</a>

            {% if user.is_authenticated %}

            <form method="post" action="{% url 'logout' %}">

              {% csrf_token %}

              <button class="btn btn-danger" type="submit">Logout</button>

            </form>

            {% else %}

            <a href="{% url 'register' %}" class="btn btn-primary
mx-2">Register</a>

            <a href="{% url 'login' %}" class="btn btn-success
mx-2">Login</a>

            {% endif %}
```

Add these three buttons which gives logout to logged in users and login to logged out users and a button which will directly take us to the main tweet_list page and display all the tweets now the main problem is that i want to add a main button on my home page which leads me directly to this app and we dont see any error on the home page.

Did it by creating a home.html and then going to views and creating a view called home then importing it into the main urls.py file and adding that home.html file on the urls.py and it was that simple.

-Vansh Bhardwaj

-vanshbhardwaj911@gmail.com

-7827046728