Django

Disclaimer:

These notes have been created by *Khushal Savalakha* (khushal.sav@gmail.com) and are based on practical applications and personal understanding of the concepts. Although I have thoroughly checked and verified the information and code provided, there is a possibility that some aspects may contain errors or inaccuracies. Please use this content as a reference and verify details independently as needed. I am sharing these notes as part of a GitHub repository to help others learn, and I hope they are beneficial.

▼ Introduction

Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. It was developed to help developers take applications from concept to completion as swiftly as possible, with an emphasis on reusability, pluggability, and the principle of "don't repeat yourself" (DRY). With a built-in admin interface, powerful ORM, and robust security features, Django has become a popular choice for building web applications.

```
web applications
freeware
open source--->source code is open for all


It is free and opensource web framework
It is written in Python

MVC architecture pattern
M--->Model
V--->View
C--->Controller

Django follows MVT(Model-View-Template) architectural pattern.
Django is maintained by DSF(Django Software Foundation)
  Python--->PSF

```

Sure, here’s a more detailed comparison of the Model-View-Controller (MVC) and Model-View-Template (MVT) architectural patterns:

Aspect	MVC (Model-View-Controller)	MVT (Model-View-Template)

Framework Examples	Commonly used in frameworks like Ruby on Rails, Spring MVC, and ASP.NET MVC.	Primarily used in the Django web framework.
Model	**Role**: Manages the data and business logic of the application. Defines the data structure and interactions with the database. **Example**: In Rails, an Active Record class represents the model, interacting with the database through ORM.	**Role**: Manages the data and business logic of the application. Defines the data structure and interactions with the database. **Example**: In Django, a model is defined as a Python class that inherits from `models.Model` and interacts with the database through Django's ORM.
View	**Role**: Responsible for displaying data to the user. Renders the user interface and presents data fetched by the controller. **Example**: In Rails, views are typically `.erb` files containing HTML with embedded Ruby code for dynamic content.	**Role**: Handles both the presentation logic and selects the appropriate template to render. **Example**: In Django, the view is a function or a class that processes requests, fetches data from models, and renders a template with the context data.
Controller	**Role**: Acts as an intermediary between models and views. Processes user inputs, calls the model to get data, and returns the view with the data. **Example**: In Rails, controllers are Ruby classes inheriting from `ActionController::Base`, defining actions like `index`, `show`, `create`, etc.	**Role**: Not explicitly present as a separate component. The view in Django handles the responsibilities of both the view and the controller in MVC. **Example**: The view function in Django takes over the role of the controller by handling the request, interacting with the model, and rendering the template.
Template	**Role**: Not a separate component. Typically integrated within the view to generate the final HTML output. **Example**: In Rails, embedded Ruby code in the view files (`.html.erb`) generates dynamic content.	**Role**: Defines the structure of the output that is rendered. Responsible for displaying data passed from the view. **Example**: In Django, templates are HTML files with embedded Django Template Language (DTL) to render dynamic content using context data provided by the view.

Khushal Savalakha

Separation of Concerns	**Description**: Clearly separates data (model), business logic (controller), and presentation (view). Each component has a distinct responsibility, making the codebase modular and maintainable. **Example**: The controller updates the model and selects a view to render based on user input.	**Description**: Separates data (model), presentation logic (view), and presentation structure (template). The view handles both fetching data from the model and passing it to the template for rendering. **Example**: The view function processes the request, interacts with the model, and renders the template with context data.
Routing	**Description**: URLs map to controller actions, determining which controller and action method to invoke based on the URL pattern. **Example**: In Rails, routes are defined in the `config/routes.rb` file, mapping URL patterns to specific controller actions.	**Description**: URLs map directly to view functions or class-based views, which handle the request and response cycle. **Example**: In Django, routes are defined in the `urls.py` file, mapping URL patterns to specific view functions or classes.
Component Interaction	**Description**: The controller receives user input, updates the model, and returns the view with updated data. **Example**: The controller interacts with the model to fetch data and passes it to the view, which renders the final output to the user.	**Description**: The view function or class receives user input, interacts with the model to fetch data, and uses the template to render the final output. **Example**: The view function in Django processes the request, fetches data from the model, and passes it to the template for rendering.
Main Focus	**Description**: Emphasizes the flow of data between the user interface and the model through the controller. Each component has a clear and distinct role, enhancing modularity and testability. **Example**: The controller handles business logic and updates the view based on the model's state.	**Description**: Emphasizes the connection between data (model) and presentation (template) through the view. The view in Django plays a central role, managing both business logic and presentation, simplifying the architecture. **Example**: The view handles business logic and renders the template with context data.

Summary

Khushal Savalakha

- **MVC** separates the application into three distinct components: Model, View, and Controller, each with its specific role and responsibilities.
- **MVT** in Django combines the responsibilities of the view and controller into a single component called the view, while templates handle the rendering of the final output. This simplifies the architecture by reducing the number of components and focusing on the interaction between data and presentation.

💡 what is the meaning of Model-View-Template(MVT) architectural pattern ?

The **Model-View-Template (MVT)** architectural pattern is a design pattern used in web development, particularly in Django, a popular Python web framework. It separates the concerns of a web application into three interconnected components:

1. **Model**:
   - Represents the data structure and business logic of the application.
   - Defines the database schema and manages the database interactions.
   - Handles data validation, querying, and manipulation.
   - In Django, models are defined as Python classes that map to database tables.
2. **View**:
   - Manages the user interface and application logic.
   - Receives HTTP requests, processes them, and returns HTTP responses.
   - Retrieves data from the model, applies business logic, and selects a template to render the response.
   - In Django, views are Python functions or classes that handle HTTP requests and responses.
3. **Template**:
   - Defines the presentation layer, responsible for rendering HTML and displaying data to the user.
   - Uses placeholders and logic to dynamically generate HTML based on the data passed from the view.
   - In Django, templates are HTML files with embedded Django Template Language (DTL) for dynamic content rendering.

Khushal Savalakha

How MVT Works Together

1. **User Request**:
   - A user sends an HTTP request to the web server (e.g., accessing a webpage).
2. **View Handling**:
   - The request is routed to the appropriate view based on the URL configuration.
   - The view processes the request, interacts with the model to retrieve or update data, and applies business logic.
3. **Model Interaction**:
   - The view interacts with the model to perform database operations, such as querying for specific records or updating data.
4. **Template Rendering**:
   - The view selects a template and passes the retrieved data to it.
   - The template uses the data to render an HTML response, dynamically generating content based on the provided data.
5. **Response**:
   - The rendered HTML is returned to the user's browser as an HTTP response, displaying the requested page or result.

Example

Model:

```
# models.py
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()
```

View:

```
# views.py
from django.shortcuts import render
from .models import Book
```


Khushal Savalakha

```
def book_list(request):
    books = Book.objects.all()
    return render(request, 'book_list.html', {'books': bo
oks})
```

Template:

```
<!-- book_list.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Book List</title>
</head>
<body>
    <h1>Book List</h1>
    <ul>
        {% for book in books %}
            <li>{{ book.title }} by {{ book.author }}</li
>
        {% endfor %}
    </ul>
</body>
</html>
```

In this example:

- The **Model** defines a `Book` class with attributes like `title`, `author`, and `published_date`.
- The **View** function `book_list` retrieves all book records from the database and passes them to the template.
- The **Template** `book_list.html` dynamically generates an HTML list of books, displaying their titles and authors.

The MVT pattern ensures a clean separation of concerns, making the codebase more maintainable, scalable, and easier to understand.

Khushal Savalakha

Top 5 Features of Django Framework:

Django was invented to meet fast-moving newsroom deadlines, while satisfying the tough requirements of experienced Web developers.

The following are main important features of Django

1) Fast:

Django was designed to help developers take applications from concept to completion as quickly as possible.

2) Fully loaded:

Django includes dozens of extras we can use to handle common Web development tasks. Django takes care of user authentication, content administration, site maps, RSS feeds, and many more tasks.

3) Security:

Django takes security seriously and helps developers avoid many common security mistakes, such as SQL injection, cross-site scripting, cross-site request forgery and clickjacking. Its user authentication system provides a secure way to manage user accounts and passwords.

DURGASOFT # 202, 2nd Floor, HUDA Maintenance

```
------------
   SQL Injection Attack

      select * from users where name='xxx' and pwd='yyy'
        name=durga
        pwd=anushka


      name=durga'--
      pwd=sunny
      select * from users where name='durga'--' and pwd='sunny'

      --    comment in sql
```

Certainly!

Cross-Site Scripting (XSS)

What It Is: XSS allows attackers to inject malicious scripts into web pages viewed by other users.

Example: An attacker posts a comment on a blog containing this script: `<script>alert('Hacked!');</script>`. When other users view the comment, the script runs and shows an alert box with "Hacked!".

How to Prevent: Validate and escape user inputs to prevent script injection.

Cross-Site Request Forgery (CSRF)

What It Is: CSRF tricks a logged-in user into making an unwanted request to a web application on which they're authenticated.

Example: An attacker creates a hidden form on a malicious page that automatically transfers money from the victim's bank account to the attacker's account when the victim visits the page.

How to Prevent: Use CSRF tokens to verify that requests come from legitimate sources and set `SameSite` attributes on cookies.

Both vulnerabilities exploit trust in a web application but in different ways—XSS exploits the trust a user has in a site, while CSRF exploits the trust a site has in the user's browser.

Example:we want to send data account details to Bank but details goes to others.

Functionality, Scalability, and Versatility in Django

Functionality:

- **Batteries-Included**: Django comes with a wide range of built-in features, such as an ORM, authentication system, admin interface, and form handling. This allows developers to quickly build robust applications without needing to integrate many third-party tools.

Example: When creating a blog, you get user authentication, an admin interface to manage posts, and built-in database handling out-of-the-box.

Scalability:

- **Modular Components**: Django's architecture allows you to scale specific parts of your application independently. You can separate the database, caching, web server, and application logic to handle increased load efficiently.
- **Middleware**: Django's middleware framework allows for easy integration of additional functionalities, such as caching, security, and user sessions, which helps in managing high traffic.

Example: An e-commerce site can handle thousands of concurrent users by distributing the load across multiple servers, using caching for faster page loads, and optimizing database queries through Django's ORM.

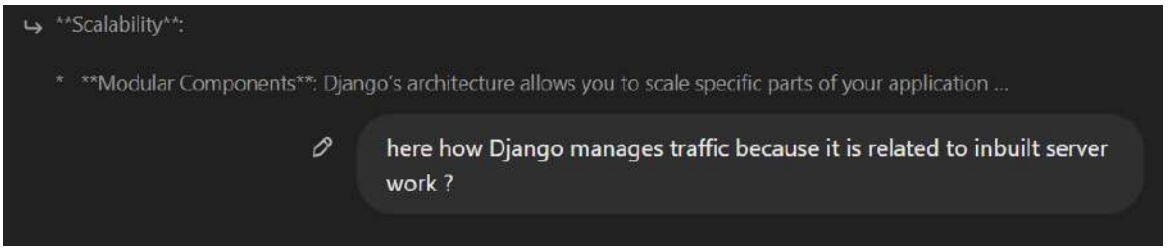
Versatility:

- **Multi-Purpose Framework**: Django is suitable for a wide variety of applications, from content management systems and social networks to scientific computing platforms and API backends.
- **Flexible Template System**: The template system allows you to design complex user interfaces, while Django's ORM and form handling can be used to manage and manipulate data effectively.

Example: Django can be used to build a social media platform with real-time chat, a personal blog with a beautiful interface, or a RESTful API for a mobile application.

Summary

Django's built-in functionality reduces development time, its architecture supports scalability by efficiently managing increased loads, and its versatility makes it suitable for a wide range of web applications, all contributing to its strength as a web framework.



Django itself is not responsible for directly managing high traffic, but its architecture and features facilitate scalability and efficient traffic management when combined with external tools and best practices.

How Django Manages Traffic

1. **Modular Components**:
    - **Separation of Concerns**: Django's architecture allows you to separate different parts of your application, such as the database, caching, web server, and application logic, enabling each component to be scaled independently.
    - **Horizontal Scaling**: You can run multiple instances of your Django application across different servers, distributing the load and improving performance.
2. **Middleware**:
    - **Caching**: Django's middleware can be configured to cache responses at various levels, reducing the load on the database and speeding up response times.
    - **Security**: Middleware can handle security tasks like preventing DDoS attacks, ensuring that only legitimate traffic reaches your application.

- **Session Management**: Middleware efficiently manages user sessions, helping to maintain performance even with many concurrent users.

Tools and Strategies for Handling High Traffic

1. **Web Servers**:
   - **Nginx/Apache**: Use these as reverse proxies to handle incoming requests and distribute them to your Django application servers.
   - **Load Balancers**: Distribute incoming traffic across multiple instances of your Django application, ensuring no single server is overwhelmed.
2. **Database Optimization**:
   - **Read Replicas**: Use database read replicas to distribute read queries, reducing the load on the primary database.
   - **Query Optimization**: Optimize database queries and use Django's ORM to minimize unnecessary database hits.
3. **Caching**:
   - **Memcached/Redis**: Use these caching systems to store frequently accessed data in memory, reducing the need to repeatedly query the database.
   - **Content Delivery Networks (CDNs)**: Cache static files (images, CSS, JavaScript) at edge locations closer to the user, reducing server load and improving response times.
4. **Asynchronous Processing**:
   - **Celery**: Use Celery with Django for handling background tasks and asynchronous processing, ensuring that long-running tasks do not block the main application.

Example

For an e-commerce site:

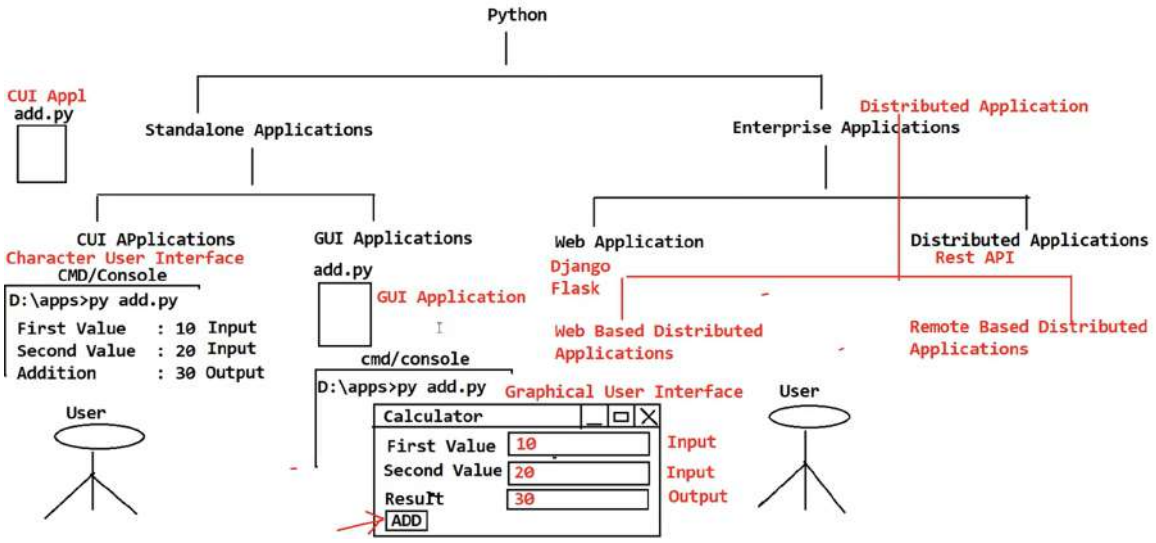
- **Web Server**: Use Nginx to handle static files and route dynamic requests to Django application servers.
- **Load Balancer**: Implement a load balancer to distribute requests across multiple instances of the Django application.
- **Caching**: Use Redis to cache product details and user sessions.
- **Database**: Optimize the database by creating indexes and using read replicas for scaling reads.

Khushal Savalakha

- **Background Tasks**: Use Celery for sending order confirmation emails and processing payments asynchronously.

By using these tools and strategies, Django applications can efficiently manage high traffic and scale to meet increased demand.

Django
was invented to meet fast-moving
newsroom deadlines
, while satisfying the tough requirements of
experienced web developers.




```
By Using Python Programming language, we are able to prepare the following two types of applications.
1. Standalone Applications
2. Enterprise Applications / Distributed Applications

1. Standalone Applications:
--> If we design and execute any application with out using client-server arch or with out distributing
application logic over multiple machines or by providing the complete application logic in single machine then
that application is called as 'Standalone' Application.
EX: Calculator

Theer are two types of Standalone Applications.
1. CUI Applications
2. GUI Applications
```

```
2. GUI Applications

Q)What is the difference between CUI Application and GUI Application?
-------------------------------------------------------------------
Ans:
----
CUI[Character User Interface] Application is a Python application, it will be designed in such a way to get
input data from users through Command prompt / Cosole and to provide output data to the users through the same
Command Prompt / Console , here Command prompt or console is acting as an user interface and it is supporting
Character data.

To Prepare CUI applications we are able to use fundamental python libraries like sys, numpy, random, math, os,
pandas,.......

GUI[Graphical User Interface] Application is a Python application, it will be designed in such a way to get
input data from users through the collection of GUI components and it will provide output data to the users
through the same Collection of GUI components, where the Collection of GUI components are acting as an 'User
Interface' and which is supporting Graphics data[Data which is passed throughthe Graphics components like Text
Fields, Check boxes, Text Areas,.... ]



2. Enterprise Applications / Distributed Applications
--> If we design and execute any application on the basis of Client-Server arch or by distributing application
logic on multiple machines then that application is called as 'Enterprise Application / Distributed
Application'.
```

ex:tKinter,matplotlib,PyQT5

1. **Standalone Applications**

- **CUI Applications (Character User Interface)**
  - **Example**: `add.py` script run in CMD/Console.
  - **Description**: User interacts via command line by providing input values and receiving the output in the console.
- **GUI Applications (Graphical User Interface)**
  - **Example**: `add.py` script run in a GUI environment.
  - **Description**: User interacts via graphical elements like buttons and text fields to input data and receive output.

2. Enterprise Applications

- **Web Application**
  - **Frameworks**: Django, Flask
  - **Description**: Web-based applications accessible through web browsers. They can range from simple websites to complex web-based systems.
- **Distributed Applications**
  - **Rest API**
  - **Description**: Applications that can be distributed across multiple servers or locations, providing APIs for interaction.
  - **Remote Based Distributed Applications**

Khushal Savalakha

- **Description**: Distributed applications accessible remotely over the internet, often through web-based interfaces or APIs.

Chart to Understand the Categorization

Here's a textual chart representing the categorization:

```
Python
├── Standalone Applications
│   ├── CUI Applications
│   │   └── Example: add.py
│   │       ├── Input via CMD/Console
│   │       └── Output in CMD/Console
│   └── GUI Applications
│       └── Example: add.py
│           ├── Input via Graphical Interface
│           └── Output in Graphical Interface
└── Enterprise Applications
    ├── Web Application
    │   ├── Django
    │   └── Flask
    └── Distributed Applications
        ├── Rest API
        └── Remote Based Distributed Applications
```

Summary

- **Standalone Applications**:
  - **CUI Applications**: Command-line based applications where users interact through the console.
  - **GUI Applications**: Applications with a graphical interface where users interact through visual elements.
- **Enterprise Applications**:
  - **Web Applications**: Built using frameworks like Django and Flask, accessible via web browsers.
  - **Distributed Applications**: Include REST APIs and remote-based systems, allowing for scalable and distributed computing.

Khushal Savalakha

1. Introduction to web development and Django
2. Django and Atom installation and development of first application
3. Django static files and templates
4. Working with Models and Databases
5. Working with Django Forms
6. Working with Django Model based forms
7. Working with Advanced Template features
8. Session management
9. User Authentication and Authorization
10. Class Based Views
11. CRUD OPERATIONS by using CBVs and FBVs
12. Django ORM

13. Working with Advanced Model concepts
14. Django middle ware
15. deployment of web application in Live environment
16. Real time project

DRF

Python
UI Technologies
Django
Django Rest Framework

Django Project vs Django Application:

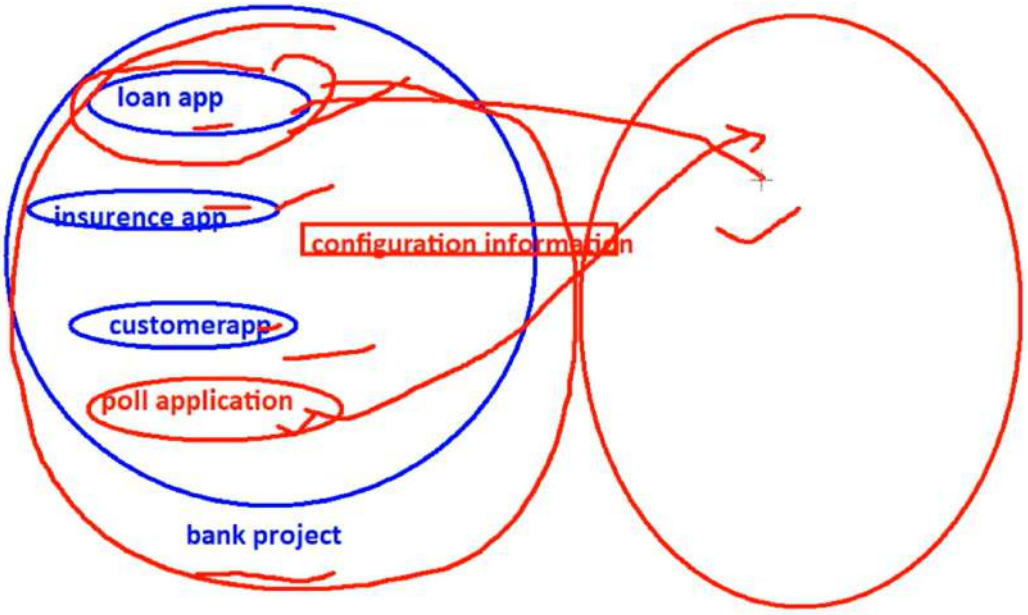
Django project is a collection of application and configurations(which database we use and other things)

- eg:bank project contains several applications like loan applications,insurance app atc.
- without existing django project,no chance of existing django application.

first we have to create project and then we have to create application.

django project =django application+configuration information

Khushal Savalakha



How to create django project:

```
PS D:\Django\Practice> django-admin

Type 'django-admin help <subcommand>' for help on a speci

Available subcommands:

[django]
    check
    compilemessages
    createcachetable
    dbshell
    diffsettings
    dumpdata
    flush
    inspectdb
    loaddata
    makemessages
    makemigrations
    migrate
    optimizemigration
    runserver
    sendtestemail
    shell
```

```
    showmigrations
    sqlflush
    sqlmigrate
    sqlsequencereset
    squashmigrations
    startapp
    startproject
    test
    testserver
Note that only Django core commands are listed as setting
PS D:\Django\Practice> django-admin startproject firstpro
PS D:\Django\Practice> tree /f
Folder PATH listing for volume New Volume
Volume serial number is F491-4E3E
D:.
|   Temp.py
|
└───firstproject
    |   manage.py
    |
    └───firstproject
            asgi.py
            settings.py
            urls.py
            wsgi.py
            __init__.py

PS D:\Django\Practice>
```

In Python:

- **Package**: A directory with an `__init__.py` file and other Python files. It organizes related code.
- `__init__.py`: Makes the directory a package and can contain code to initialize the package or import certain functions/classes.

In essence, a package groups related modules together, and `__init__.py` marks the directory as a package and may help manage what's available from it.

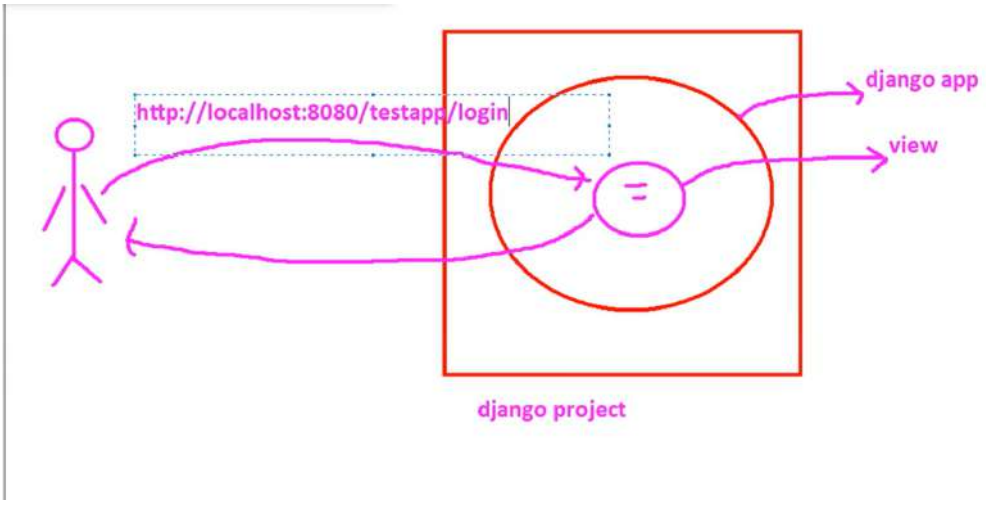
Khushal Savalakha

init.py: It is a blank python script.Because of this special file name, Django treated this folder as python package.

Note: If any folder contains **init**.py file then only that folder is treated as Python package.But this rule is applicable until Python 3.3 Version.

settings.py: In this file we have to specify all our project settings and and configurations like installed applications, middileware configurations, database configurations etc

urls.py: Here we have to store all our url-patterns of our project. For every view (web page), we have to define separate url-pattern in url.py. End user can use url-patterns to access our webpages.



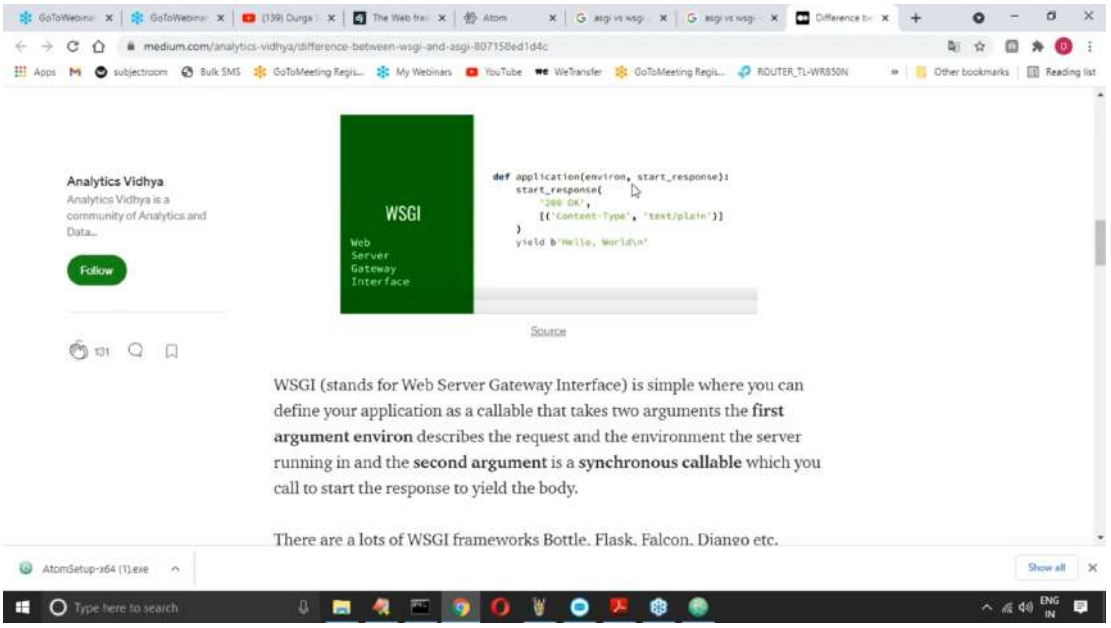
it means by using which url end user can communicate or request a view .

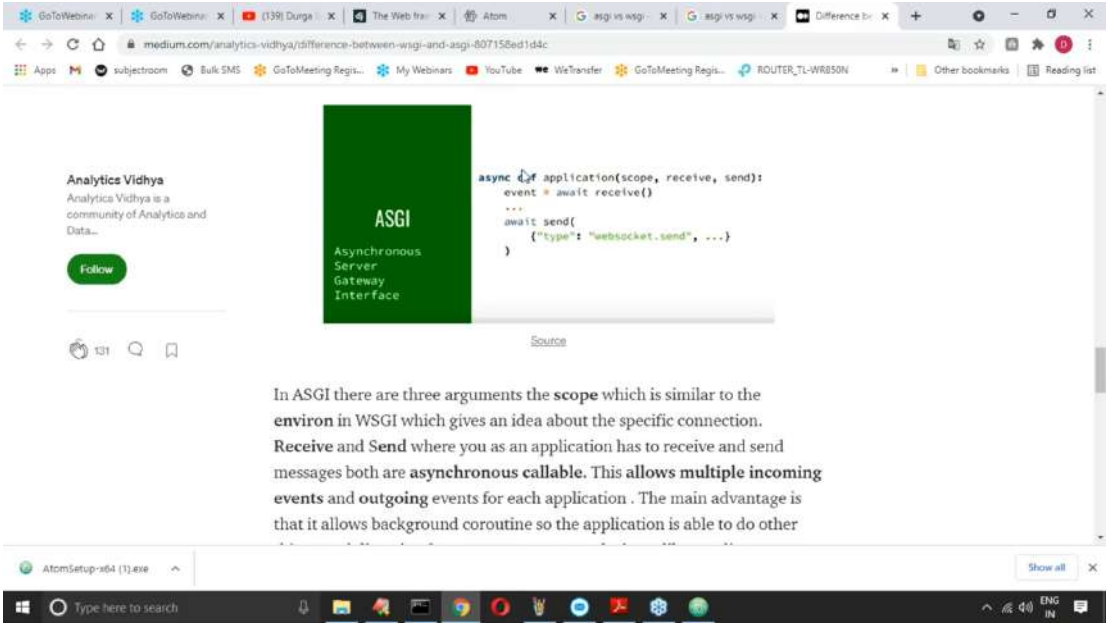
wsgi.py: wsgi Web Server Gateway Interface. We can use this file while deploying our application in production on online server.

manage.py It is a command line utility to interact with Django project in various ways like to run development server, run tests, create migrations etc.(most used)

- to start django server.
- migrations

asgi.py: asynchronous server gateway interface.





settings.py:

database configurations,applications,etc

```
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
```

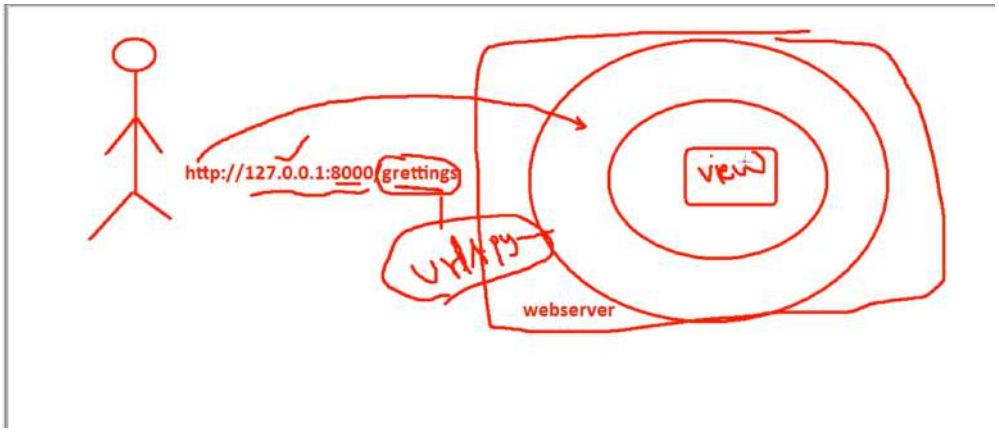
```
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'test.py'-->our application
]
```

urls.py:

- Define url's required for your application.

Role of webserver:

- webserver is responsible to provide environment to run our web applications. (project has multiple applications)
- Webserver is responsible to receive the request and forward that request to the corresponding web component(view) based on url-pattern and provide required response to the end user.



How to run django development server:

django itself is responsible to provide webserver.we can also add(configure) our server like APACHI,IISR etc.

Three most important files:

manage.py(to start django server.)

settings.py(database configurations,applications,etc)

urls.py(url's required for your application.)

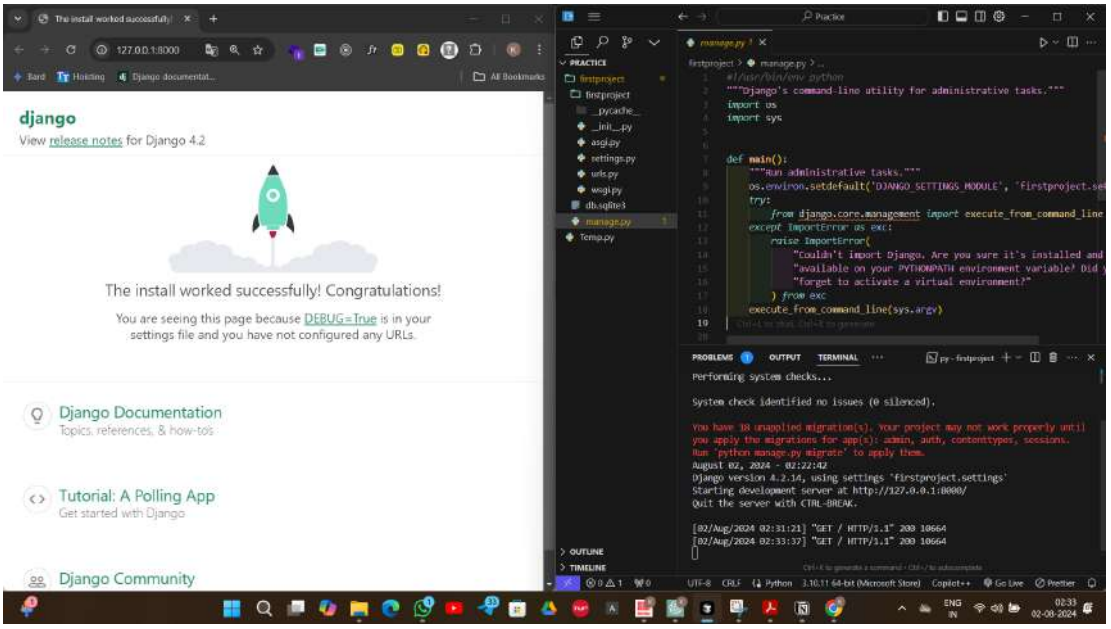
py test.py A B

py manage.py runserver

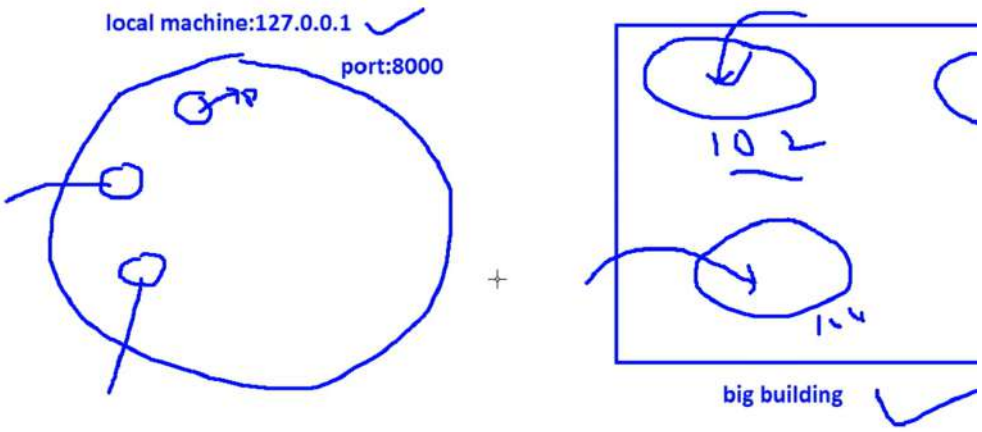
Khushal Savalakha

```
PS D:\Django\Practice> cd .\firstproject
PS D:\Django\Practice\firstproject> py manage.py runserve
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not
Run 'python manage.py migrate' to apply them.
August 02, 2024 - 02:22:42
Django version 4.2.14, using settings 'firstproject.setti
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

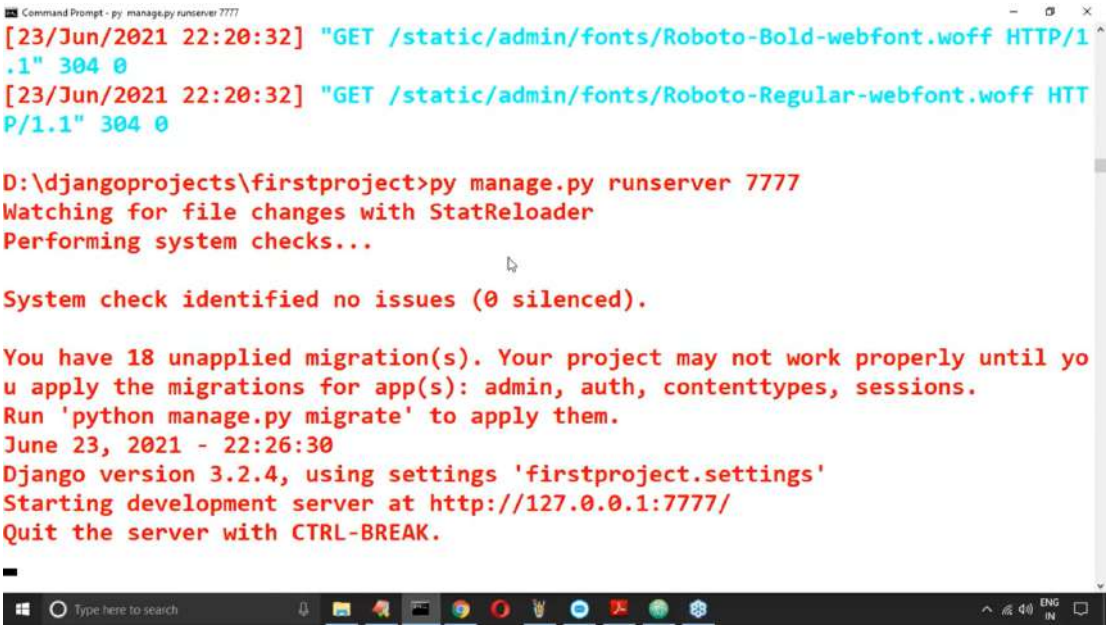
here http:***//127.0.0.1:8000/*** running on local machine.



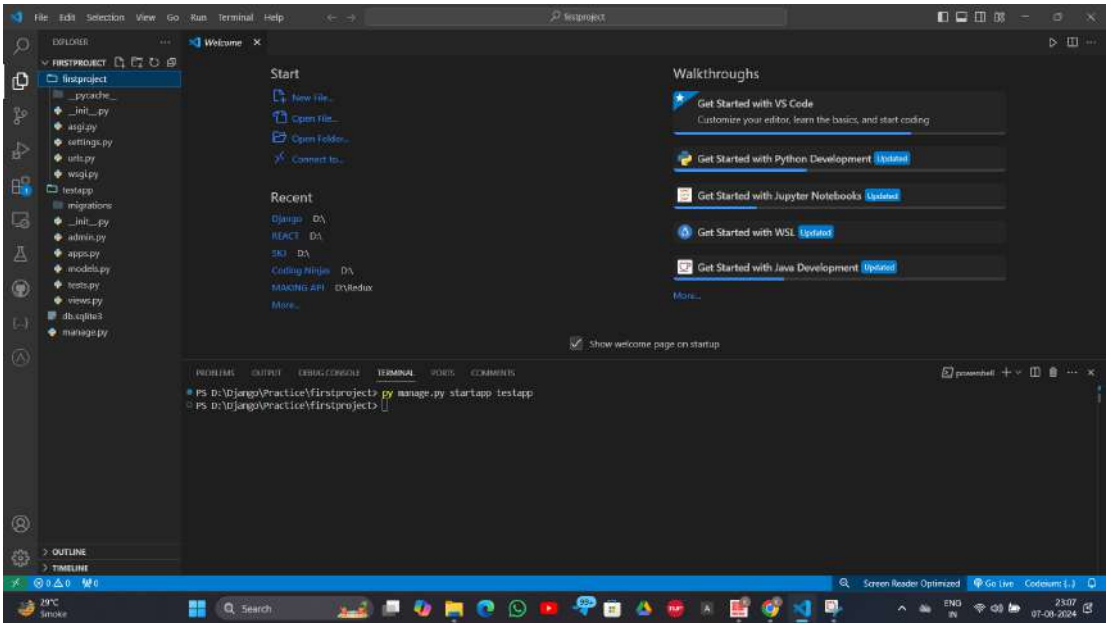
Khushal Savalakha



For every application a separate port is reserved.Here By default port is 8000.



To make new Application

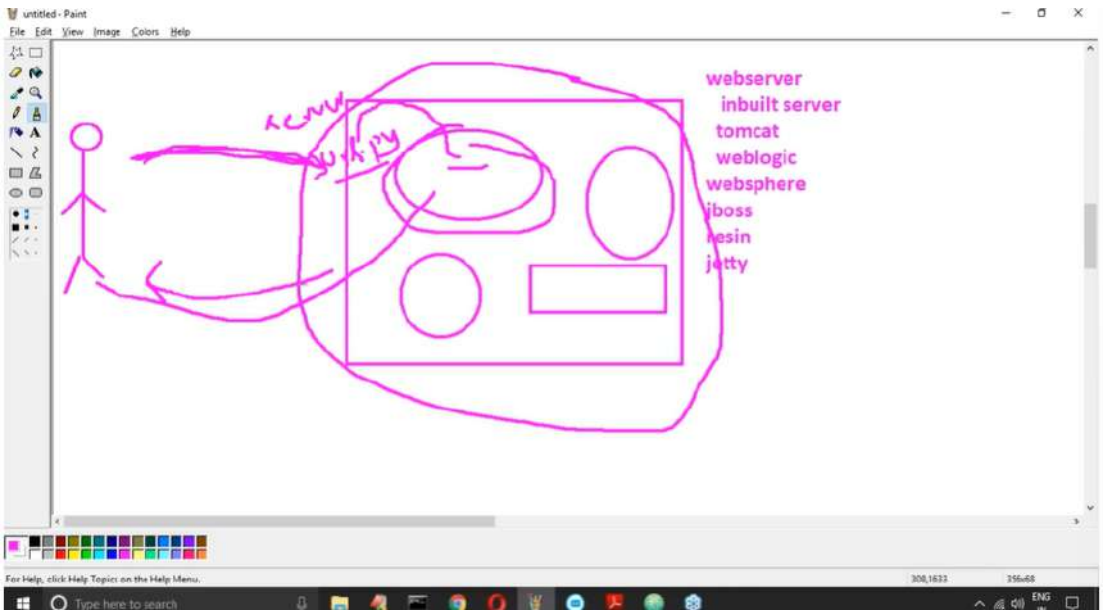



```
PS D:\Django\Practice\firstproject> py manage.py startapp
PS D:\Django\Practice\firstproject> tree
Folder PATH listing for volume New Volume
Volume serial number is F491-4E3E
D:.
├───firstproject
│   └───__pycache__
└───testapp
    └───migrations
PS D:\Django\Practice\firstproject>
```

```
py manage.py startapp testapp


testapp
        |   admin.py
        |   apps.py
        |   models.py
        |   tests.py
        |   views.py
        |   __init__.py
        |
        |---migrations
                __init__.py
```

Settings is not applicable at application level.__init__=python package

```
1. How to start the project:
----------------------------
django-admin startproject firstproject
      settings.py--->project level configuration
      manage.py--->to runserver, to create application etc
      urls.py

2. How to run server:
---------------------
py manage.py runserver

3. How to create application:
-----------------------------
py manage.py startapp testapp
            views.py
```



Yes, that's correct. A web server provides an environment to receive and process requests from clients (such as web browsers). When a client makes a request (typically via HTTP or HTTPS), the web server processes this request, executes the necessary logic (which could involve fetching data, running scripts, etc.), and then returns the appropriate response, usually in the form of a web page or other resource.

This environment includes several components:

1. **Request Handling**: The server listens for incoming requests from clients.
2. **Routing**: It directs the request to the appropriate handler based on the URL and HTTP method.
3. **Execution Environment**: The server may execute scripts, interact with databases, or perform other actions to generate the content for the response.
4. **Response Handling**: Once the content is generated, the server sends it back to the client.

This is a fundamental part of how web applications function, allowing users to interact with dynamic content and services over the web.

```
8  django project and django application
9  project = multiple applications + configuration information
10
11 1. django-admin startproject  firstproject
12
13 firstproject
14     |- manage.py   (to run server, to create applications etc)
15     |- firstproject
16           |- settings.py
17           |- urls.py
18           |- wsgi.py
19           |- asgi.py
20           |- __init__.py
21
22 2. py manage.py runserver
23 default port number: 8000
24
25 py manage.py runserver 7777
```

```
14     |- manage.py   (to run server, to create applications etc)
15     |- firstproject
16           |- settings.py
17           |- urls.py
18           |- wsgi.py
19           |- asgi.py
20           |- __init__.py
21
22 2. py manage.py runserver
23 default port number: 8000
24
25 py manage.py runserver 7777
26
27
28 http://127.0.0.1:8000/
29 http://127.0.0.1:7777/
30
31
```

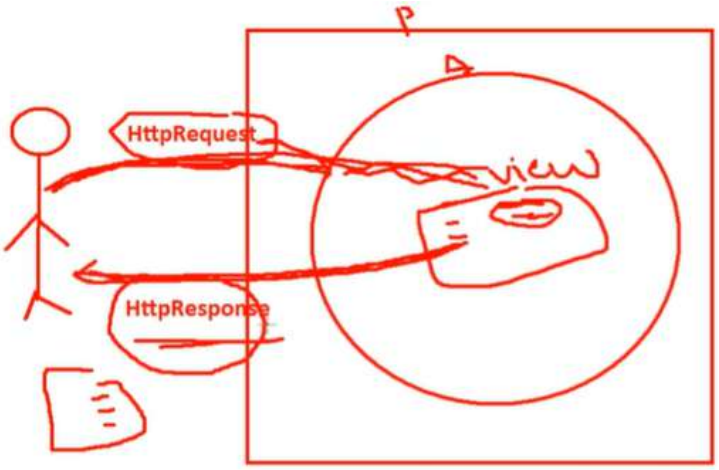
```
42
43
44 testapp
45       |    admin.py--->models concept and admin interface
46       |    apps.py---->application specific configurations
47       |    models.py--->to define model classes
48       |    tests.py-->to write test cases
49       |    views.py--->to write view functions
50       |    __init__.py
51       |
52       |---migrations--->database specific information
53            __init__.py
54
55
56
```

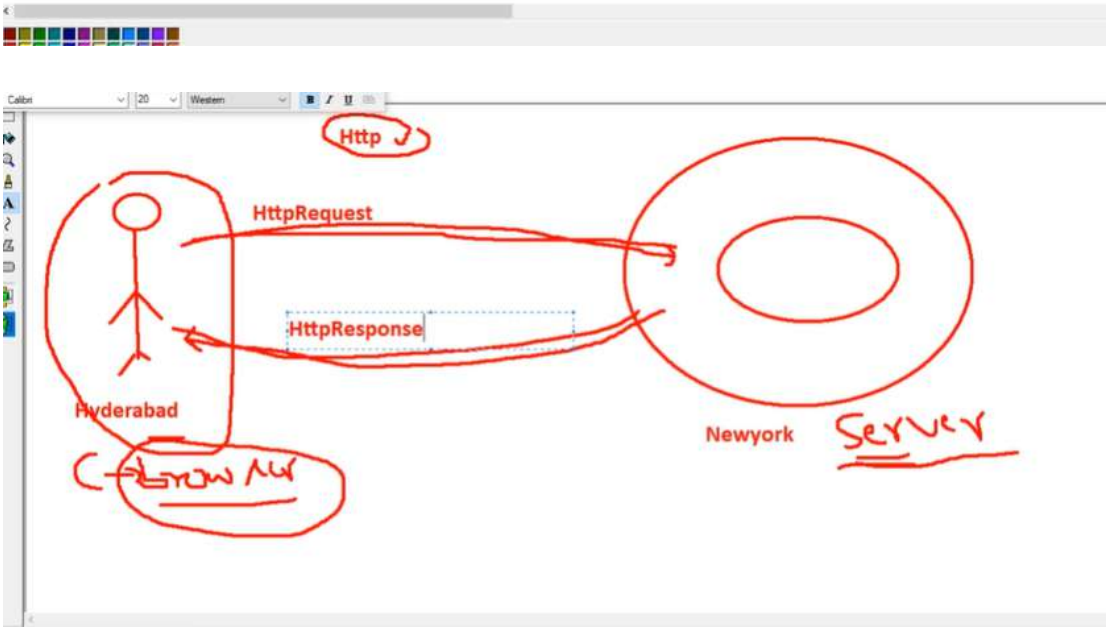

Khushal Savalakha

```
56
57
58
59 django-admin startproject firstproject
60                              settings.py
61                              urls.py
62                              manage.py
63 py manage.py startapp testapp
64                              views.py
65
66
67
68
69
```

```
67 Activities required for application:
68 ------------------------------------
69 1. We have to add our application in the settings.py file
70
71 INSTALLED_APPS = [
72     'django.contrib.admin',
73     'django.contrib.auth',
74     'django.contrib.contenttypes',
75     'django.contrib.sessions',
76     'django.contrib.messages',
77     'django.contrib.staticfiles',
78     'testapp'
79 ]
80 2. Create views for our application to provide required functionality
81
```

```
80 2. Create views for our application to provide required functionality inside
   views.py file, which is present in our application.
81  2 ways to create views
82
83  1. Function Based Views(FBVs)
84  2. Class Based Views(CBVs)
85
86  1. Function Based Views(FBVs)
87 ------------------------------
88 We have to define view function which is responsible to provide required response to
   the end user.
89
90
91
```





here http is common platform as English which can understand both browser and server.

> what is the meaning of randering?

Both explanations in the image and the text I provided refer to different aspects of the concept of rendering. Here's why they might seem slightly different but are still related:

1. **Converting vs. Processing and Generating:**
    - **Converting:** In some contexts, rendering does involve converting one form of data (like 3D models or HTML/CSS) into another form (like a 2D image or a

webpage). This is especially true in graphics, where the data for a 3D model is converted into a visual image.

- **Processing and Generating:** Rendering is also about the process that involves more than just conversion. It includes interpreting the data, performing calculations, applying textures, and other steps to generate the final visual output.

2. **Different Emphasis:**
   - The text on the left in your image emphasizes rendering as converting data into a visual or usable format. This is accurate but more focused on the end result of rendering.
   - The text on the right (which I provided) expands on the concept by explaining that rendering is a broader process. It involves not just converting but also generating the final output from raw data, which includes various intermediate steps like processing and interpreting data.

Summary:

- **Rendering as Converting:** Focuses on the final step where data is turned into something visual or usable.
- **Rendering as Processing and Generating:** Encompasses the entire process, including interpreting data, performing necessary calculations, and finally generating the output.

Both interpretations are correct, but they highlight different aspects of the rendering process.

(conversion)

1]Created a project

2]Created an application

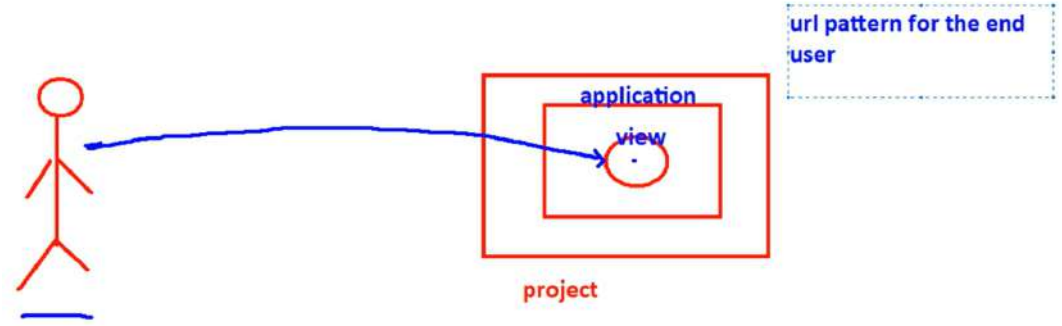
3]add that application inside settings.py

4]define view function inside views.py

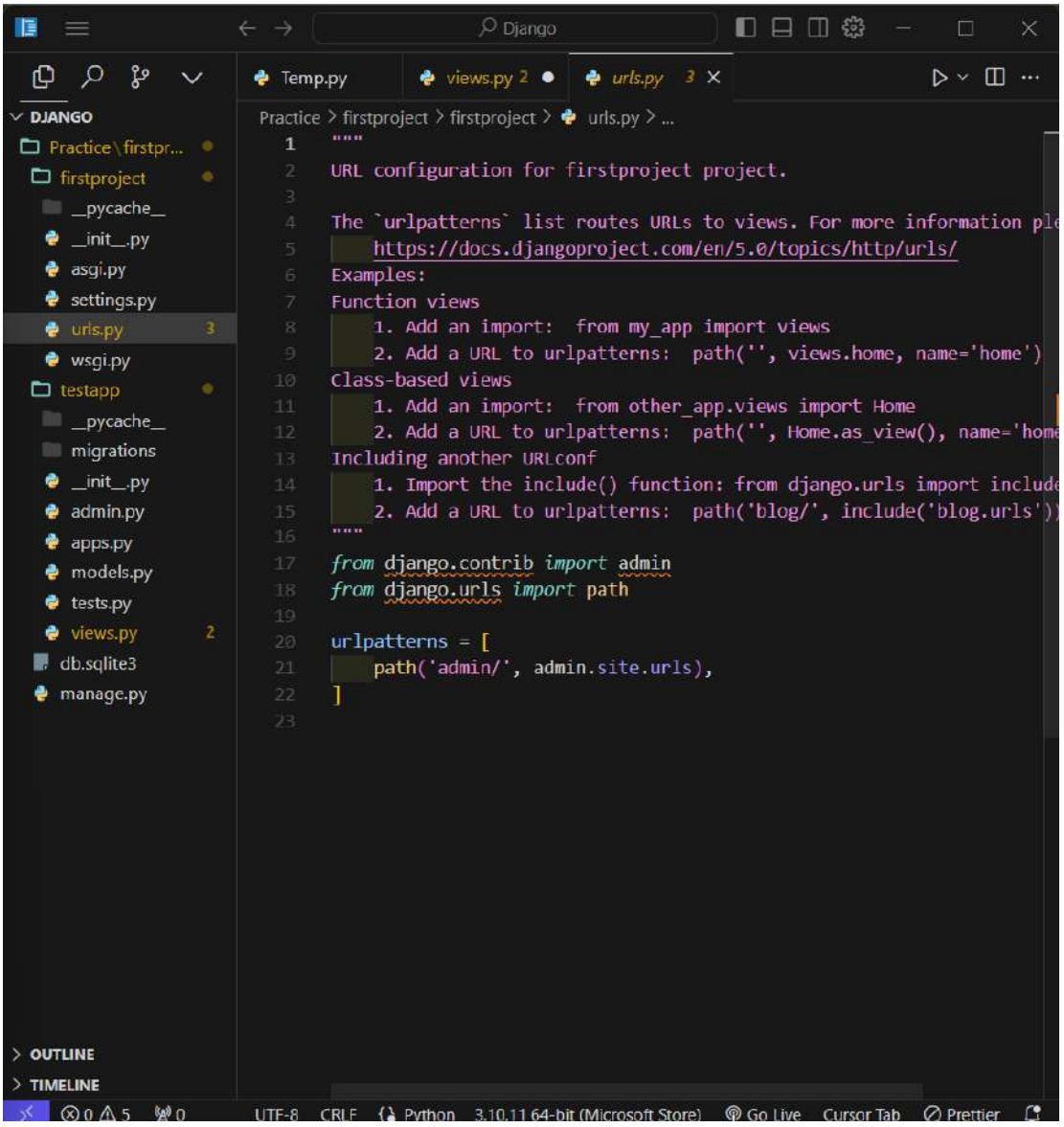
```
from django.shortcuts import render
from django.http import HttpResponse

# Create your views here.
def display(request):
```


Khushal Savalakha

```
    s='<h1>Welcome to DURGA django classes ...</h1>'
    return HttpResponse(s)
```



url-pattern:hello



In Django, the `urls.py` file shown in your image is at the project level, meaning it handles the URL routing for the entire Django project. It typically includes references to other apps' URLs or directly maps URLs to views within the project.

`views.py` in the Context of Django Projects

The `views.py` file, however, usually exists at the **app level** rather than the project level. Each Django app typically has its own `views.py` file, which contains the logic for handling requests and returning responses. Here's a breakdown:

1. **Project-Level `urls.py`:**
    - This file is responsible for routing URLs to the appropriate views across the entire project.

Khushal Savalakha

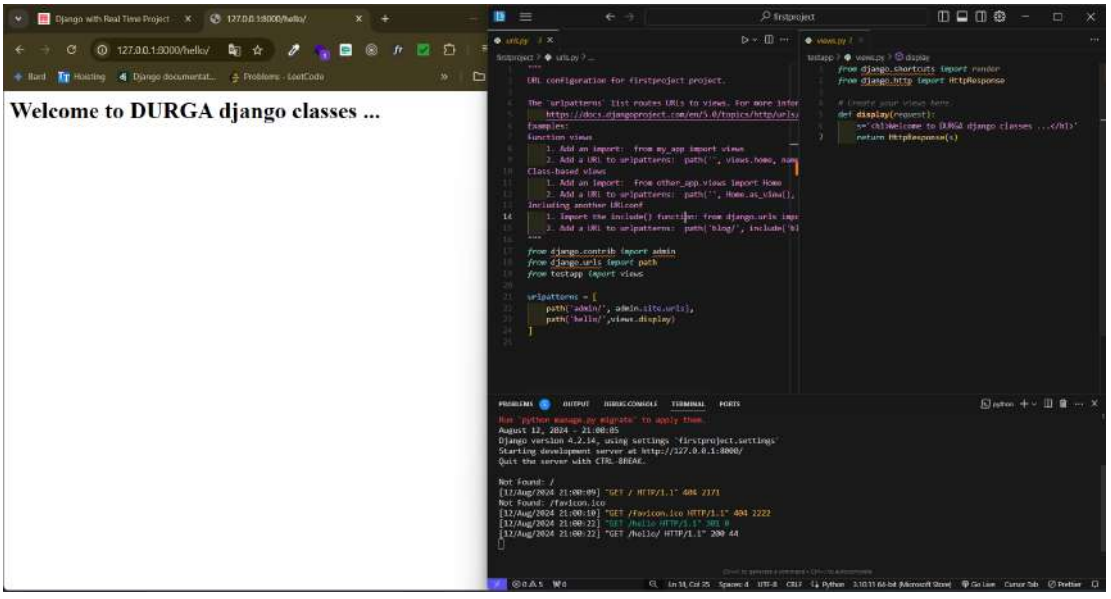
- It can include URLs from multiple apps using the `include()` function or directly reference specific views.

2. **App-Level `views.py`:**
   - The `views.py` file in a Django app contains the actual view functions or class-based views that process incoming HTTP requests and return responses, such as rendering a template or returning JSON data.
   - These views are mapped to URLs in the app's `urls.py` (if it exists) or directly in the project's `urls.py`.

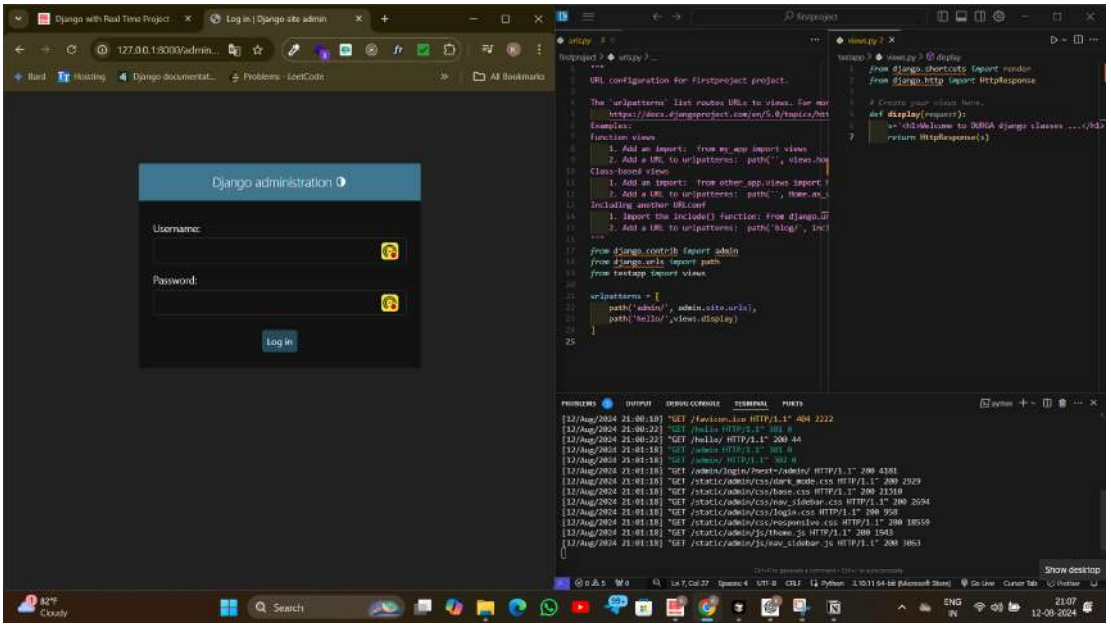
Summary

- `urls.py`: Project-level, maps URLs to views across the project, may reference app-specific URLs.
- `views.py`: App-level, contains the logic that is triggered when certain URLs are accessed.

If your project doesn't have an app-specific `urls.py`, the views from `views.py` would be mapped directly in the project-level `urls.py`.



Khushal Savalakha




```
Note:
1. Each view should take atleast one argument: request
2. Each view should return HttpResponse object with our required response


3. Define url-pattern for our view function inside urls.py:
-------------------------------------------------------
from django.contrib import admin
from django.urls import path
from testapp import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('hello/', views.display),
]

```

in old versions,instead of path() function we have to use url() function.

```
add application inside settings.py
define view function inside views.py
define url-pattern for view inside urls.py file
runserver
send request

http://127.0.0.1:8000/hello   =====>web server will get request===>open urls.py and
identity the corresponding view function===>execute that view function and return
HttpResponse to the end user
```

```
1. Creation of django project
django-admin startproject firstproject

2. Creation of django application
py manage.py startapp testapp

3. Add application to settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'testapp'
]
```

```
4. Define our functionality inside views.py file

views.py
from django.http import HttpResponse
def greetings(request):
     s='<h1>Hello welcome to django sessions</h1>'
     return HttpResponse(s)
```

```
5. Create url-pattern for our view
   This url-pattern meant for end user.
urls.py:
--------
from django.urls import path
from testapp import views
urlpatterns=[
   .....
   path('hello/',views.display)
]

6. start server
py manage.py runserver

7. send the request
http://127.0.0.1:8000/hello
```

💡 in Django framework what is the use of settings.py and how it is working with views.py and urls.py and manage.py explain all steps with example depthly.

In the Django framework, `settings.py`, `views.py`, `urls.py`, and `manage.py` play crucial roles in creating and managing web applications. Let's break down each file and explain how they interact with one another.

1. `settings.py`

Purpose:

- This file contains the configuration settings for your Django project. It includes database configurations, installed apps, middleware, static files settings, templates settings, and more.

Key Components:

- **INSTALLED_APPS**: Lists all the apps that are part of your Django project.
- **MIDDLEWARE**: A list of middleware components that process requests and responses.
- **DATABASES**: Configurations for connecting to databases.
- **STATIC_URL** and **STATICFILES_DIRS**: Settings for handling static files like CSS, JavaScript, and images.
- **TEMPLATES**: Settings for configuring the template engine.

Example:

```
# settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'myapp',  # Your custom app
]
```


Khushal Savalakha

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / "db.sqlite3",
    }
}

STATIC_URL = '/static/'
STATICFILES_DIRS = [BASE_DIR / "static"]
```

2. `views.py`

Purpose:

- This file contains the logic for handling requests and returning responses. Views are Python functions or classes that receive web requests, process them, and return a response, typically an HTML page, JSON, or other data.

How It Works:

- When a user accesses a URL, Django uses the URL dispatcher (`urls.py`) to determine which view should handle the request. The view processes any data, interacts with models, and returns a response.

Example:

```
# views.py
from django.http import HttpResponse

def home(request):
    return HttpResponse("Welcome to the Home Page")

def about(request):
    return HttpResponse("This is the About Page")
```

3. `urls.py`

Purpose:

- This file maps URLs to views. It acts as a bridge between the user's request and the corresponding view. When a user enters a URL, Django uses the `urls.py` file to determine which view to call.

Khushal Savalakha

How It Works:

- The URL patterns are defined using regular expressions or path converters that match the requested URL and route it to the appropriate view function.

Example:

```
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('', views.home, name='home'),
    path('about/', views.about, name='about'),
]
```

4. `manage.py`

Purpose:

- This is a command-line utility that allows you to interact with your Django project. You can use it to start the development server, make migrations, run tests, and more.

How It Works:

- `manage.py` acts as a wrapper around Django's command-line utility, providing commands to interact with your project. It uses settings from `settings.py` to know how to configure the environment.

Example:
To start the development server, you would run:

```
python manage.py runserver
```

This command starts the server using the configurations defined in `settings.py`.

Interaction Between Files

1. **User Requests a URL**: When a user types a URL in the browser and hits enter, the request is sent to the Django server.
2. **URL Matching**: Django's URL dispatcher (`urls.py`) matches the requested URL with the URL patterns defined in `urls.py`. If a match is found, it routes the

Khushal Savalakha

request to the corresponding view.

3. **View Handling**: The view function (in `views.py`) associated with the matched URL processes the request. It may query the database, use templates, and more, depending on the logic in the view.
4. **Response Generation**: The view returns a response, such as an HTML page, JSON, or another type of content. This response is sent back to the user's browser.
5. **Settings Configuration**: Throughout the process, `settings.py` provides configurations such as database connections, installed apps, and static file handling.
6. **manage.py Usage**: While developing, you use `manage.py` to run the server, create migrations, or perform other tasks related to the Django project.

Example in Action

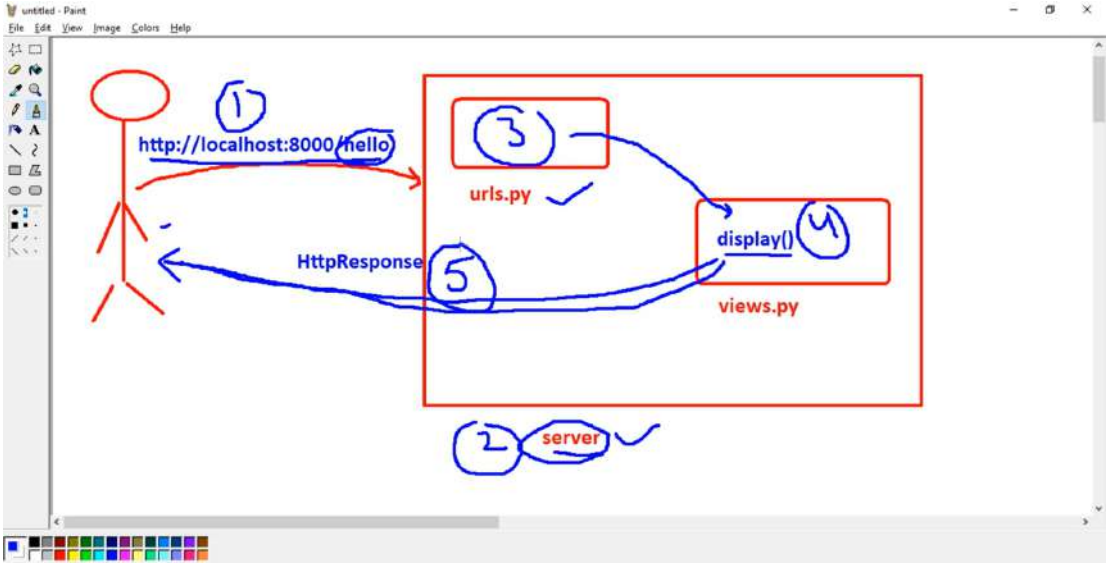
Imagine you have a simple Django project with a home page and an about page.

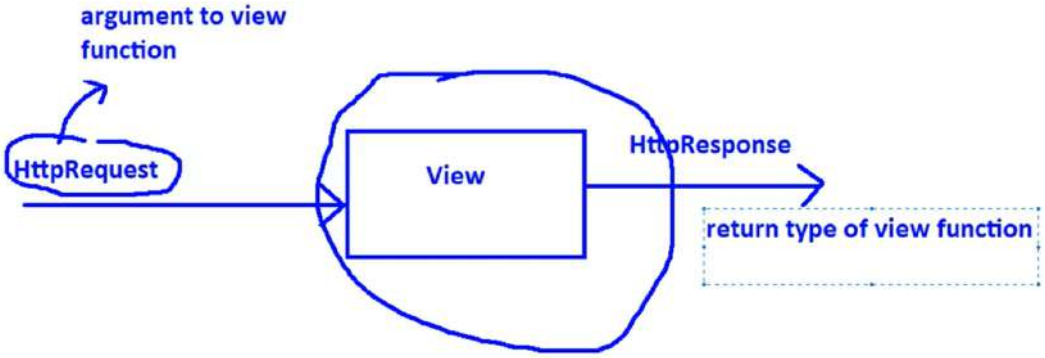
- **User enters `http://localhost:8000/`**: Django uses `urls.py` to find the matching pattern, which routes to the `home` view in `views.py`.
- **`home` view**: This view function returns an HTTP response with the text "Welcome to the Home Page."
- **Response sent back to the browser**: The user sees "Welcome to the Home Page" on their screen.

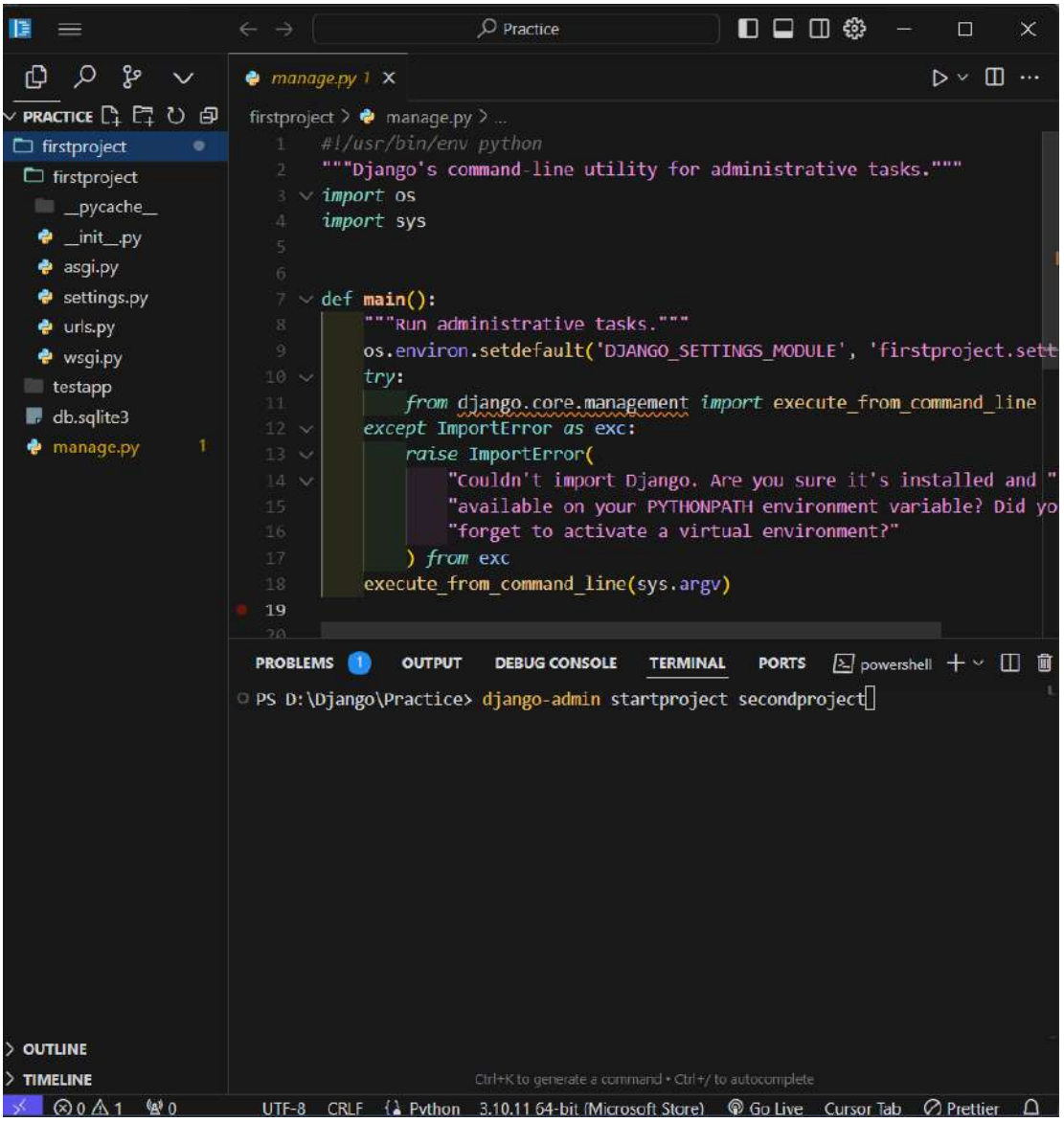
If the user then navigates to `http://localhost:8000/about/`, Django repeats the process, but this time routes to the `about` view, which returns "This is the About Page."

This is how `settings.py`, `views.py`, `urls.py`, and `manage.py` work together in a Django project.

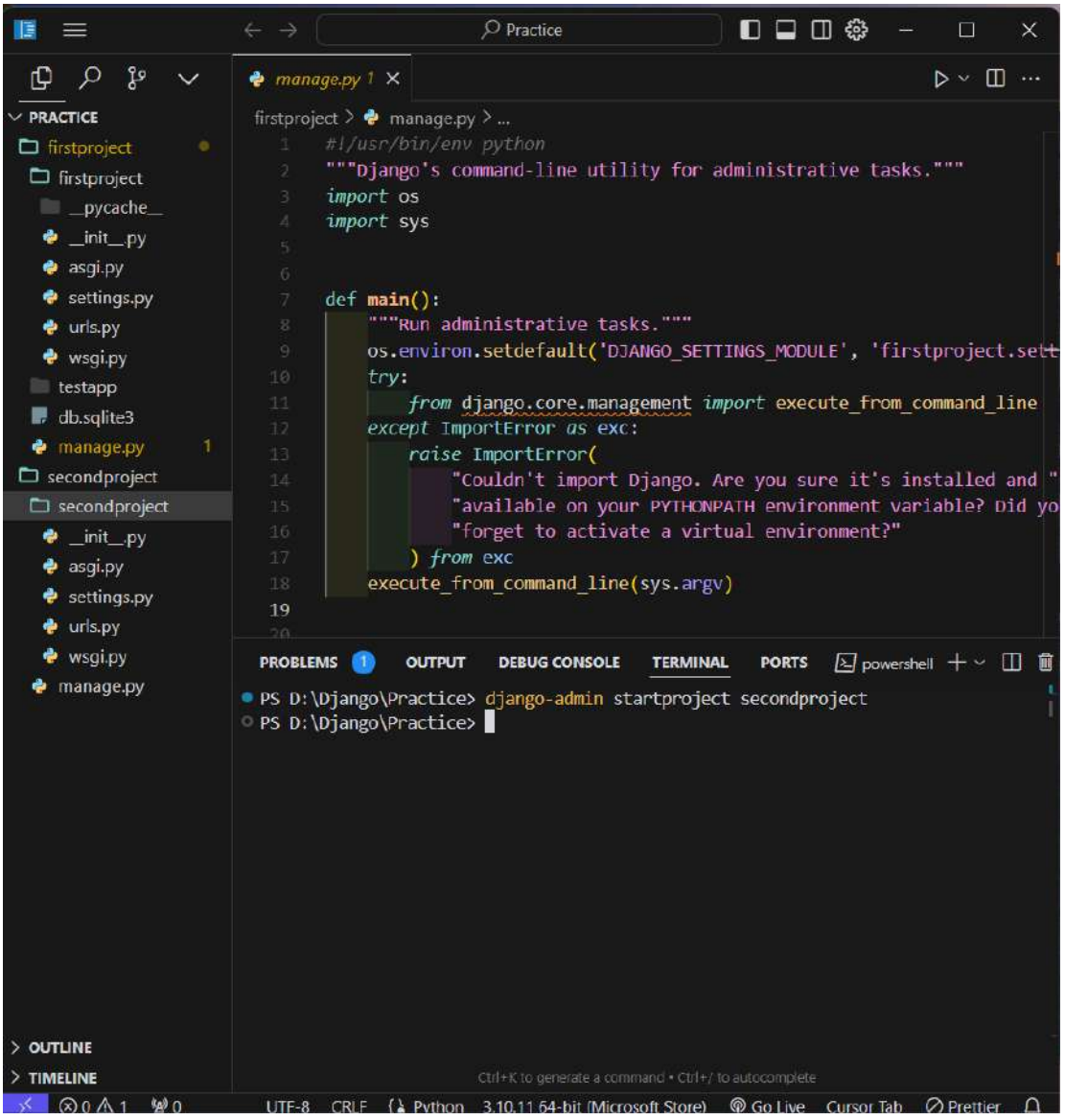
Khushal Savalakha

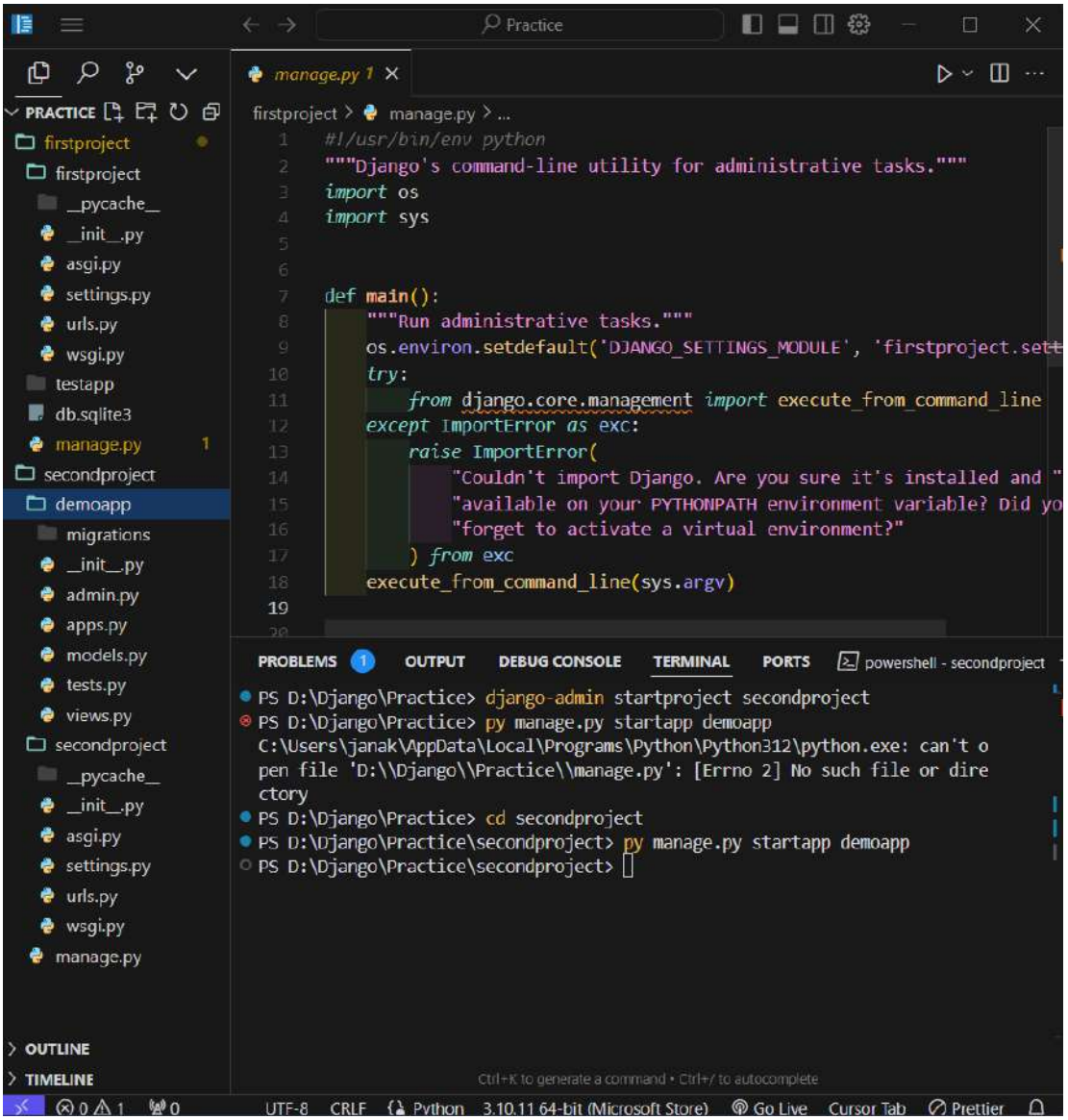

untitled - Paint
File Edit View Image Colors Help
1
http://localhost:8000/hello
3
urls.py
display()
4
HttpResponse
5
views.py
2
server


argument to view function
HttpRequest
View
HttpResponse
return type of view function



Khushal Savalakha

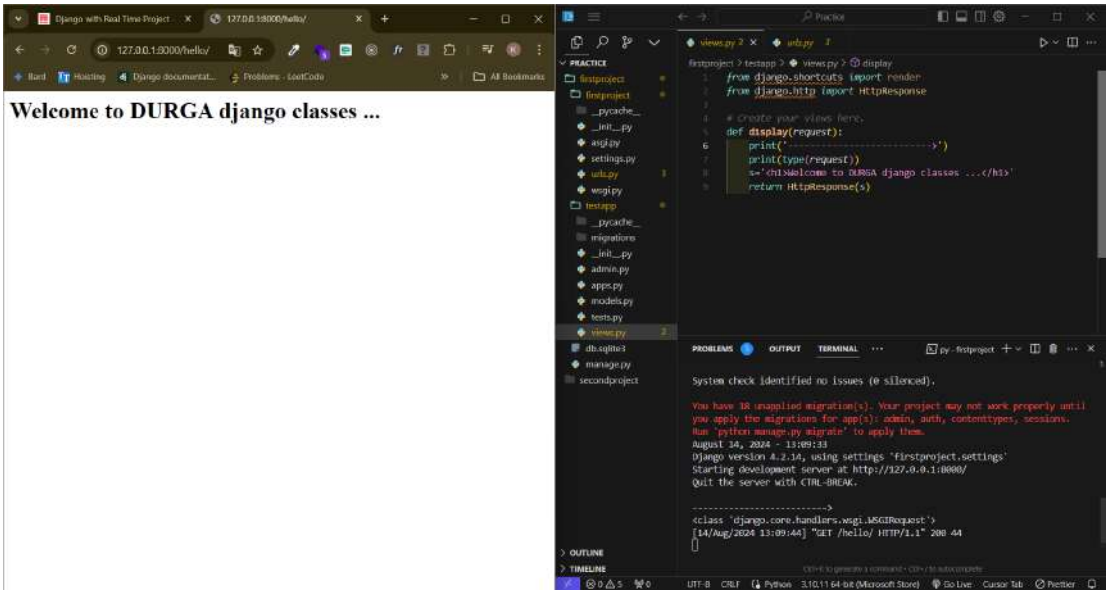

Practice
manage.py 1
firstproject > manage.py > ...
PRACTICE
firstproject
firstproject
__pycache__
__init__.py
asgi.py
settings.py
urls.py
wsgi.py
testapp
db.sqlite3
manage.py 1
secondproject
secondproject
__init__.py
asgi.py
settings.py
urls.py
wsgi.py
manage.py
#!/usr/bin/env python
"""Django's command-line utility for administrative tasks."""
import os
import sys
def main():
"""Run administrative tasks."""
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'firstproject.sett
try:
from django.core.management import execute_from_command_line
except ImportError as exc:
raise ImportError(
"Couldn't import Django. Are you sure it's installed and "
"available on your PYTHONPATH environment variable? Did yo
"forget to activate a virtual environment?"
) from exc
execute_from_command_line(sys.argv)
PROBLEMS 1
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
powershell
PS D:\Django\Practice> django-admin startproject secondproject
PS D:\Django\Practice>
OUTLINE
TIMELINE
Ctrl+K to generate a command • Ctrl+/ to autocomplete
UTF-8
CRLF
Python 3.10.11 64-bit (Microsoft Store)
Go Live
Cursor Tab
Prettier



Develop web application which will provide current server time?

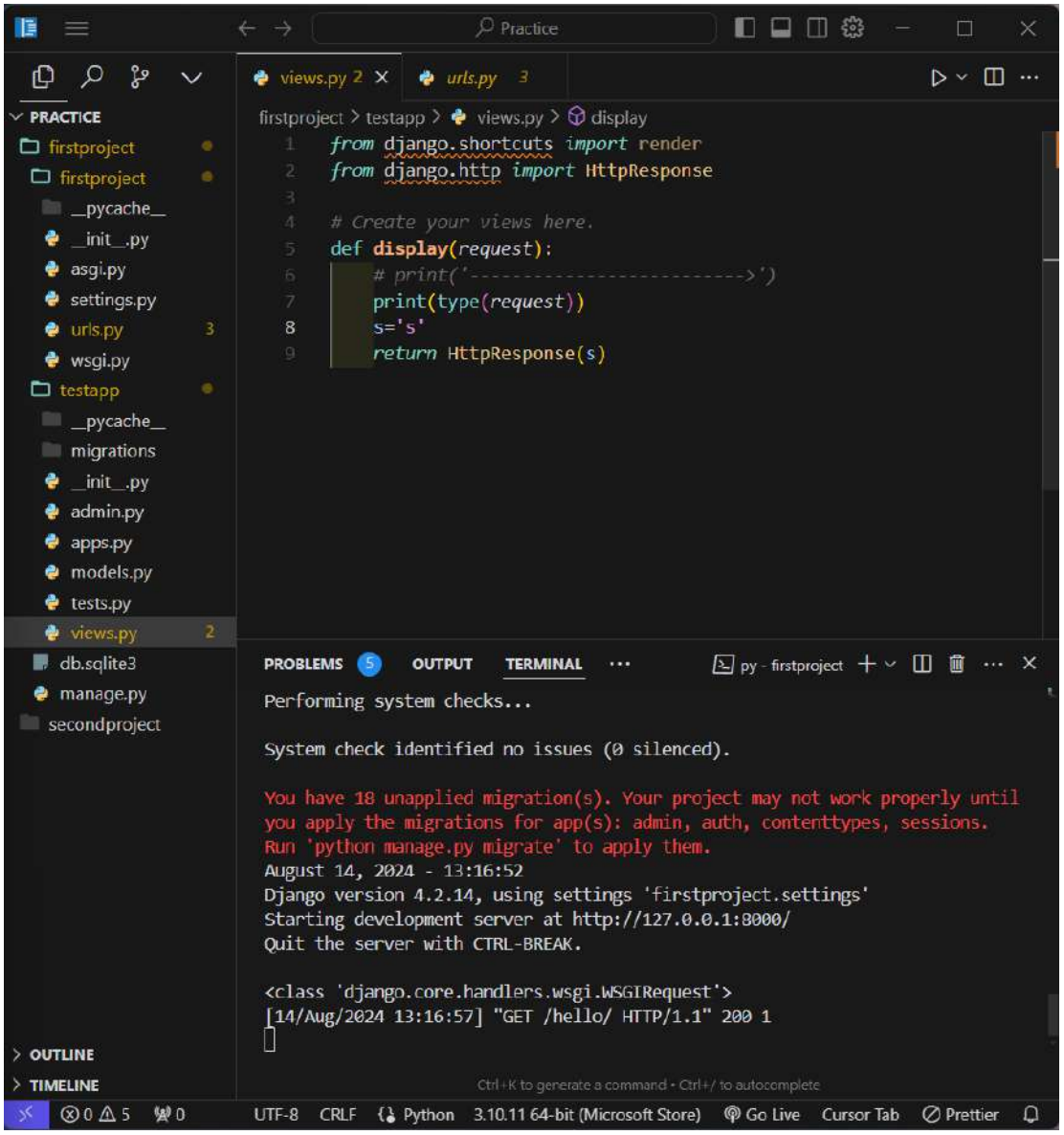
1. Creation of django project
django-admin startproject secondproject

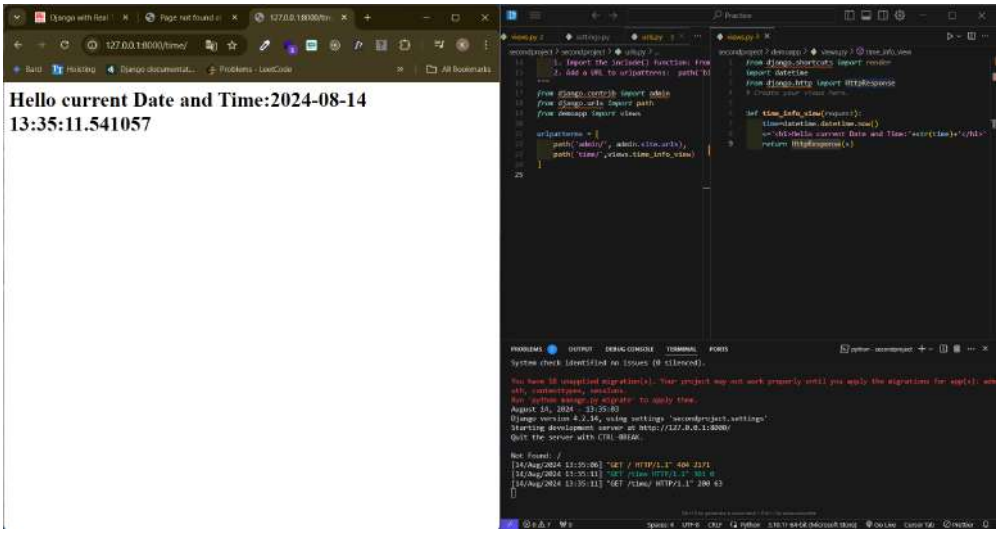
2. Creation of django application
py manage.py startapp demoapp



here we can’t see data like MERN stack in inspect console.

200 means success response and 89 is content length.This is length of HttpResponse(s).


Practice
views.py 2
urls.py 3
firstproject > testapp > views.py > display
from django.shortcuts import render
from django.http import HttpResponse
# Create your views here.
def display(request):
# print('------------------------->')
print(type(request))
s='s'
return HttpResponse(s)
PRACTICE
firstproject
__pycache__
__init__.py
asgi.py
settings.py
urls.py 3
wsgi.py
testapp
__pycache__
migrations
__init__.py
admin.py
apps.py
models.py
tests.py
views.py 2
db.sqlite3
manage.py
secondproject
PROBLEMS 5
OUTPUT
TERMINAL
py - firstproject
Performing system checks...
System check identified no issues (0 silenced).
You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
August 14, 2024 - 13:16:52
Django version 4.2.14, using settings 'firstproject.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
<class 'django.core.handlers.wsgi.WSGIRequest'>
[14/Aug/2024 13:16:57] "GET /hello/ HTTP/1.1" 200 1
OUTLINE
TIMELINE
Ctrl+K to generate a command • Ctrl+/ to autocomplete
UTF-8
CRLF
Python
3.10.11 64-bit (Microsoft Store)
Go Live
Cursor Tab
Prettier


Hello current Date and Time:2024-08-14
13:35:11.541057

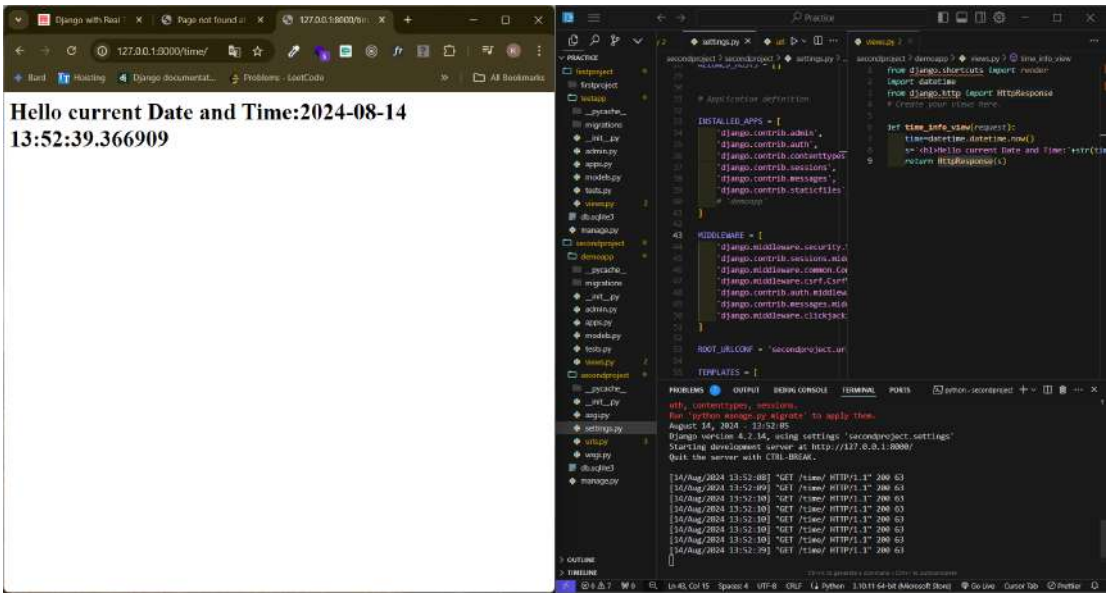
```
Select Command Prompt - py manage.py runserver
    response = get_response(request)
  File "C:\Python38\lib\site-packages\django\core\handlers\base.py", line 181,
 in _get_response
    response = wrapped_callback(request, *callback_args, **callback_kwargs)
TypeError: display() takes 0 positional arguments but 1 was given
[25/Jun/2021 22:33:26] "GET /hello/ HTTP/1.1" 500 58363
```

if we can’t write any parameter in function.

```

3. Add application in the settings.py

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'demoapp'
]

```

```
4. Creation of view function

from django.shortcuts import render
import datetime
from django.http import HttpResponse
# Create your views here.
def time_info_view(request):
    time=datetime.datetime.now()
    s = '<h1> Hello Current Date and Time:'+str(time)+'</h1>'
    return HttpResponse(s)

```

```
5. Define url-pattern inside urls.py
from django.contrib import admin
from django.urls import path
from demoapp import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('time/', views.time_info_view),
]


```



> here i don't mention demoapp aplication in settings.py then why he does not give error while importing or using data of demoapp in second project ?

In Django, an app must be listed in the `INSTALLED_APPS` section of the `settings.py` file for it to be recognized by the project. However, in your case, it seems that even though `demoapp` is not listed in `INSTALLED_APPS`, you are still able to access and use it.

The reason you are not seeing an error is that `INSTALLED_APPS` is primarily used by Django to track apps for certain operations, like:

- Database migrations.
- Admin interface.
- Template loading.
- Static files.

However, for basic functionality like serving a view from `demoapp`, Django does not strictly require the app to be listed in `INSTALLED_APPS`. As long as the view is correctly imported and registered in `urls.py`, Django will serve it.

But keep in mind, if you try to use features that depend on `INSTALLED_APPS` (like models, migrations, or admin), you might encounter issues. To ensure everything works smoothly and follows best practices, you should add `demoapp` to the `INSTALLED_APPS` list in `settings.py`.

> ↪ In Django, an app must be listed in the `INSTALLED_APPS` section of the `settings.py` file for it to be recognized by the project. However, in your case, it seems that even though `demoapp` is not listed in `INSTALLED_APPS`, you are still able to access and use it....
>
> explain simply why i don't get any error ?

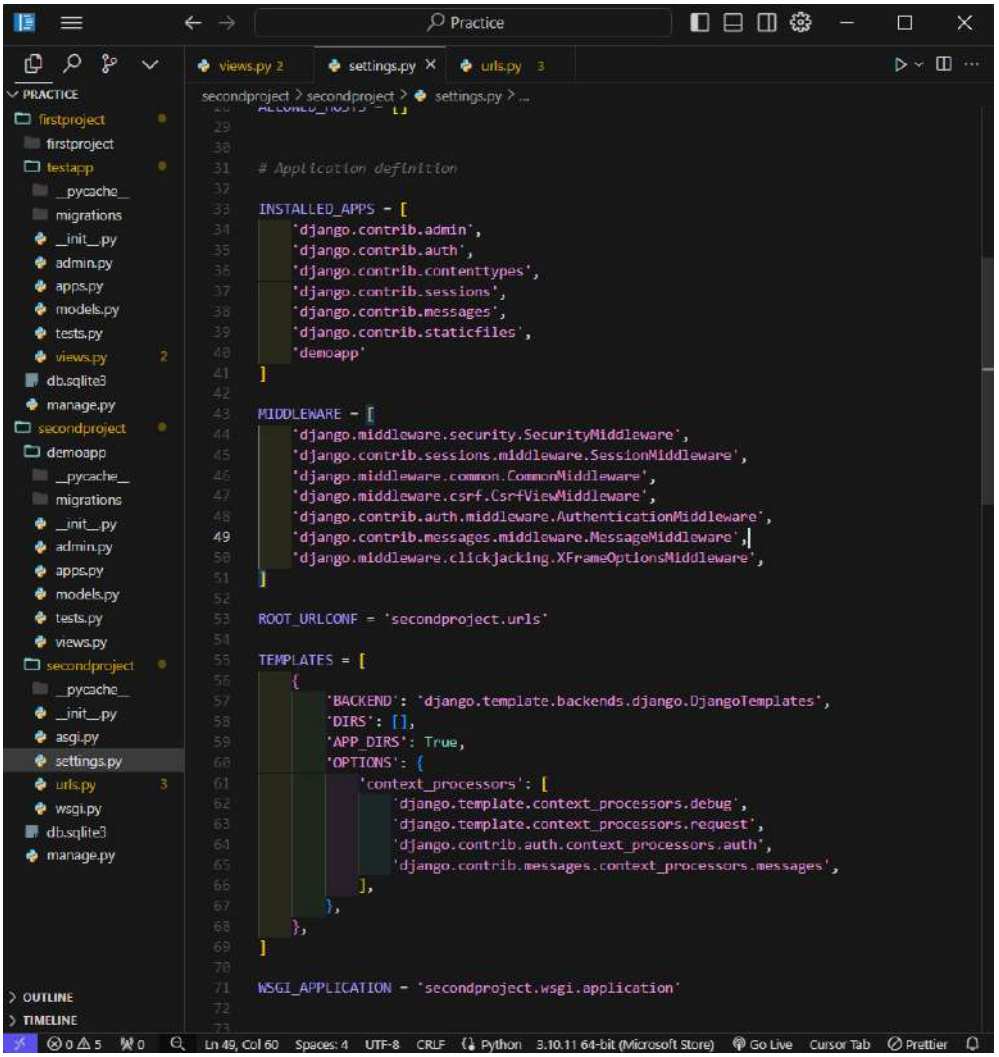
You don't get an error because Django doesn't require an app to be listed in `INSTALLED_APPS` just to use its views. `INSTALLED_APPS` is needed for things like database models, admin interfaces, and templates. But for basic tasks like serving a webpage, Django just needs the view function to be correctly imported and linked in `urls.py`.

So even though `demoapp` isn't in `INSTALLED_APPS`, Django can still find and run the view you wrote because it's imported directly in your code.

```
791
792 urlpatterns = [
793     path('admin/', admin.site.urls),
794     path('time/', views.time_info_view),
795 ]
796
797
798 6. Run server related to secondproject
799
800 py manage.py runserver
801
802 7. Send the requrest with correct url pattern
803
804 http://127.0.0.1:8000/time
805 http://localhost:8000/time
806
807
```

```
Single Application, But multiple views
Single project but multiple Applications
Defining urlpatterns at application level instead of project level
```


Khushal Savalakha



> 💡 tell me when we make any project then why we got 2 folder with same name what is logic and reason behind it ?

In Django, when you create a project, you usually end up with two directories that have the same name. Here's why this happens and the logic behind it:

Outer Directory (Project Root)

- **Purpose**: This is the main directory for your Django project. It contains all the files and directories that belong to the project.
- **Contents**: It typically includes:
  - **manage.py**: A command-line utility that helps you interact with your Django project.

- **Additional folders**: These might include `apps`, `templates`, `static`, and more as your project grows.

Inner Directory (Project Configuration)

- **Purpose**: This is a Python package (i.e., a directory with an `__init__.py` file) that contains the project's settings and configuration.
- **Contents**: It typically includes:
  - **settings.py**: Configuration settings for your project.
  - **urls.py**: URL declarations for your project.
  - **wsgi.py**: An entry-point for WSGI-compatible web servers to serve your project.
  - **asgi.py**: An entry-point for ASGI-compatible web servers (for handling WebSockets and other asynchronous features).

Why Two Directories?

- **Separation of Concerns**: The outer directory is more of a container for your project, while the inner directory is specifically for configuration. This separation helps in organizing the project structure cleanly.
- **Project Flexibility**: Having the inner directory as a package allows you to import settings and configurations easily if you need to run the project in different environments or under different configurations.
- **Avoiding Confusion**: By having an outer directory with the same name as the project, it’s easier to understand that everything inside this directory belongs to a single project.

So, the two folders serve different purposes: one organizes the entire project, and the other focuses on configuring the project's behavior.

```
from django.shortcuts import render
import datetime
from django.http import HttpResponse
# Create your views here.

def time_info_view(request):
    time=datetime.datetime.now()
    # s='<h1>Hello current Date and Time:'+str(time)+'</h
```


Khushal Savalakha

```
    s='<h1>Hello current Date and Time:{{time}}</h1>'
   return HttpResponse(s)
why it gives error ?
```

The error occurs because you are using Django template syntax (`{{time}}`) directly in a string within the view function, which Django does not interpret as template code when generating the `HttpResponse`.

In Django, if you want to use template syntax like `{{time}}`, you should create a separate HTML template file and pass the context (i.e., the variables you want to use) to that template. Django will then render the template with the provided context.

Here's how you can correct it:

1. **Create a template file**:

   Let's create a template file called `time_info.html` inside your `templates` directory.

   ```
   <!-- time_info.html -->
   <h1>Hello current Date and Time: {{ time }}</h1>
   ```

2. **Modify the view**:

   Update your `time_info_view` function to render the template and pass the current time as context.

   ```
   from django.shortcuts import render
   import datetime

   def time_info_view(request):
       current_time = datetime.datetime.now()
       return render(request, 'time_info.html', {'tim
   e': current_time})
   ```

This way, the `time_info.html` template will receive the `time` variable, and Django will replace `{{ time }}` with the actual value of `current_time` before sending the response to the client.

If you want to use inline HTML with dynamic content directly in the view function (without templates), you should format the string in Python:

```
from django.http import HttpResponse
import datetime
```


Khushal Savalakha

```
def time_info_view(request):
    current_time = datetime.datetime.now()
    s = f'<h1>Hello current Date and Time: {current_ti
me}</h1>'
    return HttpResponse(s)
```

This version uses Python's f-string formatting to include the `current_time` directly in the HTML string.

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('time/',views.time_info_view)
]
what is the difference between

urlpatterns = [
    path(r'^admin/', admin.site.urls),
    path(r'^time/',views.time_info_view)
]
?
```

The difference between the two versions of `urlpatterns` involves the use of regular expressions (`r'^pattern/'`) in the second version versus the plain string paths in the first version.

1. First Version: Plain String Paths

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('time/', views.time_info_view),
]
```

- **Explanation**: This version uses the `path()` function with plain string patterns, which are more readable and recommended in Django 2.0 and later. The `path()` function automatically handles routing without needing regular expressions.
- **Use Case**: It's simple and clear when the URL patterns are straightforward, such as paths that don't require complex matching.

Khushal Savalakha

2. Second Version: Regular Expressions

```
urlpatterns = [
    path(r'^admin/', admin.site.urls),
    path(r'^time/', views.time_info_view),
]
```

- **Explanation**: The `r'^pattern/'` syntax indicates that the patterns are raw string regular expressions. The caret (`^`) anchors the match to the start of the string, which is unnecessary for simple routes. In older versions of Django (pre-2.0), regular expressions were used by default in URL patterns, so you'd often see `url()` instead of `path()`.
- **Use Case**: If your URL patterns need to match more complex conditions, such as optional parameters or specific formats, you'd use regular expressions. However, for simple paths like `/admin/` and `/time/`, regular expressions are overkill and less readable.

Key Takeaway:

- **Use `path()` with plain strings** (first version) for simple routes in Django 2.0 and later, as it's more readable and sufficient for most cases.
- **Use regular expressions** (second version) only if you need complex pattern matching. For most modern Django projects, this is rarely needed.

```
1 import datetime
2 date = datetime.datetime.now()
3 print(date)
4 h = int(date.strftime('%H'))
5 print(h)
```


Khushal Savalakha

```
import datetime
date=datetime.datetime.now()
print(date)
2024-08-15 15:03:27.624087
h=int(date.strftime('%H'))
print(h)
15
s=int(date.strftime('%I'))
print(s)
3
```

Let's break down the code step by step:

1. Import the `datetime` Module

```
import datetime
```

- This imports the `datetime` module, which provides classes for manipulating dates and times.

2. Get the Current Date and Time

```
date = datetime.datetime.now()
```


Khushal Savalakha

- `datetime.datetime.now()` returns the current date and time as a `datetime` object.
- The output of `print(date)` would show something like `2024-08-15 15:03:27.624087`, which includes the date, hour, minute, second, and microsecond.

3. Extract the Hour in 24-Hour Format

```
h = int(date.strftime('%H'))
```

- `date.strftime('%H')` formats the `date` object as a string, specifically extracting the hour in **24-hour format**.
- In your example, the `date` is `2024-08-15 15:03:27.624087`, so `%H` would return `"15"`, which is the hour in 24-hour format (3 PM).
- `int()` converts the string `"15"` into an integer `15`.
- The result is stored in the variable `h`, and `print(h)` outputs `15`.

4. Extract the Hour in 12-Hour Format

```
s = int(date.strftime('%I'))
```

- `date.strftime('%I')` formats the `date` object as a string, specifically extracting the hour in **12-hour format**.
- In your example, the `date` is `2024-08-15 15:03:27.624087`, so `%I` would return `"03"`, which is the hour in 12-hour format (3 PM).
- `int()` converts the string `"03"` into an integer `3`.
- The result is stored in the variable `s`, and `print(s)` outputs `3`.

Explanation of `%H` and `%I`

- `%H`: This format code extracts the hour in **24-hour format**. It ranges from `00` (midnight) to `23` (11 PM). For example, 3 PM in 24-hour format would be `15`.
- `%I`: This format code extracts the hour in **12-hour format**. It ranges from `01` to `12`, with `12` representing both midnight and noon. For example, 3 PM in 12-hour format would be `03`.

Summary:

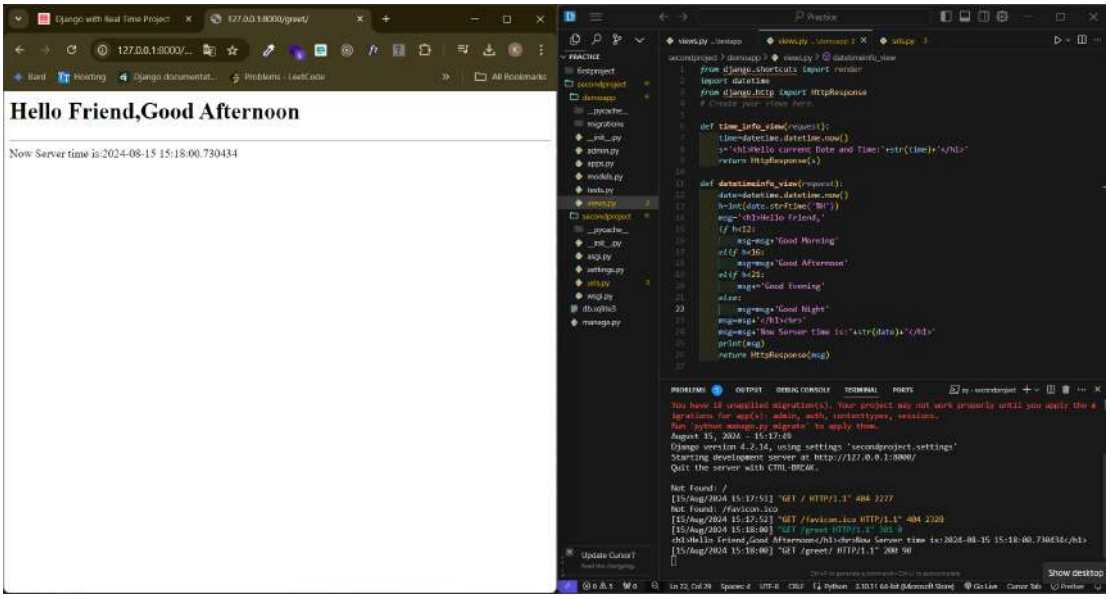
- `%H` gives the hour in 24-hour format (e.g., `15` for 3 PM).

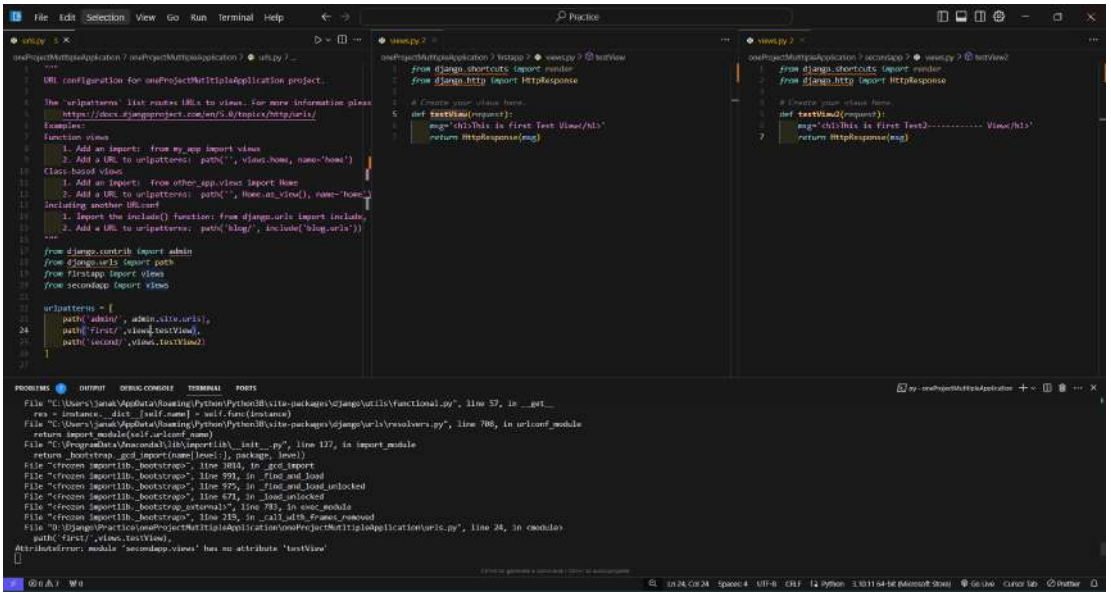
Khushal Savalakha

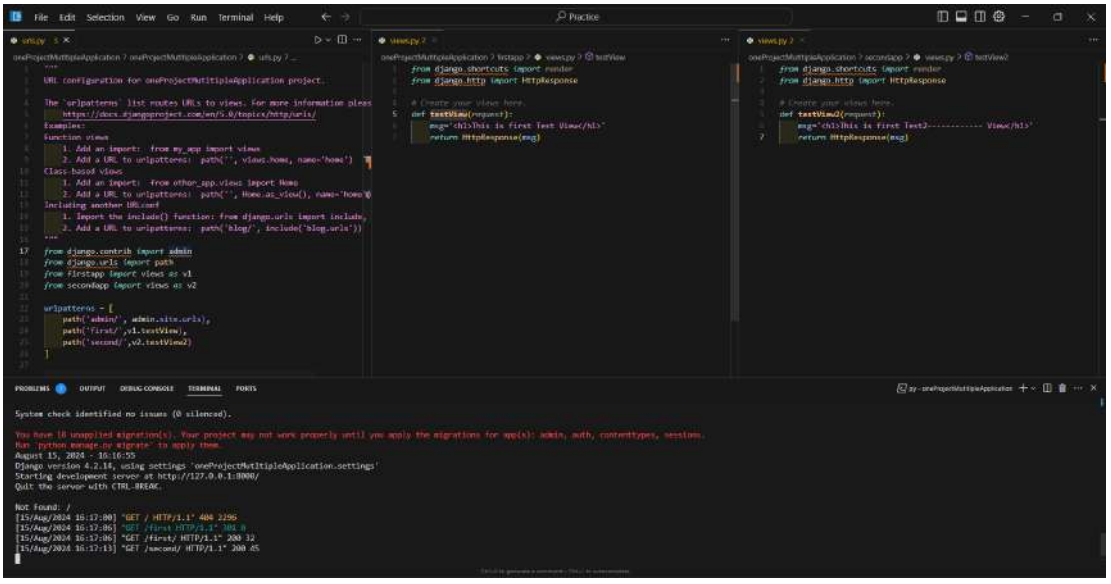
- `%I` gives the hour in 12-hour format (e.g., `03` for 3 PM).
- The code extracts and prints the current hour in both formats.

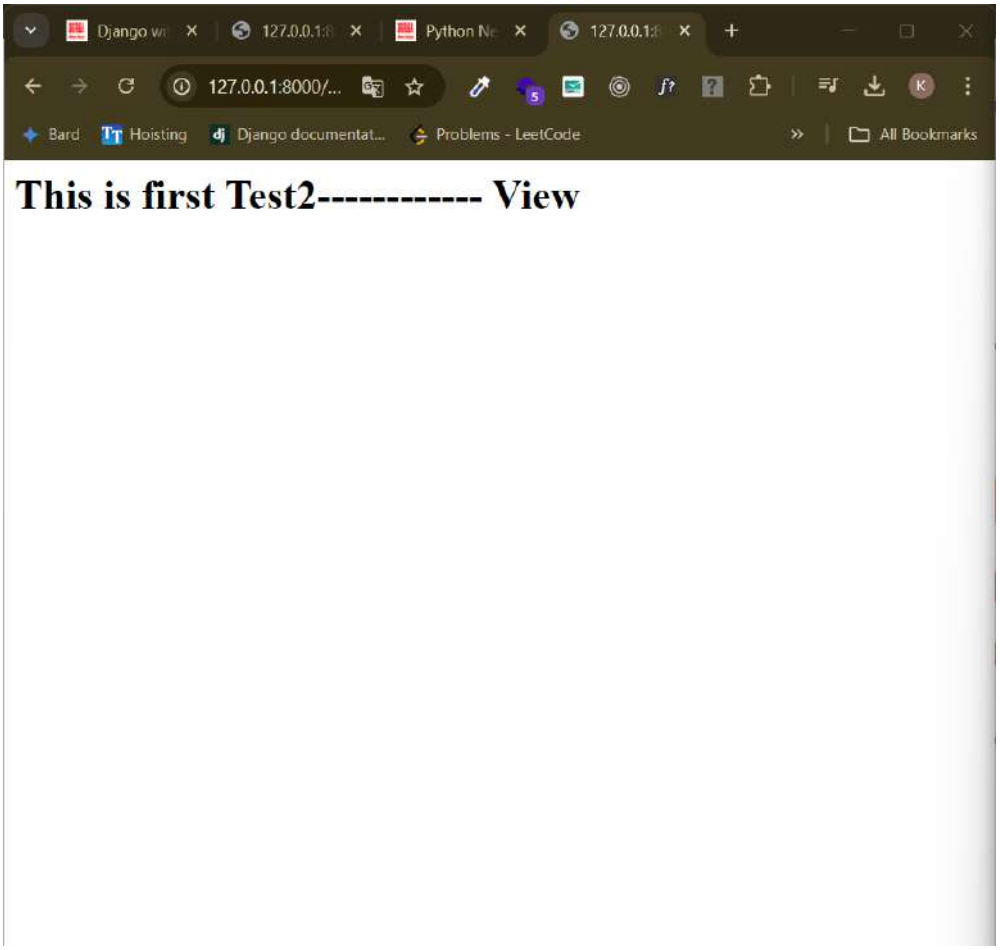
```
from django.shortcuts import render
import datetime
from django.http import HttpResponse
# Create your views here.

def time_info_view(request):
    time=datetime.datetime.now()
    s='<h1>Hello current Date and Time:'+str(time)+'</h1>
    return HttpResponse(s)

def datetimeinfo_view(request):
    date=datetime.datetime.now()
    h=int(date.strftime('%H'))
    msg='<h1>Hello Friend,'
    if h<12:
        msg=msg+'Good Morning'
    elif h<16:
        msg=msg+'Good Afternoon'
    elif h<21:
        msg+='Good Evening'
    else:
        msg=msg+'Good Night'
    msg=msg+'</h1><hr>'
    msg=msg+'Now Server time is:'+str(date)+'</h1>'
    print(msg)
    return HttpResponse(msg)
```

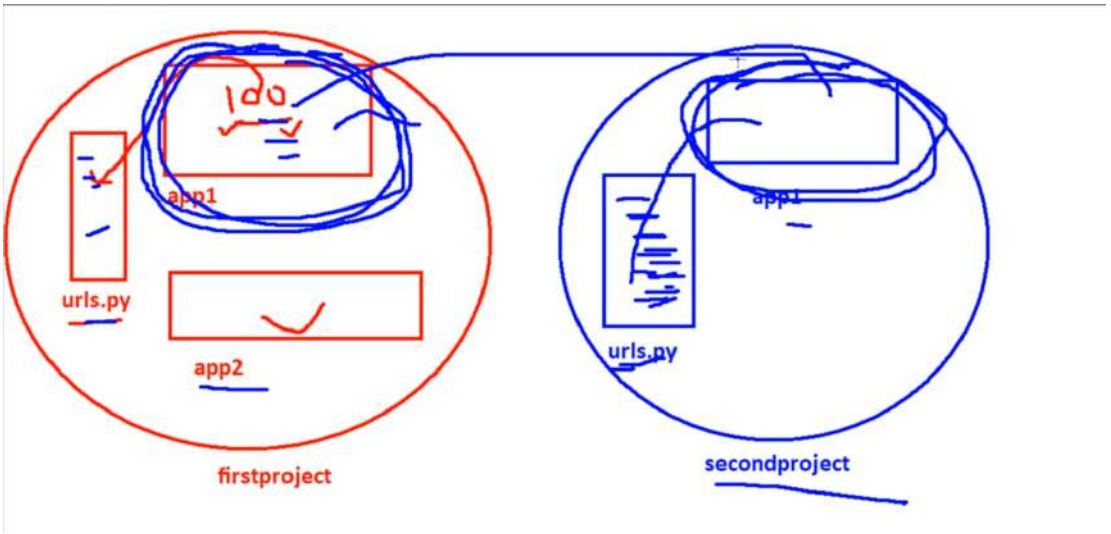

Khushal Savalakha


Hello Friend,Good Afternoon
Now Server time is:2024-08-15 15:18:00.730434










```
"""
URL configuration for baseproject project.

The `urlpatterns` list routes URLs to views. For more inf
    https://docs.djangoproject.com/en/5.0/topics/http/url
```

```
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, na
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view()
Including another URLconf
    1. Import the include() function: from django.urls im
    2. Add a URL to urlpatterns:  path('blog/', include('
"""
#data of baseproject
from django.contrib import admin
from django.urls import path,include
from testapp import urls

urlpatterns = [
    path('admin/', admin.site.urls),
    path('testapp/',include('testapp.urls')),
]
```

```
from django.urls import path
from .import views
#testapp application
urlpatterns=[
     path('first/',views.first_view),
    path('second/',views.second_view),
    path('third/',views.third_view),
    path('forth/',views.forth_view),
    path('fifth/',views.fifth_view)
]
```

http://127.0.0.1:8000/testapp/first

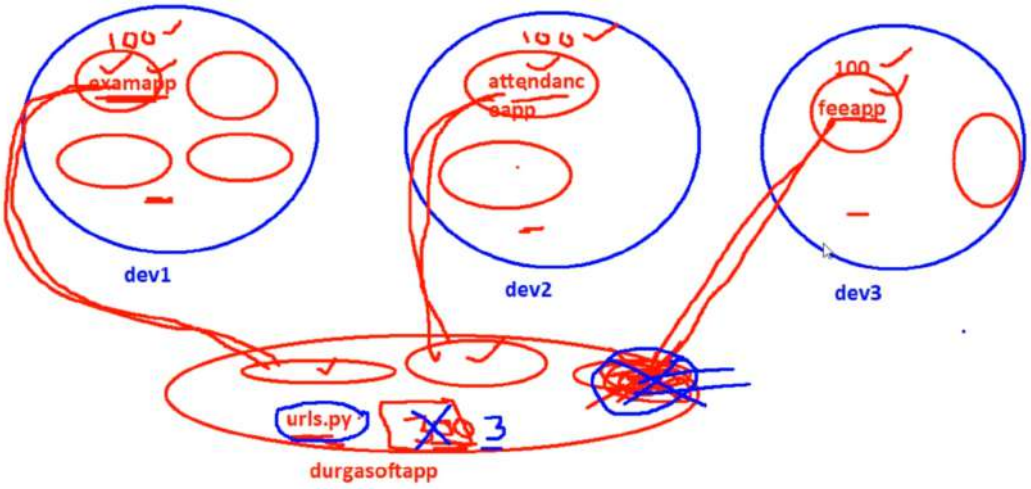
project level urls.py → identify the corresponding application level urls file

application level urls file it will identify view function and execute that view function , provide required response

Khushal Savalakha

Third View Response

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('testapp/', include('testapp.urls')),
]

application level:
------------------
from django.urls import path
from . import views
urlpatterns = [
  path('first/',views.first_view),
  path('second/',views.second_view),
  path('third/',views.third_view),
  path('fourth/',views.fourth_view),
  path('fifth/',views.fifth_view),
]
```



```
steps-1:
--------
1. start project
2. start app
3. add app in the settings.py
4. define views functions inside views.py
5. create urls.py file inside application and define all required urls for our view functions.
```

```
Advantages:
1. It promotes reusability of django applications across multiple projects.
2. project level urls.py file will be clean and more readable.
3. Maintainability of the project will be improved.
```

▼ Set Url name

To write the `href` attributes in your HTML files that correspond to the URL patterns you've defined in your Django `urlpatterns`, you can use the `name` you provided in the `path` function along with Django's `{% url %}` template tag. Here's how you can write the `href` attributes for the paths in your `urlpatterns`:

1. **Home Page (`views.home_page_view`):**

   ```
   <a href="{% url 'home' %}">Home</a>
   ```

   Since you didn't assign a name to the home path in your `urlpatterns`, you can either add a name to this path or use the URL directly:

   ```
   path('', views.home_page_view, name='home')
   ```

   Then use:

   ```
   <a href="{% url 'home' %}">Home</a>
   ```

2. **Java Examples Page (`views.java_page_view`):**

   ```
   <a href="{% url 'java' %}">Java Examples</a>
   ```

3. **Python Examples Page (`views.python_page_view`):**

   ```
   <a href="{% url 'python' %}">Python Examples</a>
   ```

4. **Aptitude Examples Page (`views.aptitude_page_view`):**

Khushal Savalakha

```
<a href="{% url 'aptitude' %}">Aptitude Examples</a>
```

If you haven't named your `home` path, and you want to link to it without using a name, you can use:

```
<a href="/">Home</a>
```

This approach ensures that your links will work even if the URL structure changes, as they are based on the `name` attribute you defined in the `urlpatterns`.

▼ Django Templates And Static Files

```
Django Templates:
------------------
View meant for business logic
Template meant for presentation logic/frontend

MVC design pattern/architecture:
--------------------------------
M--->Model (Business logic)
V--->View (Presentation logic)
C--->Controller(coordination)

MVT design pattern/architecture::
---------------------------------
M--->Model
V--->View(Business logic-->Python file)
T--->Template(presentation logic--->html file)

```

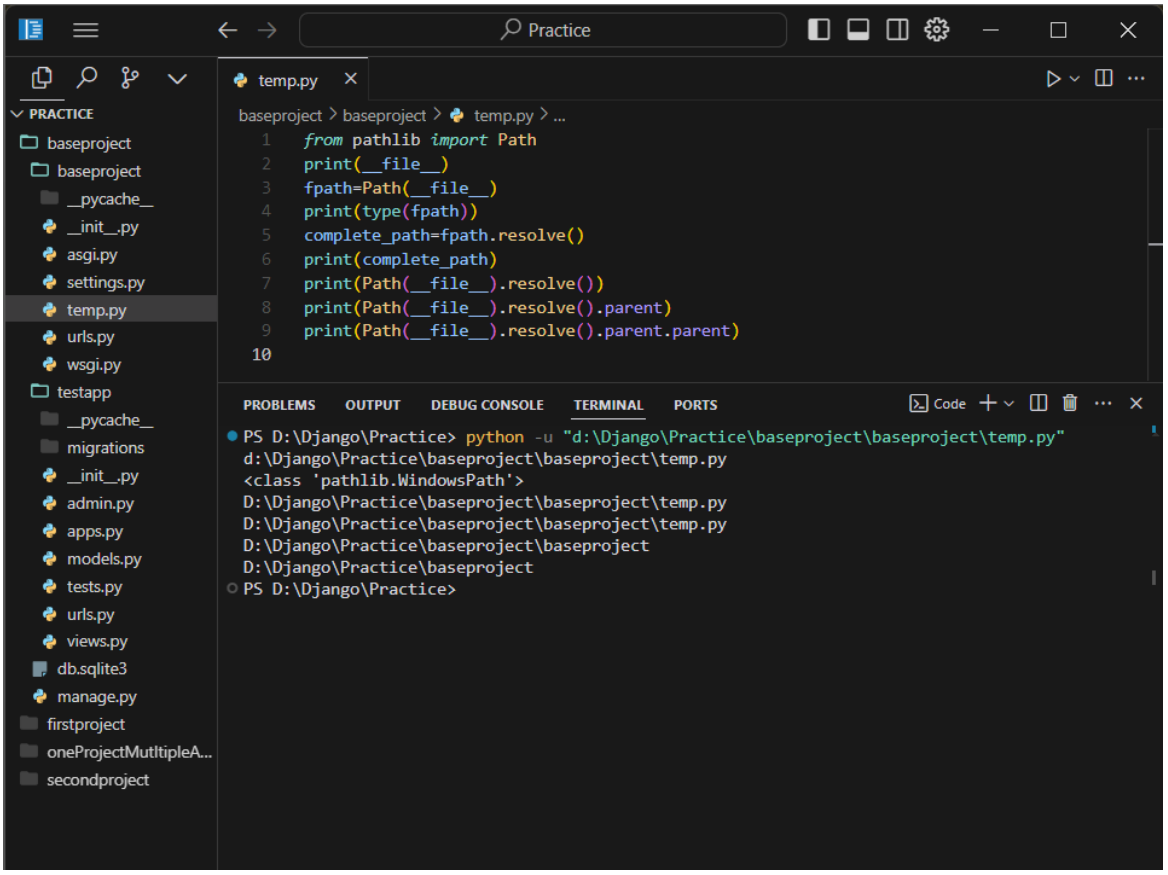
```
It is not recommended to write html inside python file(views.py) because

1. It reduces readability, because python code mixed with html code
2. No separation of roles. Python developer has to take care about both python and
html code.
3. It does not promote reusability of the code.

We can overcome these problems by separating html code into separate html file.
This html file is nothing but template.
From views.py we can use these template files based on our requirement.
```

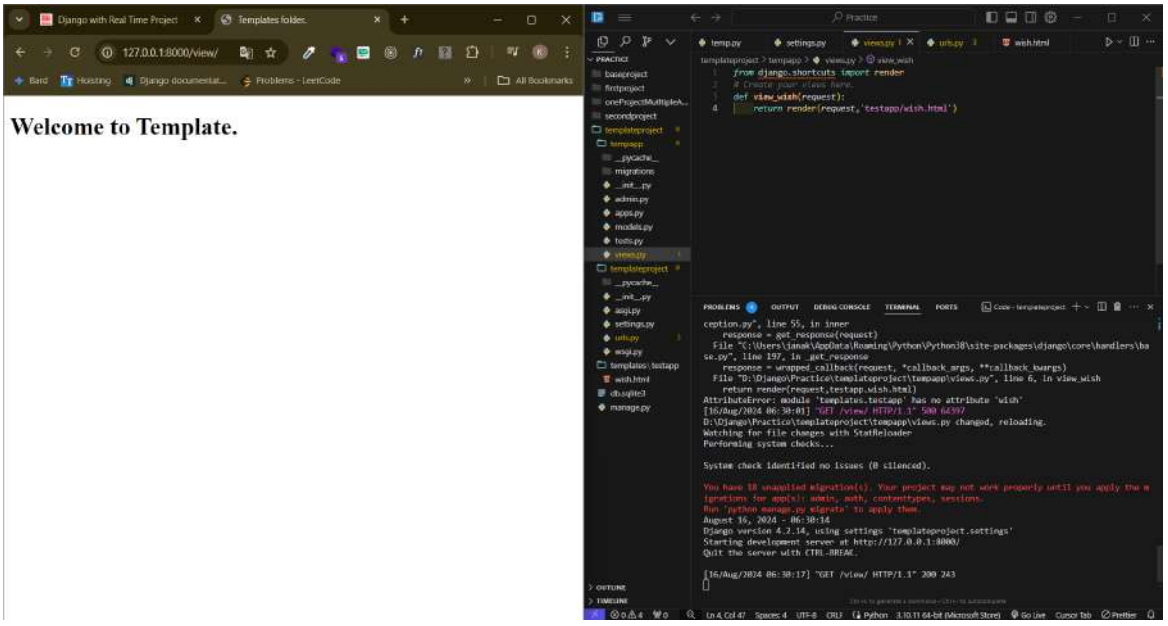
```
Python stuff required to develop template based application
-----------------------------------------------------------
pathlib --->module name
Path--->class name

pathlib module provides various classes representing underlying operating system
specific paths. This is Python's inbuilt module.
```


Khushal Savalakha

```python
from pathlib import Path
print(__file__) #It returns the name of the file: test.py
fpath = Path(__file__)
print(type(fpath)) #<class 'pathlib.WindowsPath'>
complete_path=fpath.resolve()
print(complete_path) #D:\durgaclasses\test.py
print(Path(__file__).resolve()) #D:\durgaclasses\test.py
print(Path(__file__).resolve().parent) #D:\durgaclasses
print(Path(__file__).resolve().parent.parent) #D:\


D:\durgaclasses>py test.py
test.py
<class 'pathlib.WindowsPath'>
D:\durgaclasses\test.py
D:\durgaclasses\test.py
D:\durgaclasses
D:\
```

```
from pathlib import Path
print(__file__)
fpath=Path(__file__)
print(type(fpath))
complete_path=fpath.resolve()
print(complete_path)
print(Path(__file__).resolve())
print(Path(__file__).resolve().parent)
print(Path(__file__).resolve().parent.parent)
```






```
# Build paths inside the project like this: BASE_DIR / 's
ubdir'.
BASE_DIR = Path(__file__).resolve().parent.parent
```

```
# print('-->',BASE_DIR) --> D:\Django\Practice\templatepr
oject

TEMPLATE_DIR=BASE_DIR/'templates'

# Quick-start development settings - unsuitable for produ
ction
# See https://docs.djangoproject.com/en/5.0/howto/deploym
ent/checklist/

# SECURITY WARNING: keep the secret key used in productio
n secret!
SECRET_KEY = 'django-insecure-7kb!%(2u2$up3*#!%gaez0is#t1
0wr60_37-t_6j97uchn%8!('

# SECURITY WARNING: don't run with debug turned on in pro
duction!
DEBUG = True

ALLOWED_HOSTS = []


# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'tempapp'
]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddlewar
e',
    'django.middleware.common.CommonMiddleware',
```


Khushal Savalakha

```
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddlew
are',
    'django.contrib.messages.middleware.MessageMiddlewar
e',
    'django.middleware.clickjacking.XFrameOptionsMiddlewa
re',
]

ROOT_URLCONF = 'templateproject.urls'

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.Djang
oTemplates',
        'DIRS': [TEMPLATE_DIR],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debu
g',
                'django.template.context_processors.reque
st',
                'django.contrib.auth.context_processors.a
uth',
                'django.contrib.messages.context_processo
rs.messages',
            ],
        },
    },
]


├───templateproject
│   │   asgi.py
│   │   settings.py
│   │   urls.py
│   │   wsgi.py
│   │   __init__.py
```


Khushal Savalakha

```
|    |
|    └───__pycache__
|            settings.cpython-38.pyc
|            urls.cpython-38.pyc
|            wsgi.cpython-38.pyc
|            __init__.cpython-38.pyc
|
└───templates
    └───testapp
            wish.html

-->views.py
from django.shortcuts import render
# Create your views here.
def view_wish(request):
    return render(request,'testapp/wish.html')
here python saves all path which is important by this if
we write just import statement it will find it's path jus
t like this in templates if we don't write TEMPLATE_DIR t
hen it will not able to find path of wish.html am i true
?
```

Your statements and understanding of how Django handles paths and template rendering are generally correct, but I'll clarify a few details to make sure everything is accurate.

Key Points:

1. **Python Path Resolution**:
   - When you define paths like `BASE_DIR = Path(__file__).resolve().parent.parent`, Python resolves the path to the directory where your Django project resides. This is crucial because Django needs to know where to find files like templates, static files, and the database.
2. **Template Resolution**:
   - The `TEMPLATE_DIR = BASE_DIR / 'templates'` line defines the location where your templates will be stored. This directory is passed to Django in the `TEMPLATES` configuration under the `DIRS` option. By doing this, Django knows to look for templates in the `templates` directory when rendering views.
3. **Using the `render()` Function**:

Khushal Savalakha

- In your `views.py`, the `render()` function combines a template with a context (optional) and returns an HttpResponse object. When you call `render(request, 'testapp/wish.html')`, Django searches for the `wish.html` template within the directories specified in the `TEMPLATES` setting, including the `TEMPLATE_DIR`.

4. **APP_DIRS Option**:
   - The `APP_DIRS: True` setting in the `TEMPLATES` configuration tells Django to automatically look for templates inside each app's `templates` directory. If your app `tempapp` has a `templates/testapp/` folder, Django will look there too.
5. **Why Paths Are Important**:
   - Paths in Django (like `TEMPLATE_DIR`) ensure that Django can locate resources like templates, static files, and the database. Without correctly defined paths, Django won't be able to find the necessary files, and your application may throw errors.

Summary:

- **Correct**: If you don't define the `TEMPLATE_DIR` or set up the paths properly, Django won't know where to find `wish.html`.
- **Correct**: Python handles path resolution based on how you define them in the code, allowing Django to locate files in your project structure.

So, your overall understanding is spot-on! Paths in Django (and Python in general) are crucial for ensuring that the framework can find and use the files it needs to operate correctly.

> 💡 I am just asking that here wish.html file is in templates/testapp so every time django find path every directory except templates when i define path TEMPLATE_DIR templates but this time it will go in templates folder and find wish.html am i true ?

Yes, you're correct!

Here's a more detailed explanation:

1. **When You Define `TEMPLATE_DIR`**:
   - By defining `TEMPLATE_DIR = BASE_DIR / 'templates'` and including this in the `DIRS` option within the `TEMPLATES` setting, you are telling Django to specifically look in the `templates` directory at the project level.
2. **How Django Searches for Templates**:

Khushal Savalakha

- When you call `render(request, 'testapp/wish.html')`, Django first looks in the directories specified by `DIRS` in the `TEMPLATES` setting. Since you defined `TEMPLATE_DIR` as one of these directories, Django will look inside `BASE_DIR/templates/`.
- Within the `templates` directory, Django will look for a subdirectory named `testapp` and then the file `wish.html`.

3. **Without `TEMPLATE_DIR`**:
   - If you hadn't defined `TEMPLATE_DIR`, Django would still look for templates, but it would rely on the `APP_DIRS` setting. With `APP_DIRS: True`, Django would look for `wish.html` inside `tempapp/templates/testapp/wish.html` by default (since `APP_DIRS` enables Django to search for templates inside each app's `templates` directory).
4. **In Your Case**:
   - Since you've specified `TEMPLATE_DIR`, Django knows to start its search inside the `templates` directory. So yes, every time Django tries to find `wish.html`, it will look inside the `templates/testapp/` directory because of the `TEMPLATE_DIR` path definition.

Summary:

- **True**: By defining `TEMPLATE_DIR`, Django knows to start its search inside the `templates/` folder.
- **True**: Django will then look inside `testapp/` within the `templates/` folder to find `wish.html`.

↳ * * inside `tempapp/templates/testapp/wish.html` by default (since `APP_DIRS` enables Django to search for templates inside each app's `templates` directory).

but here templates is not app's directory so basically it will not go to search in templates folder am i true ?

Yes, you're correct. If the `templates` folder is **not** inside the app's directory (e.g., `tempapp/templates`), Django will **not** automatically search in that folder unless it is explicitly specified in the `DIRS` setting within the `TEMPLATES` configuration.

Here's how it works:

1. **App Directory Templates (When `APP_DIRS` is `True`)**:

Khushal Savalakha

- When `APP_DIRS` is set to `True`, Django will automatically look for templates inside each app's `templates` directory. For example, if you have a folder structure like `tempapp/templates/testapp/wish.html`, Django will search there without needing to specify this path explicitly in `DIRS`.

2. **Project-Level Templates Directory**:

   - However, if your `templates` directory is at the **project level** (not inside an app), Django **will not** automatically search this directory unless you explicitly add it to the `DIRS` list in the `TEMPLATES` setting.

   For example, if you have a directory structure like:

```
└──templates
    └──testapp
        └──wish.html
```

   And the `templates` folder is **not** inside any specific app, you must specify this path using `TEMPLATE_DIR` and include it in `DIRS`. Otherwise, Django won't look there.

In Your Case:

- Since you are using a project-level `templates` directory (outside of `tempapp`), Django will only search for `wish.html` inside `templates/testapp/` because you've defined `TEMPLATE_DIR` in the `DIRS` setting.
- If you hadn't defined this `TEMPLATE_DIR`, Django would **not** automatically search the `templates/` folder, and it wouldn't find `wish.html` unless it was inside the app's `templates` directory.

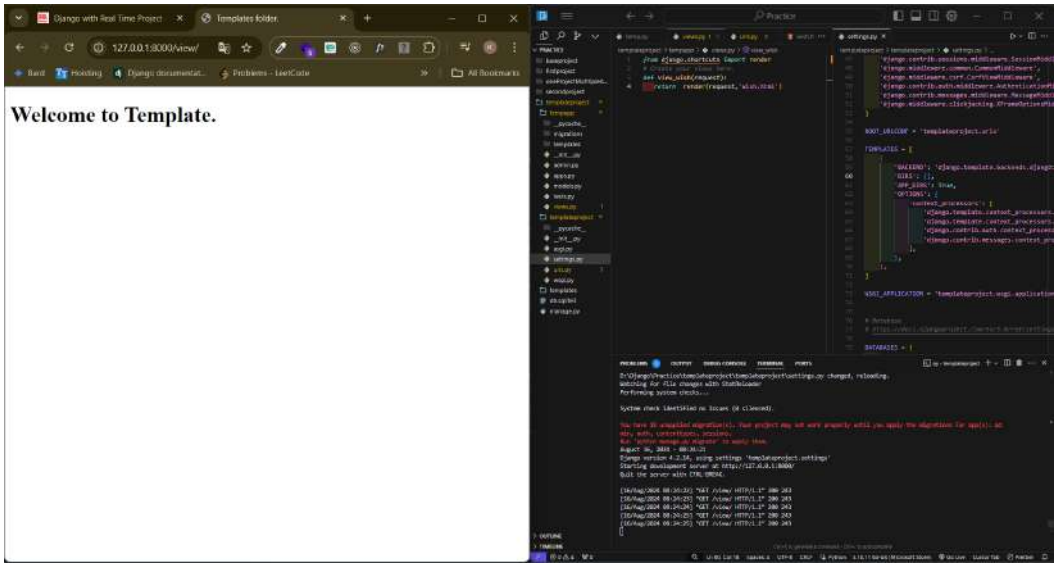
Summary:

- **Correct**: If the `templates` folder is at the project level and not inside an app, Django won't automatically search there unless you specify it in `DIRS`.
- **Correct**: With `APP_DIRS: True`, Django will search for templates inside each app's `templates` directory, but not in a project-level `templates` directory unless configured explicitly.

💡 **but still if we define testapp at in tempapp application django still not able to find the wish.html because it does not have data about it .**

Khushal Savalakha

```
Java or .Net based web applications

MVC Architecture
-----------------
M--->Model (Business logic)
V--->View (Presentation logic)
C--->Controller(coordination activities)


Django follows MVT:
---------I---------
M--->Model(Database related)
V--->View (Business Logic)
T--->Templates(Presentation Logic)
```



> 💡 it means templates is an default keyword if we assign other name to folder then it will do not find data in that folder even if it is application level folder am i true ?

Yes, you're correct. The name `templates` is a conventional folder name that Django recognizes by default when `APP_DIRS: True` is set. If you change the folder name to something else (like `custom_templates` or `views_html`), Django will **not** automatically find templates in that folder, even if it is at the application level.

Explanation:

Khushal Savalakha

1. **Default Behavior with `APP_DIRS: True`**:
   - When `APP_DIRS` is set to `True`, Django automatically searches for templates inside each app's `templates` folder. For example, if you have a structure like `tempapp/templates/`, Django will find templates in this directory.
2. **Changing the Folder Name**:
   - If you rename the `templates` folder to something else, like `custom_templates`, Django will **not** know to look there, even if `APP_DIRS` is `True`. Django specifically looks for folders named `templates`.
3. **Manually Specifying Other Folders**:
   - If you want Django to look in a folder with a different name (e.g., `custom_templates`), you must explicitly specify that path in the `DIRS` setting within the `TEMPLATES` configuration in `settings.py`. For example:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.djang
o.DjangoTemplates',
        'DIRS': [BASE_DIR / 'custom_templates'],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processor
s.debug',
                'django.template.context_processor
s.request',
                'django.contrib.auth.context_proces
sors.auth',
                'django.contrib.messages.context_pr
ocessors.messages',
            ],
        },
    },
]
```

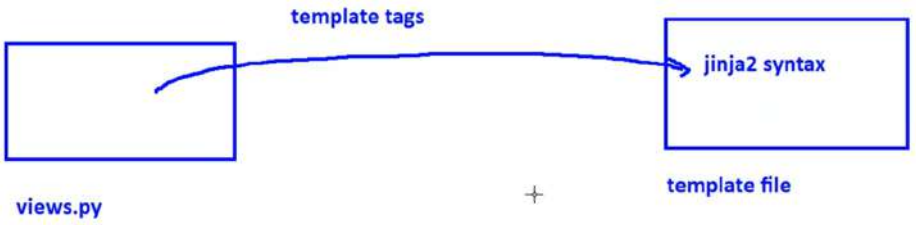
   - This configuration tells Django to look for templates in the `custom_templates` directory.

Khushal Savalakha

Summary:

- **Correct**: If you change the folder name from `templates` to something else, Django will not find templates in that folder unless you explicitly specify the new folder path in the `DIRS` setting.
- The name `templates` is recognized by Django by default, and if you deviate from this, manual configuration is required.

```
Django Templates:
-----------------
View meant for business logic
Template meant for presentation logic/frontend

MVC design pattern/architecture:
--------------------------------
M--->Model (Business logic)
V--->View (Presentation logic)
C--->Controller(coordination)

MVT design pattern/architecture::
---------------------------------
M--->Model
V--->View(Business logic-->Python file)
T--->Template(presentation logic--->html file)
```



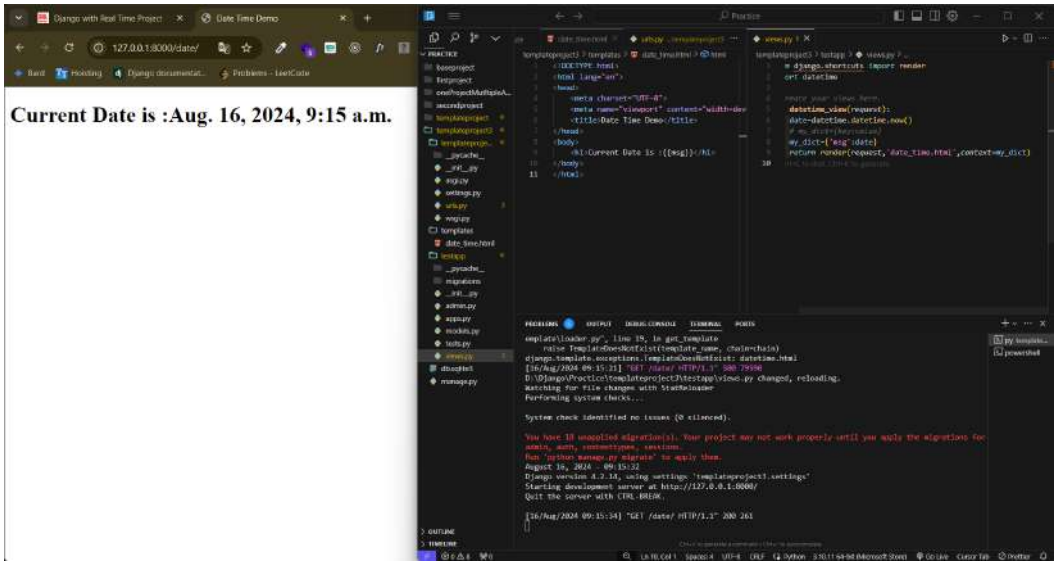

```
template tags.

Template tags also known as template variables.

Take special care about template tags. It is neither python syntax nor html syntax.

It it Jinja2 Syntax.
template tag syntax for inserting text data: {{msg}}
```

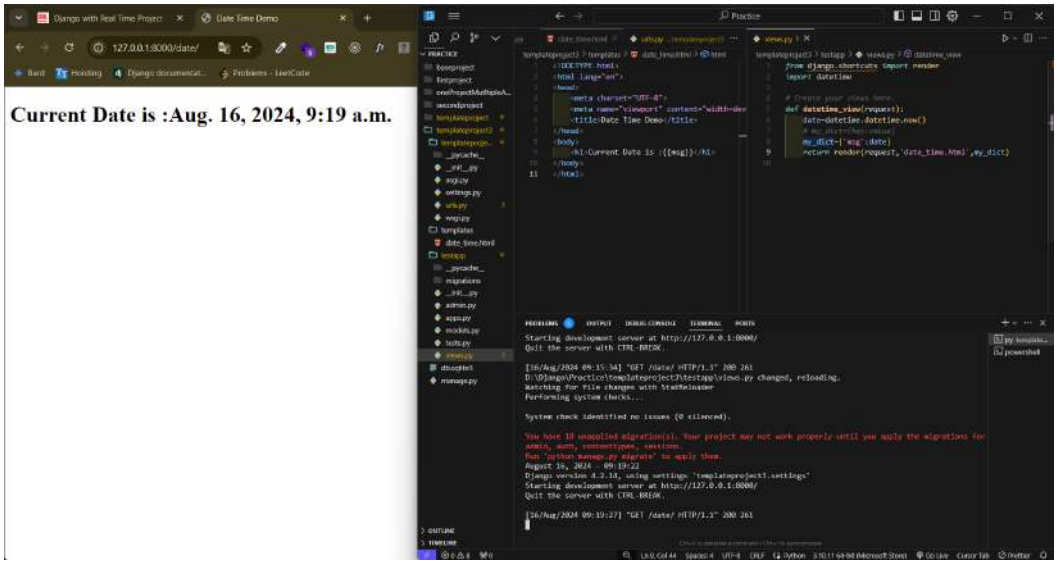
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initi
    <title>Date Time Demo</title>
```


Khushal Savalakha

```
</head>
<body>
    <h1>Current Date is :{{msg}}</h1>
</body>
</html>
```

```
from django.shortcuts import render
import datetime

# Create your views here.
def datetime_view(request):
    date=datetime.datetime.now()
    # my_dict={key:value}
    my_dict={'msg':date}
    return render(request,'date_time.html',context=my_dict)
```





it is not necessory to write context while sending data through jinja2

```
from django.shortcuts import render
import datetime

# Create your views here.
def datetime_view(request):
    date=datetime.datetime.now()
    # my_dict={key:value}
    # my_dict={'msg':date}
    return render(request,'date_time.html',{'msg':date})
#this is also working
```

```
from django.shortcuts import render
import datetime
import random

# Create your views here.
def datetime_view(request):
    date=datetime.datetime.now()
    # my_dict={key:value}
    # my_dict={'msg':date}
    return render(request,'date_time.html',{'msg':date})


def result_view(request):
```


Khushal Savalakha

```
    msg_list=[
        'Very Soon,you are going to get marriage !!!',
        'The Golden days ahead',
        'Better to sleep more time even in office',
        'Tomorrow wil be the best day of your life',
        'Tommorrow is the perfect day .',
        'very soon you will get job'
    ]
    names_list=['Sunny','Mallika','Katrina','Kareena','Samar
    time=datetime.datetime.now()
    h=int(time.strftime('%H'))
    if h<12:
        s='Good Morning'
    elif h<16:
        s='Good Afternoon'
    elif h<21:
        s='Good Evening'
    else:
        s='Good Night'
    name=random.choice(names_list)
    msg=random.choice(msg_list)
    my_dict={'time':time,'name':name,'msg':msg,'wish':s}
    return render(request,'astrology.html',my_dict)
```

Jinja2.pdf

3.1 Synopsis

A template is simply a text file. It can generate any text-based format (HTML, XML, CSV, LaTeX, etc.). It doesn't have a specific extension, `.html` or `.xml` are just fine.

A template contains **variables** or **expressions**, which get replaced with values when the template is evaluated, and tags, which control the logic of the template. The template syntax is heavily inspired by Django and Python.

Below is a minimal template that illustrates a few basics. We will cover the details later in that document:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>My Webpage</title>
</head>
<body>
    <ul id="navigation">
    {% for item in navigation %}
        <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
    {% endfor %}
    </ul>

    <h1>My Webpage</h1>
    {{ a_variable }}
</body>
</html>
```

This covers the default settings. The application developer might have changed the syntax from '`{% foo %}`' to '`<% foo %>`' or something similar.

There are two kinds of delimiers. '`{% ... %}`' and '`{{ ... }}`'. The first one is used to execute statements such as for-loops or assign values, the latter prints the result of the expression to the template.

Jinja2 is a templating language for Python that is used to generate HTML, XML, or other markup formats. It provides a powerful way to embed dynamic content and logic within templates. Below is a comprehensive overview of Jinja2 syntax, covering various features and combinations to help you create complex templates.

Basic Syntax

1. **Variables**

   Variables are enclosed in `{{ ... }}`.

   ```
   Hello, {{ name }}!
   ```

2. **Comments**

   Comments are enclosed in `{# ... #}`.

   ```
   {# This is a comment #}
   ```

3. **Expressions**

   You can perform calculations and string manipulations within expressions.

Khushal Savalakha

```
{{ 3 + 4 }}          {# Outputs: 7 #}
{{ "Hello" + "World" }} {# Outputs: HelloWorld #}
```

Control Structures

1. **If Statements**

   Conditional logic using `if`, `elif`, and `else`.

   ```
   {% if user.is_authenticated %}
     Welcome, {{ user.name }}!
   {% elif user.is_guest %}
     Welcome, Guest!
   {% else %}
     Please log in.
   {% endif %}
   ```

2. **For Loops**

   Loop through items in a list or dictionary.

   ```
   {% for item in items %}
     <p>{{ item.name }}: {{ item.value }}</p>
   {% endfor %}
   ```

   Loop with an index:

   ```
   {% for item in items %}
     <p>{{ loop.index }}: {{ item }}</p>
   {% endfor %}
   ```

3. **Macros**

   Define reusable code blocks similar to functions.

   ```
   {% macro input(name, value='', type='text') %}
     <input type="{{ type }}" name="{{ name }}" value="{{
   value }}">
   {% endmacro %}
   ```


Khushal Savalakha

```
{{ input('username') }}
```

4. **Filters**

   Modify the output of variables using filters.

```
{{ name|capitalize }}
{{ description|truncate(50) }}
```

Template Inheritance

1. **Blocks**

   Define sections that child templates can override.

```
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}Default Title{% endblock %}
</title>
</head>
<body>
  {% block content %}
  <p>This is the default content.</p>
  {% endblock %}
</body>
</html>
```

2. **Extending Templates**

   Create a child template that extends a parent template.

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

{% block title %}Home Page{% endblock %}

{% block content %}
<h1>Welcome to the Home Page</h1>
{% endblock %}
```


Khushal Savalakha

Advanced Features

1. **Set Variables**

   Assign values to variables.

   ```
   {% set total = 0 %}
   {% for item in items %}
     {% set total = total + item.price %}
   {% endfor %}
   Total: {{ total }}
   ```

2. **Include Statements**

   Insert the content of one template into another.

   ```
   {% include 'header.html' %}
   <p>Main content here...</p>
   {% include 'footer.html' %}
   ```

3. **Custom Filters and Tests**

   Define custom filters and tests in Python code and use them in templates.

   ```
   # Define a custom filter in Python
   def reverse_string(s):
       return s[::-1]

   # Register the custom filter
   env.filters['reverse'] = reverse_string
   ```

   ```
   {{ 'hello' | reverse }} {# Outputs: olleh #}
   ```

4. **Whitespace Control**

   Trim whitespace in templates.

   ```
   {% if condition -%}
   No extra spaces
   {%- endif %}
   ```


Khushal Savalakha

5. **Raw Blocks**

   Ignore template syntax within raw blocks.

   ```
   {% raw %}
   {{ This will not be processed }}
   {% endraw %}
   ```

Complex Combinations

Combining various Jinja2 features can help build complex logic and structures:

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

{% block title %}User Profile{% endblock %}

{% block content %}
  <h1>{{ user.name|capitalize }}'s Profile</h1>
  <p>Email: {{ user.email }}</p>

  {% if user.friends %}
    <h2>Friends</h2>
    <ul>
    {% for friend in user.friends %}
      <li>{{ friend.name }}{% if friend.online %} (Onlin
e){% endif %}</li>
    {% endfor %}
    </ul>
  {% else %}
    <p>No friends yet.</p>
  {% endif %}

  <h2>Actions</h2>
  <ul>
    {% if user.can_edit %}
      <li><a href="/edit-profile">Edit Profile</a></li>
    {% endif %}
    <li><a href="/logout">Log Out</a></li>
```


Khushal Savalakha

```
  </ul>
{% endblock %}
```

This example demonstrates extending a base template, using blocks, conditionals, loops, and filters to construct a user profile page with complex logic.

Conclusion

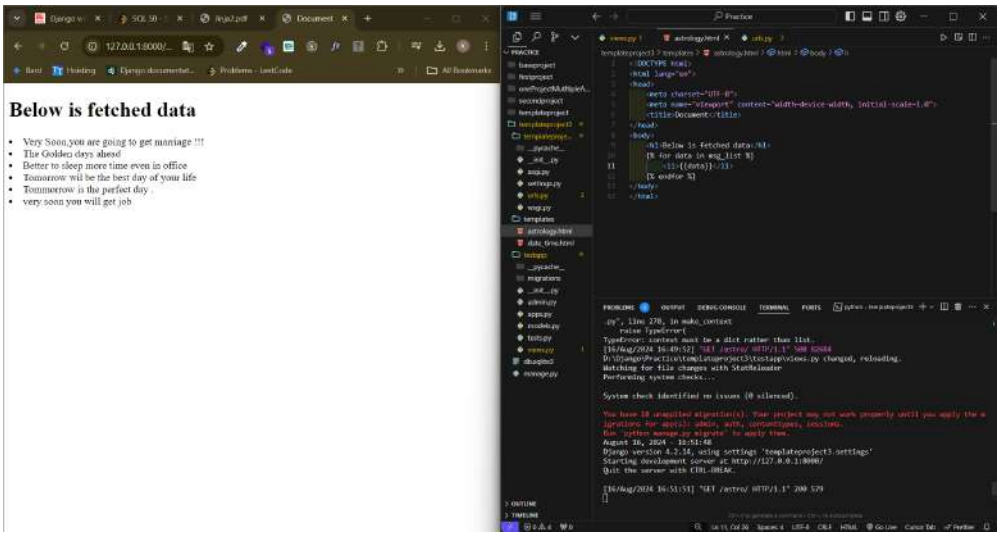
Jinja2 provides a rich set of features for creating dynamic and complex templates. By combining these features, you can implement virtually any logic needed for your application.

```
from django.shortcuts import render
import datetime
import random

# Create your views here.
def datetime_view(request):
    date=datetime.datetime.now()
    # my_dict={key:value}
    # my_dict={'msg':date}
    return render(request,'date_time.html',{'msg':date})


def result_view(request):
    msg_list=[
        'Very Soon,you are going to get marriage !!!',
        'The Golden days ahead',
        'Better to sleep more time even in office',
        'Tomorrow wil be the best day of your life',
        'Tommorrow is the perfect day .',
        'very soon you will get job'
    ]

    return render(request, 'astrology.html', {'msg_list': ms
```

```
<!DOCTYPE html>
<html lang="en">
<head>
```


Khushal Savalakha

```
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initi
    <title>Document</title>
</head>
<body>
    <h1>Below is fetched data</h1>
    {% for data in msg_list %}
        <li>{{data}}</li>
    {% endfor %}
</body>
</html>
```



wertyu

from django.shortcuts import render
import datetime

def datetime_view(request):
date=datetime.datetime.now()
# my_dict={key:value}
my_dict={'msg':date}
return render(request,'date_time.html',context=my_dict)

then when we have to write context explain when and why?

In Django, the `render` function is commonly used to combine a given template with a context dictionary and return an `HttpResponse` object with that rendered text. The `context`

Khushal Savalakha

parameter in the `render` function is optional, but it provides a clear and explicit way to pass data from your view to your template.

Understanding the `render` Function

The `render` function is defined as follows:

```
def render(request, template_name, context=None, content_
type=None, status=None, using=None)
```

Here's a breakdown of its parameters:

- `request`: The HTTP request object.
- `template_name`: The name of the template to use for rendering.
- `context`: A dictionary containing data to be passed to the template.
- `content_type`: The MIME type to use for the resulting document (optional).
- `status`: The status code for the response (optional).
- `using`: The name of the template engine to use (optional).

When to Use the `context` Parameter

- **Passing Data to Templates**: Use the `context` parameter when you need to pass dynamic data from your view to the template. This data is accessible within the template using the keys defined in the context dictionary.
- **Explicit vs. Implicit Context**: You can pass the context as a positional argument (i.e., without explicitly naming it) or as a keyword argument using `context=...`. Using `context=...` can make your code clearer, especially if you have multiple optional arguments like `content_type` or `status`.

Example 1: Using Context

```
from django.shortcuts import render
import datetime

def datetime_view(request):
    date = datetime.datetime.now()
    # Create a context dictionary
    my_dict = {'msg': date}
    # Pass the context dictionary using the 'context' key
```


Khushal Savalakha

```
word
    return render(request, 'date_time.html', context=my_d
ict)
```

In this example, the `context` parameter is used explicitly, making it clear what data is being passed to the template.

Example 2: Without Using `context` Keyword

```
from django.shortcuts import render
import datetime

def datetime_view(request):
    date = datetime.datetime.now()
    my_dict = {'msg': date}
    # Pass the context dictionary as a positional argumen
t
    return render(request, 'date_time.html', my_dict)
```

Here, the context dictionary is passed directly as a positional argument. This is perfectly valid, but if you add more optional parameters later, using the keyword argument can enhance clarity.

Why Use the `context` Keyword

1. **Readability**: Using `context=...` makes it clear that the data is being passed to the template as context, which improves code readability.
2. **Maintenance**: If you or someone else needs to maintain the code, explicitly naming the `context` argument reduces ambiguity and makes it easier to add other optional parameters in the future.
3. **Consistency**: Following a consistent pattern in your codebase helps avoid confusion, especially in larger projects where multiple developers might be working on the same codebase.

Summary

While the `context` parameter is optional, using it explicitly can enhance the readability and maintainability of your code. It's especially useful when you have multiple optional arguments to pass to the `render` function or when you want to make your intent clear in the code.

Khushal Savalakha

▼ ***Working With Static Files:***

Working with Static Files:

- Up to this just we injected normal text data into template by using template tags.
- But sometimes our requirement is to insert static files like images,css files etc inside template file.

Process to include Static Files inside Template:

1) Create a folder named with 'static' inside main project folder. It is exactly same as creating 'templates' folder.
2) In that 'static' folder create 'images' folder to place image files.
3) Add static directory path to settings.py file, so that Django can aware of our images.

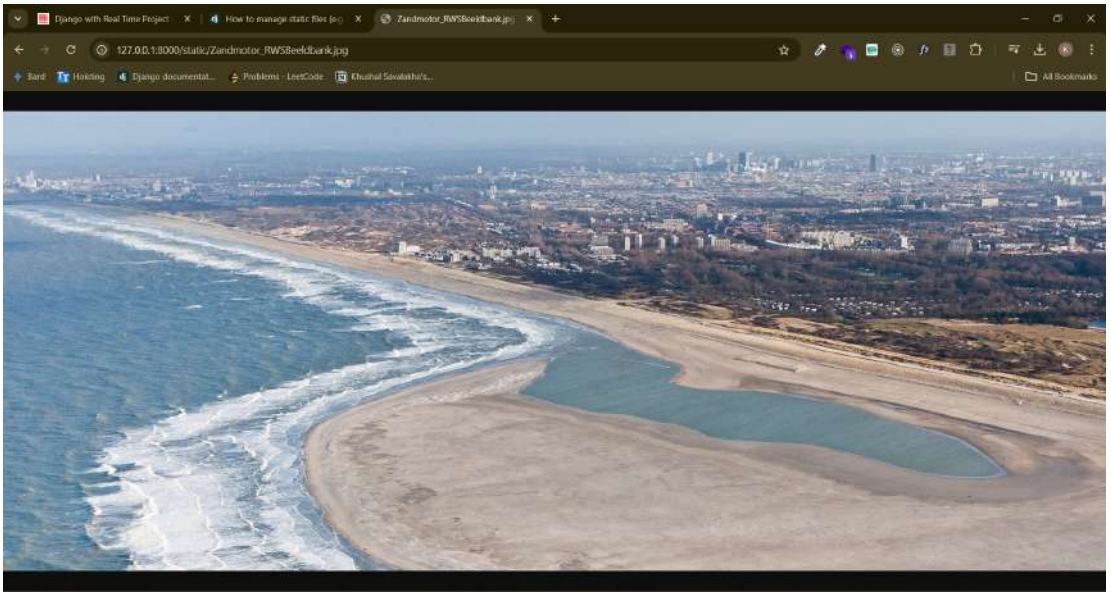
settings.py

```
1) STATIC_DIR=os.path.join(BASE_DIR,'static')
2)
3) ..
4) STATIC_URL = '/static/'
5)
6) STATICFILES_DIRS=[
7) STATIC_DIR,
8) ]
```

4) Make sure all paths are correct or not
   http://127.0.0.1:8000/static/images/divine3.jpg

5) Use Template Tags to insert image
   At the beginning of HTML just after <!DOCTYPE html> we have to include the following template tag {% load staticfiles %}

   Just we are conveying to the Django to load all static files.
   We have to include image file as follows <img src="{% static "images/divine3.jpg"% }">



Khushal Savalakha

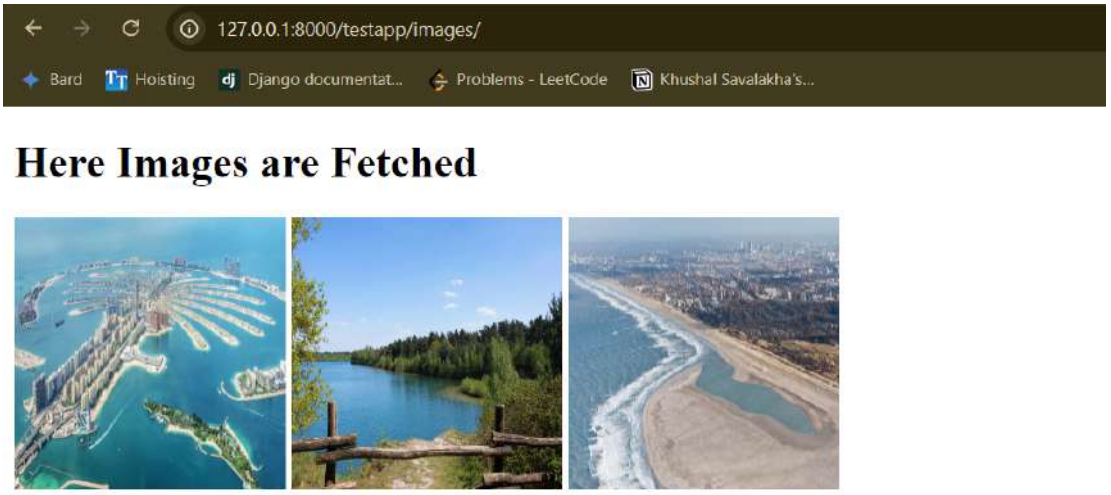
```
"""
URL configuration for staticfileproject project.

The `urlpatterns` list routes URLs to views. For more inf
    https://docs.djangoproject.com/en/5.0/topics/http/url
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, na
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view()
Including another URLconf
    1. Import the include() function: from django.urls im
    2. Add a URL to urlpatterns:  path('blog/', include('
"""
from django.contrib import admin
from django.urls import path,include
from testapp import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('testapp/',include('testapp.urls'))
]
```

```
from django.urls import path
from . import views

urlpatterns=[
    path('images/',views.show_images)
]
```

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
```


Khushal Savalakha

```
    <meta name="viewport" content="width=device-width, in
    <title>Static Files Demo</title>
</head>
<body>
    <h1>Here Images are Fetched</h1>
    <img src="{% static 'images/dubai_palm_jumeirah_islan
    <img src="{% static 'images/Natherland.jpg' %}" style
    <img src="{% static 'images/Zandmotor_RWSBeeldbank.jp
</body>
</html>
```



<u>HOW TO INCLUDE CSS FILES:</u>

```
from django.shortcuts import render

# Create your views here.
def show_images(request):
    return render(request,'show_image.html')


def index_views(request):
    return render(request,'index.html')

def movie_review(request):
    head_msg='Movies Information'
    sub_msg1='RRR ready to release'
```

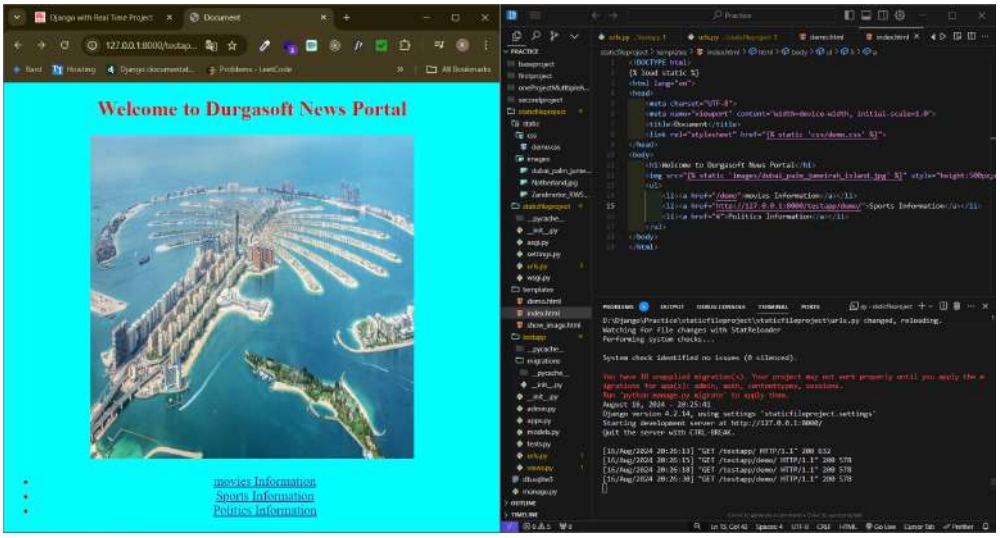
```
    sub_msg2='Background support must be required to get
    sub_msg3='Indian cine industry struggling like anythi
    sub_msg4='Biggest benefit of covid for industry:OTT'
    sub_msg5='Written by SKJ'
    my_dict={'head_msg':head_msg,'sub_msg1':sub_msg1,'sub
    return render(request,'demo.html',my_dict)
```

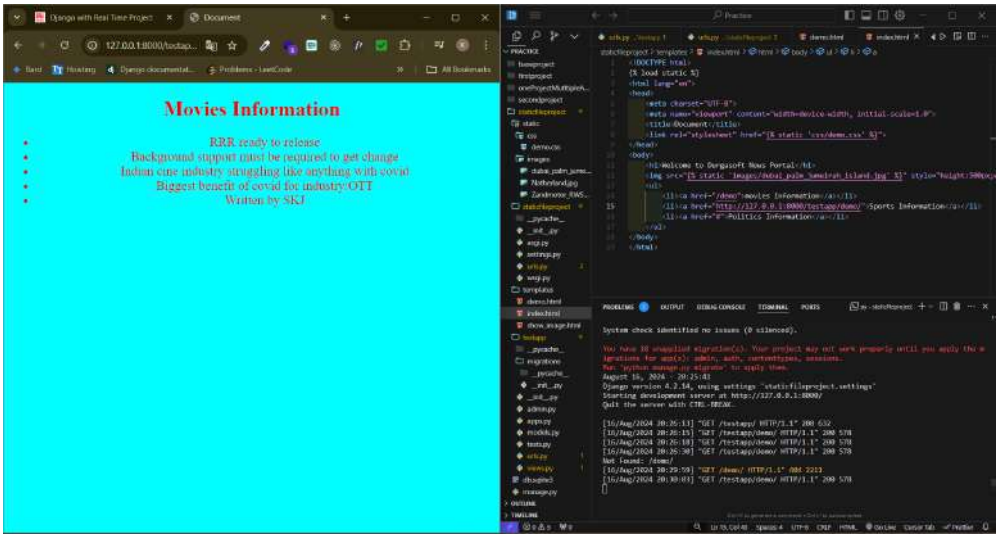
```
<!DOCTYPE html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, in
    <title>Document</title>
    <link rel="stylesheet" href="{% static 'css/demo.css'
</head>
<body>
    <h1>{{head_msg}}</h1>
    <ul>
        <li>{{sub_msg1}}</li>
        <li>{{sub_msg2}}</li>
        <li>{{sub_msg3}}</li>
        <li>{{sub_msg4}}</li>
        <li>{{sub_msg5}}</li>
    </ul>
</body>
</html>
```

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, in
    <title>Document</title>
    <link rel="stylesheet" href="{% static 'css/demo.css'
</head>
<body>
```


Khushal Savalakha

```
    <h1>Welcome to Durgasoft News Portal</h1>
    <img src="{% static 'images/dubai_palm_jumeirah_islar
    <ul>
        <li><a href="/demo">movies Information</a></li>
        <li><a href="http://127.0.0.1:8000/testapp/demo/'
        <li><a href="#">Politics Information</a></li>
    </ul>
</body>
</html>
```



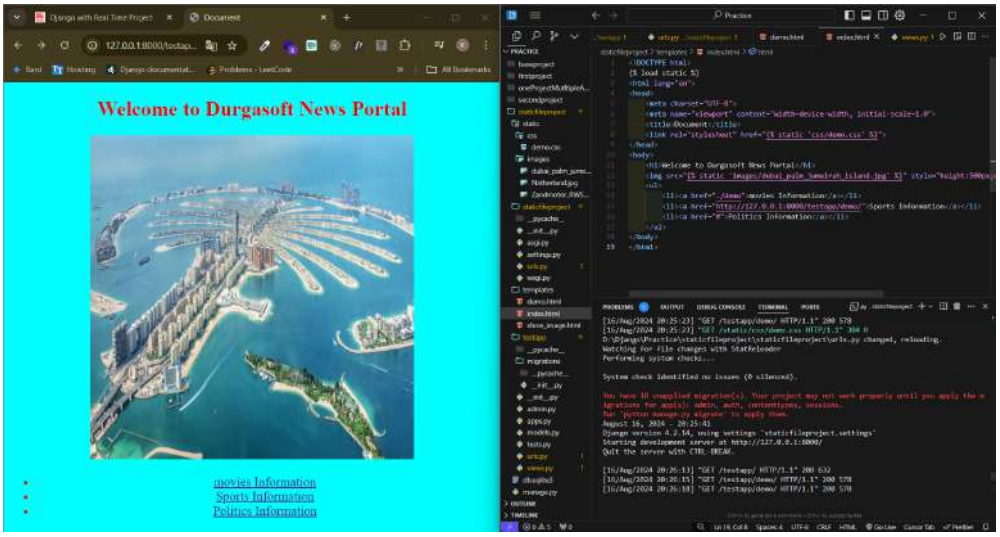


if we want output from application based url.

```
<!DOCTYPE html>
{% load static %}
```


Khushal Savalakha

```html
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, in
    <title>Document</title>
    <link rel="stylesheet" href="{% static 'css/demo.css'
</head>
<body>
    <h1>Welcome to Durgasoft News Portal</h1>
    <img src="{% static 'images/dubai_palm_jumeirah_islan
    <ul>
        <li><a href="./demo">movies Information</a></li>
        <li><a href="http://127.0.0.1:8000/testapp/demo/"
        <li><a href="#">Politics Information</a></li>
    </ul>
</body>
</html>
```



▼ Working with Models and Databases

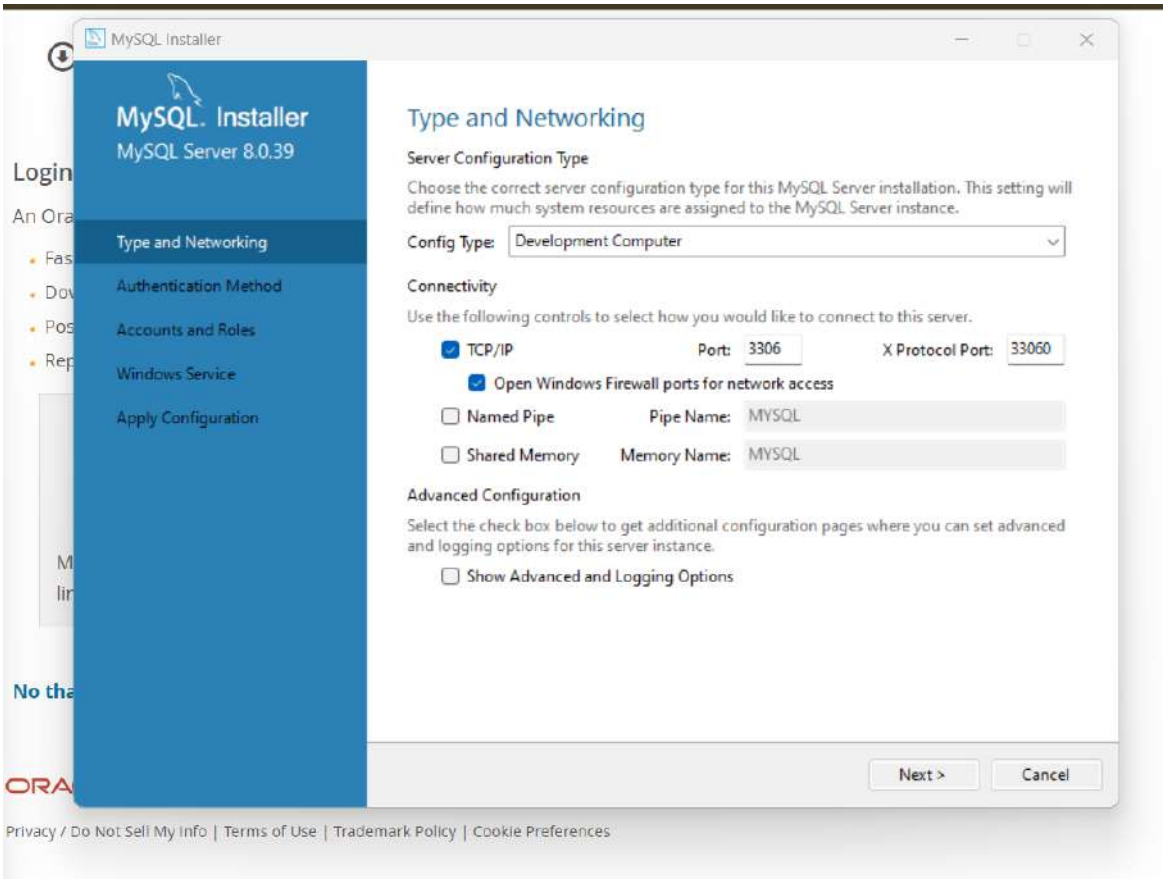
```
# Database
# https://docs.djangoproject.com/en/5.0/ref/settings/#databa

DATABASES = {
    'default': {
```

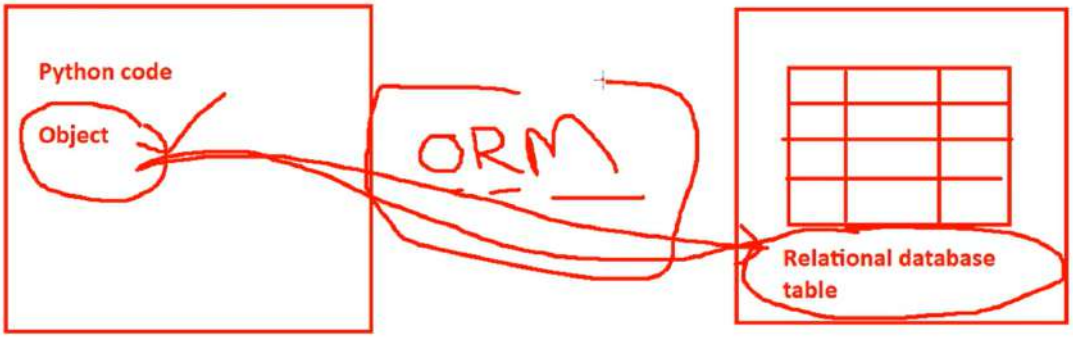
```
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

```
PS D:\Django\Practice> django-admin startproject modelprojec
PS D:\Django\Practice> cd modelproject
PS D:\Django\Practice\modelproject> py manage.py startapp te
PS D:\Django\Practice\modelproject> ^C
PS D:\Django\Practice\modelproject> cd ..
PS D:\Django\Practice> py manage.py shell
C:\Users\janak\AppData\Local\Programs\Python\Python312\pytho
PS D:\Django\Practice> cd modelproject
PS D:\Django\Practice\modelproject> py manage.py shell
Python 3.8.5 (default, Sep  3 2020, 21:29:08) [MSC v.1916 64
Type 'copyright', 'credits' or 'license' for more informatic
IPython 7.19.0 -- An enhanced Interactive Python. Type '?' f

In [1]: from django.db import connection

In [2]: c=connection.cursor()

In [3]: c
Out[3]: <django.db.backends.utils.CursorDebugWrapper at 0x20

In [4]:
```

- How to Check Django Database Connection: ☕ We can check whether django database configurations are properly configured or not by using the following commands from the shell ☕
- D:\djangoprojects\modelProject>py manage.py shell
- >>> from django.db import connection
- >>> c = connection.cursor()
- ☕ **If we are not getting any error means our database configurations are proper.**

Khushal Savalakha





(ORM)Object Relational Maping we just have to write object then object relational maping is happen .In java Hybernate is ORM frame work sam as Django it self as In built ORM.

```
If we don't want sqlite3, then we have to configure  our own database with the following parameters:

1. ENGINE: Name of database engine
2. NAME: DATABASE NAME
3. USER: Database login user name
4. PASSWORD: Database login password
5. HOST: The machine on which database server is running.
6. PORT: The port number on which database server is running.
```

```
1767 mysql: 3306
1768 oracle: 1521
1769 postgres 5432
1770 mongodb: 27017
1771 etc
1772
```

```
# DATABASES = {
#     'default': {
#         'ENGINE': 'django.db.backends.sqlite3',
#         'NAME': BASE_DIR / 'db.sqlite3',
#     }
# }

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',  # Postgr
        'NAME': 'Django_Trial',                     # Replace with
        'USER': 'postgres',
        'PASSWORD': 'khushal',
        'HOST': 'localhost',
        'PORT': '5432',                             # Defaul
    }
}
```

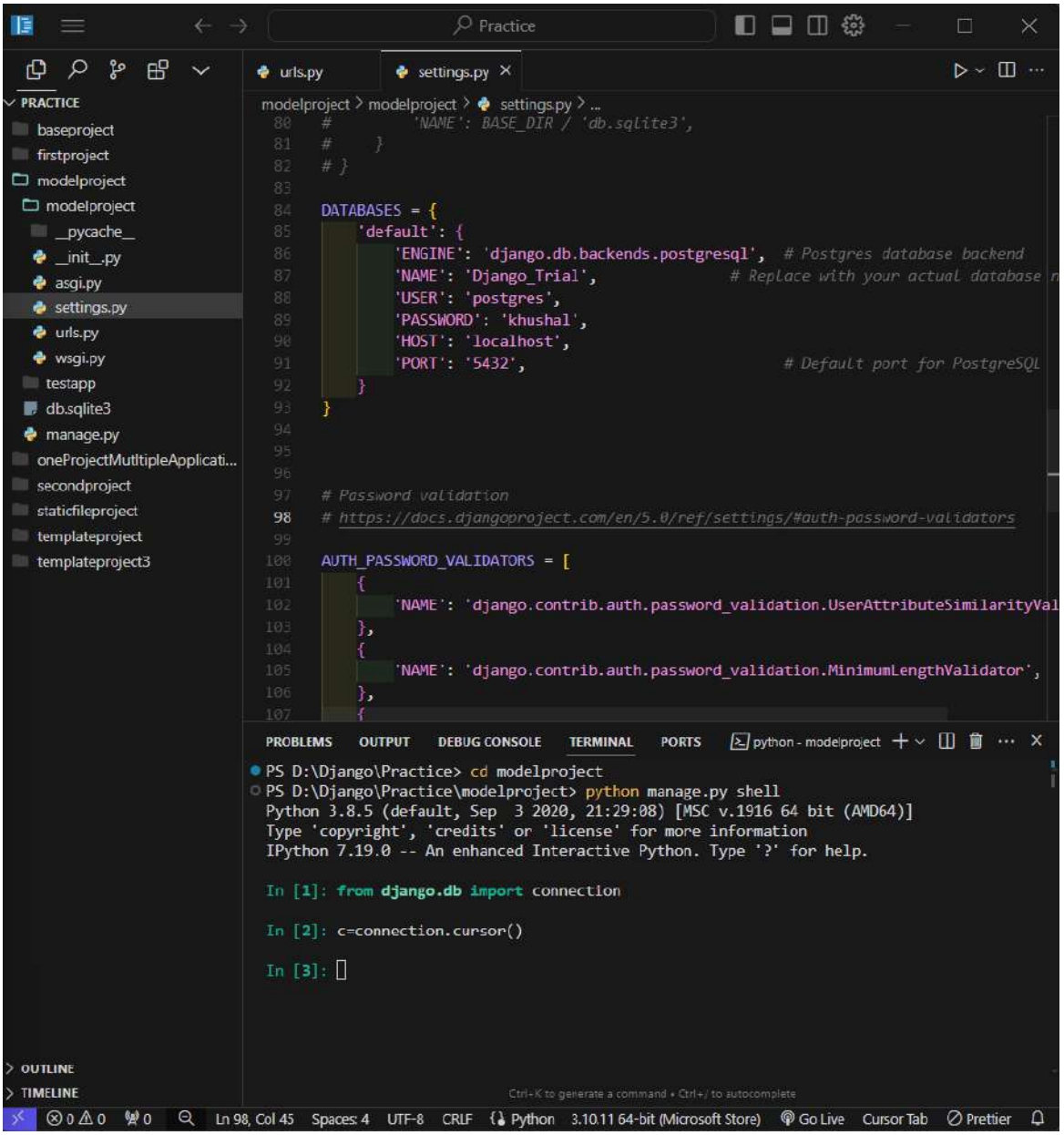
In django we have to write 1 python class this python class which is going to represent database table this python class by default consider as model class.which contains database information.Each model is going to represent or map to 1 databse table.

💡 Ex:If 2 tables are requires then how many model class are required ?

2 model classes are required

behaviour means which is primary key ,how many max character it contains

Khushal Savalakha




```
Model class:
------------
A model is a python class which contains database information.
Model contains fields and behaviour of data what we are storing.
Each model maps to one database table.
Every model is a python class which is the child class of django.db.models.Model class.
Each attribute of the model represents a database field(column name in table)
We have to write all model classes inside 'models.py' file.
```

```
from django.db import models

# Create your models here.
```


Khushal Savalakha

```
class Employee(models.Model):
    eno=models.IntegerField()
    ename=models.CharField(max_length=30)
    esal=models.FloatField(max_length=30)
    eaddr=models.CharField(max_length=30)

explain what is the meaning of behaviour in the image and al
```

Explanation of "Behaviour" in the Image

In the context of the image you shared, **"behavior"** refers to the **methods** and functions within a model class that define how the data (attributes or fields) of that model behave or are manipulated. For example, you can have methods in the model class that perform calculations, validate data, or modify data before saving it to the database.

The Role of `models.Model` in Django

The `models.Model` is a base class provided by Django that every model class should inherit from. When you define a class that inherits from `models.Model`, it allows you to create a blueprint for a database table. This class will then map to a table in the database, and each attribute in the class will represent a column in that table.

How `models.Model` Works with an Example

Let's break down the `Employee` model you've shared:

```
from django.db import models

class Employee(models.Model):
    eno = models.IntegerField()
    ename = models.CharField(max_length=30)
    esal = models.FloatField(max_length=30)
    eaddr = models.CharField(max_length=30)
```

1. `eno = models.IntegerField()`:
    - This creates a column in the database table named `eno` that stores integer values. It corresponds to the employee number.
2. `ename = models.CharField(max_length=30)`:

Khushal Savalakha

- This creates a column `ename` that stores string values (character fields) with a maximum length of 30 characters. It corresponds to the employee's name.

3. `esal = models.FloatField(max_length=30)`:
   - This creates a column `esal` that stores floating-point numbers, which is used to store the employee's salary.
4. `eaddr = models.CharField(max_length=30)`:
   - This creates a column `eaddr` that stores the employee's address as a string, with a maximum length of 30 characters.

Summary of `models.Model` Work

- **Database Interaction:** Django models automatically interact with the database for you. When you save a model instance, Django translates it into SQL and stores the data in the corresponding table.
- **ORM (Object-Relational Mapping):** By inheriting from `models.Model`, your class becomes part of Django's ORM, allowing you to interact with database records using Python objects instead of writing raw SQL queries.
- **Auto-generated Methods:** Inheriting from `models.Model` gives you various methods like `save()`, `delete()`, and more, which makes it easy to perform CRUD (Create, Read, Update, Delete) operations on your data.

In summary, `models.Model` is the backbone of Django's ORM system, linking your Python code to the database in a way that allows you to work with your data using simple Python classes.

💡 explain here what is the meaning of Instance and how work flow is going i don't understand explain what is the reason behind model.Model use?

What is an Instance in Django?

An **instance** in Django refers to an **object** created from a model class. When you define a model class, such as `Employee`, it acts as a blueprint for creating instances (or objects) that represent rows in the corresponding database table.

Example of Creating an Instance

Given your `Employee` model:

Khushal Savalakha

```
from django.db import models

class Employee(models.Model):
    eno = models.IntegerField()
    ename = models.CharField(max_length=30)
    esal = models.FloatField()
    eaddr = models.CharField(max_length=30)
```

You can create an instance of `Employee` like this:

```
emp1 = Employee(eno=1, ename="John Doe", esal=50000.00, e
addr="123 Main St")
```

Here, `emp1` is an **instance** of the `Employee` model, representing one row in the `Employee` table in the database. This instance has specific values for `eno`, `ename`, `esal`, and `eaddr`.

Workflow of Instances in Django

1. **Creation**:
   - You create an instance using the model class. The instance holds the data that you want to save in the database.
2. **Saving**:
   - When you call the `save()` method on the instance, Django translates it into an SQL `INSERT` or `UPDATE` statement and saves the data to the corresponding table in the database.

   ```
   emp1.save()
   ```

3. **Retrieval**:
   - You can retrieve instances from the database using Django's query methods. For example, to retrieve all employees:

   ```
   employees = Employee.objects.all()
   ```

4. **Modification**:
   - You can modify the fields of an instance and then call `save()` again to update the corresponding row in the database.

Khushal Savalakha

```
emp1.ename = "Jane Doe"
emp1.save()
```

5. **Deletion**:
   - You can delete an instance, which will remove the corresponding row from the database.

```
emp1.delete()
```

Why Use `models.Model`?

The `models.Model` class is central to Django's ORM (Object-Relational Mapping) system, and there are several reasons for using it:

1. **Database Abstraction**:
   - By inheriting from `models.Model`, you abstract away the complexities of SQL. You work with Python objects rather than writing raw SQL queries, making database interactions easier and more intuitive.
2. **Automatic Table Creation**:
   - When you define a model class and run migrations, Django automatically creates a corresponding table in the database. This table's columns are based on the fields defined in your model.
3. **CRUD Operations**:
   - `models.Model` provides built-in methods for creating, retrieving, updating, and deleting records in the database, making data management straightforward.
4. **Validation**:
   - Fields in a model can have validations (e.g., `max_length` for `CharField`), ensuring data integrity before it is saved to the database.
5. **Relationships**:
   - Django models can easily define relationships between tables using fields like `ForeignKey`, `ManyToManyField`, and `OneToOneField`, which would otherwise require complex SQL queries.

Summary

Khushal Savalakha

- **Instance**: An object created from a Django model class that represents a single row in the corresponding database table.
- `models.Model`: The base class that allows your model to interact with the database. It simplifies database operations, manages data validation, and automatically handles the creation of database tables and relationships between them.

In essence, `models.Model` is the foundation of Django's ORM, allowing developers to work with databases in a more Pythonic and abstracted way, making it easier to build, maintain, and scale applications.

> 💡 means we are defining what type of table structure are there and it will save data which has match data type in backend it's like mongoose in which we define data type and column so here model fill data like defined structure in function and save it and load it in database am i true ?

Yes, you're on the right track! Django's `models.Model` is very similar to how Mongoose works with MongoDB, especially in the way it defines the structure of data.

Here's a Breakdown:

- **Table Structure Definition**:
  - In Django, when you define a model class (e.g., `Employee`), you're essentially defining the structure of a table in the database. Each field in the model (like `eno`, `ename`, etc.) corresponds to a column in the database table.
- **Data Type Validation**:
  - Django checks that the data you're trying to save matches the field types you defined. For instance, if you defined `eno` as an `IntegerField`, Django will ensure that only integer values can be saved in this field, similar to how Mongoose enforces schema types.
- **Saving Data**:
  - When you create an instance of your model and call `save()`, Django stores the data in the database, ensuring that the data adheres to the structure you've defined in your model.
- **Loading Data**:
  - When you retrieve data from the database, Django uses the model to structure that data back into Python objects, making it easy to work with in your application.

Khushal Savalakha

Comparing with Mongoose:

- **Mongoose**:
  - You define schemas that describe the structure of documents in a MongoDB collection. Each field in the schema has a type, and Mongoose enforces these types when saving documents to the database.
- **Django Models**:
  - Similarly, you define models that describe the structure of tables in a relational database. Each field in the model has a type (like `CharField`, `IntegerField`), and Django enforces these types when saving rows to the database.

Conclusion:

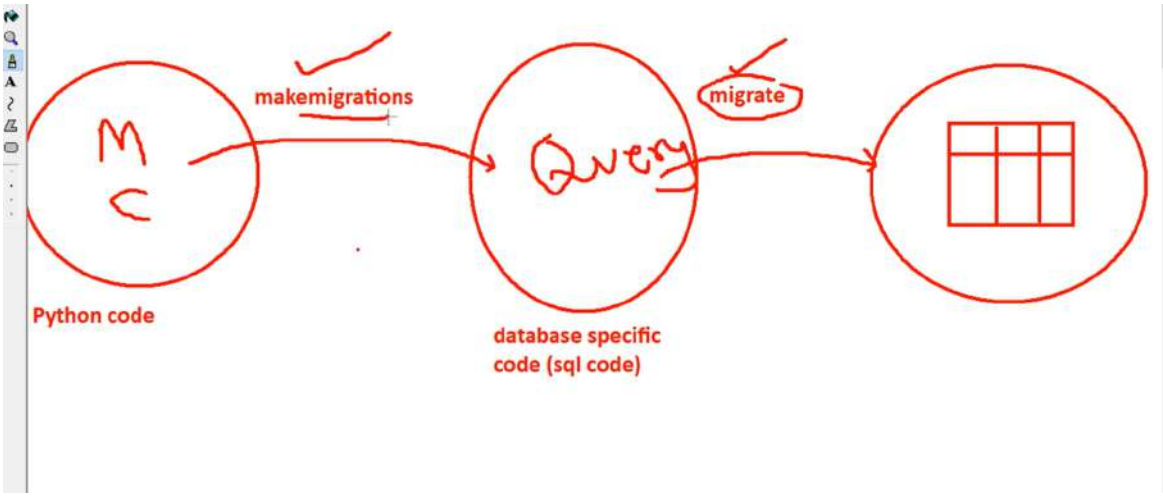
- Both **Django models** and **Mongoose schemas** serve the purpose of defining the structure of the data you'll store, validating that data, and facilitating the saving and retrieval of that data from the database. The key difference is that Django is typically used with relational databases (like PostgreSQL, MySQL), whereas Mongoose is used with MongoDB, a NoSQL database.

So yes, your understanding is correct! Django models define the structure of the data, enforce data types, and handle interactions with the database, much like Mongoose does for MongoDB.

```
# Create your models here.
class Employee(models.Model):
    eno = models.IntegerField()
    ename = models.CharField(max_length=30)
    esal = models.FloatField()
    eaddr = models.CharField(max_length=30)

Note: This model class will be converted into database table. Django is responsible for this conversion.

table name:appname_modelclassname
          :testapp_employee
```

```
table name:appname_modelclassname
          :testapp_employee
fields: eno,ename,esal and eaddr. And one extra field: id
behaviour: eno is of integer type,ename is of string type and max_length is 30 characters etc


Model class = database table name+field names+field behaviours
```


Khushal Savalakha



makemigrations:Convert into database specific Query Form.

migrate:Execute the Querry.

- **makemigrations**:

  Convert Django models into database-specific SQL queries.

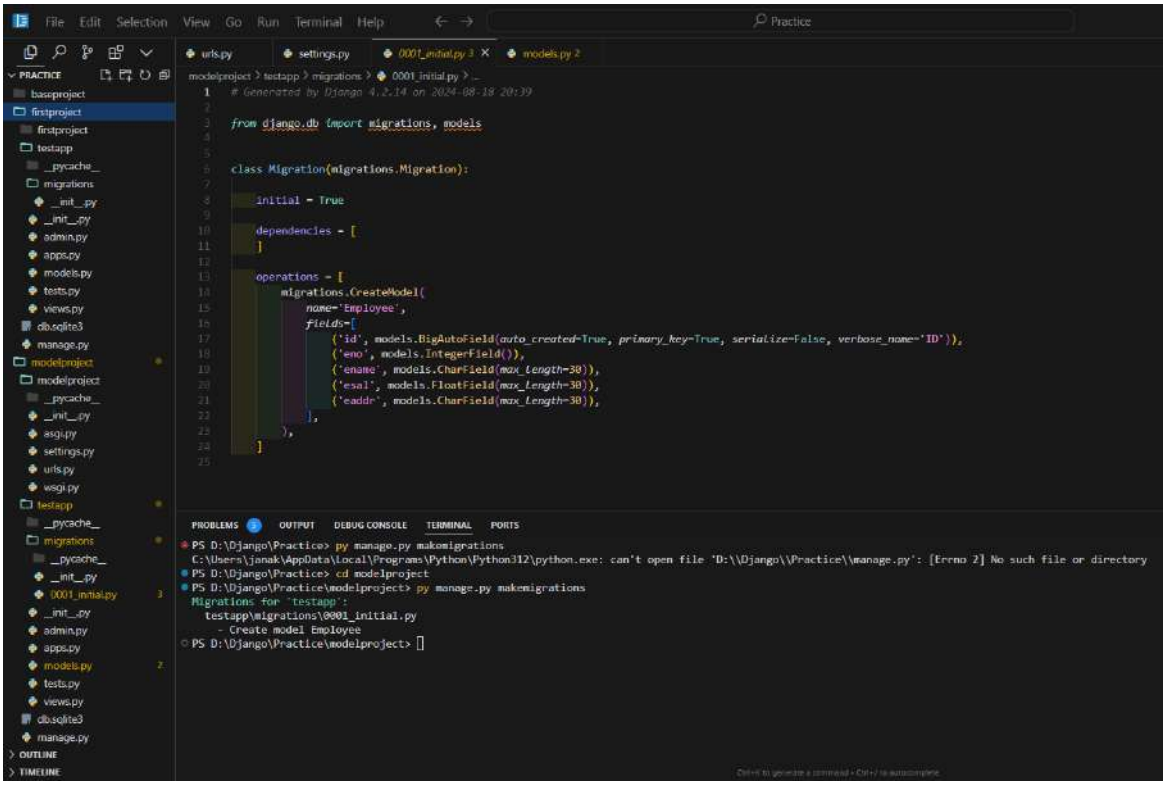
  This command scans your Django models and generates SQL statements based on the changes. It creates migration files containing database-specific queries like `CREATE TABLE`, `ALTER TABLE`, etc.

- **migrate**:

  Execute the SQL queries on the database.

  This command applies the migrations (SQL queries) generated by `makemigrations` to the actual database, modifying the schema accordingly (e.g., creating tables, adding columns).

This way, `makemigrations` is like creating the blueprint, and `migrate` is executing that blueprint on your database.



💡 Migrations and Migrate

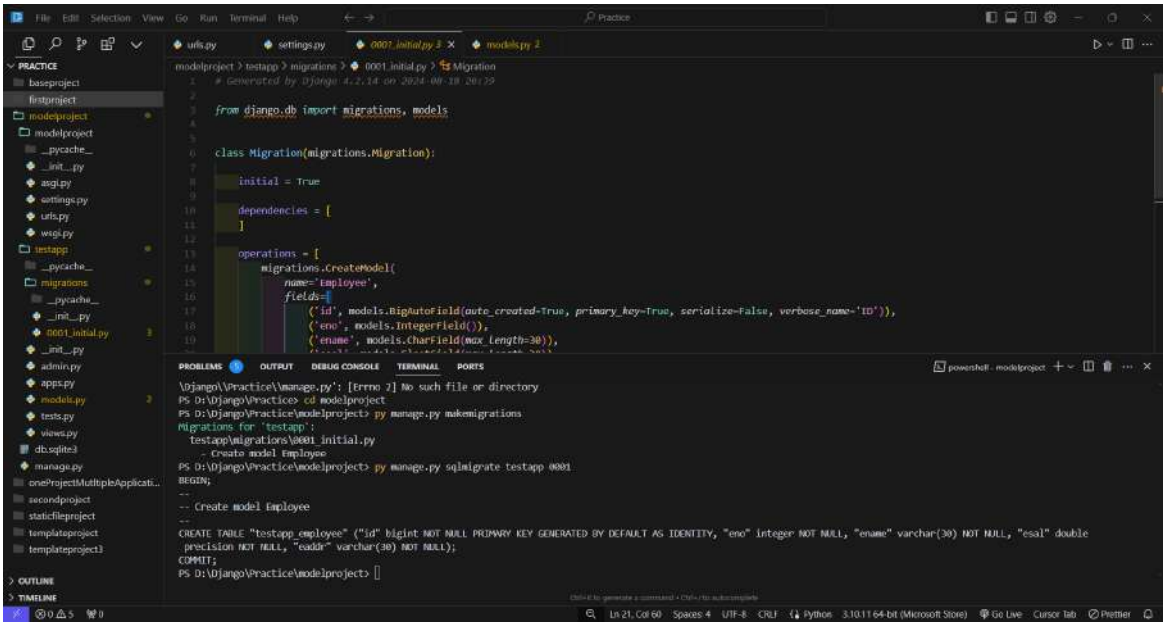
```
PS D:\Django\Practice\modelproject3> python manage.py help

Type 'manage.py help <subcommand>' for help on a specific su

Available subcommands:

[auth]
    changepassword
    createsuperuser

[contenttypes]
    remove_stale_contenttypes

[django]
    check
    compilemessages
    createcachetable
```

```
    dbshell
    diffsettings
    dumpdata
    flush
    inspectdb
    loaddata
    makemessages
    makemigrations
    migrate
    optimizemigration
    sendtestemail
    shell
    showmigrations
    sqlflush
    sqlmigrate
    sqlsequencereset
    squashmigrations
    startapp
    startproject
    test
    testserver

[sessions]
    clearsessions

[staticfiles]
    collectstatic
    findstatic
    runserver
PS D:\Django\Practice\modelproject3>
```

```
PS D:\Django\Practice> cd modelproject
PS D:\Django\Practice\modelproject> py manage.py makemigrati
Migrations for 'testapp':
  testapp\migrations\0001_initial.py
    - Create model Employee
PS D:\Django\Practice\modelproject> py manage.py sqlmigrate
BEGIN;
```


Khushal Savalakha

```
--
-- Create model Employee
--
CREATE TABLE "testapp_employee" ("id" bigint NOT NULL PRIMAR
COMMIT;
PS D:\Django\Practice\modelproject>
```

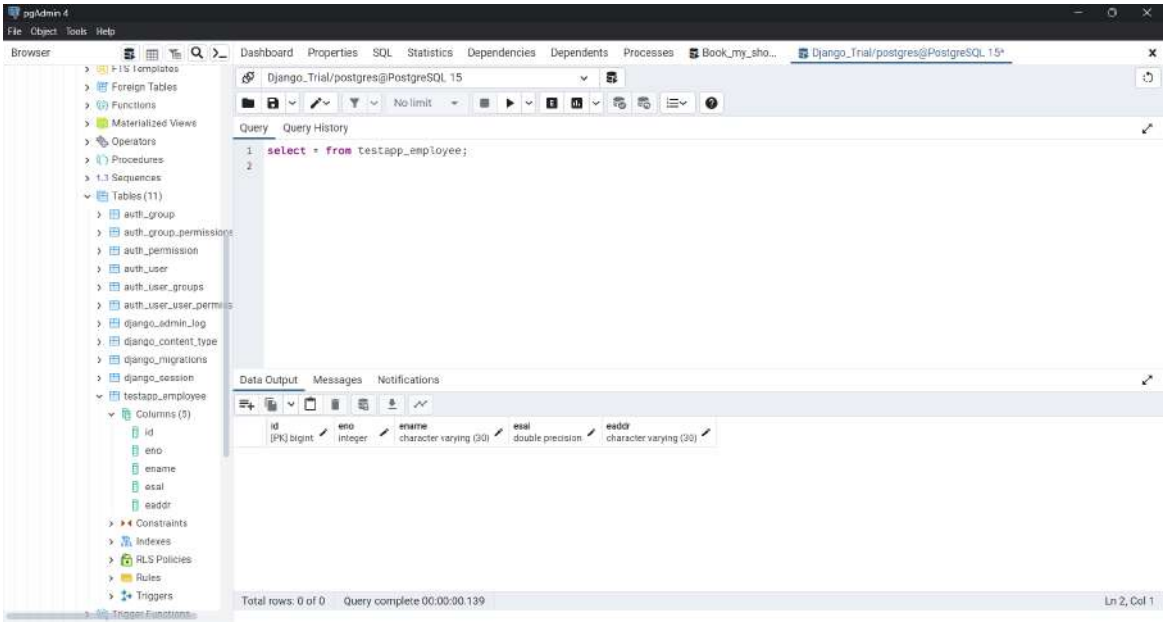


here we just convert data into specific from still we have to execute the query.

> 💡 CREATE TABLE "testapp_employee" ("id" bigint NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY, "eno" integer NOT NULL, "ename" varchar(30) NOT NULL, "esal" double precision NOT NULL, "eaddr" varchar(30) NOT NULL);
> COMMIT;

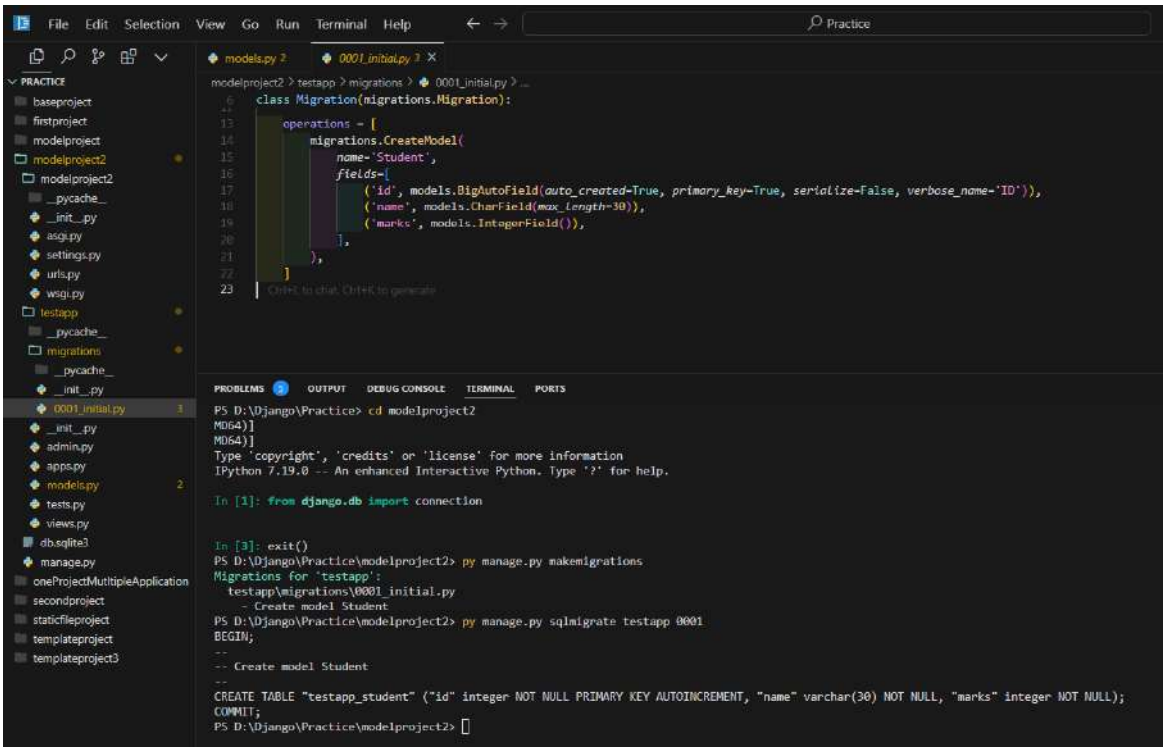
```
'id' field:
----------
1. For every table django will generate a special column named with id.
2. id is Primary key.(unique value for every record)
3. It is auto increment field. While inserting data we are not required to provide value for this field.
4. This field is of type: BigAutoField
5. We can override behaviour of id field and we can make our own field as id.
6. Every field is bydefault 'Not NULL'
```

```
PS D:\Django\Practice\modelproject> py manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions,
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages...
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
  Applying testapp.0001_initial... OK
PS D:\Django\Practice\modelproject>
```



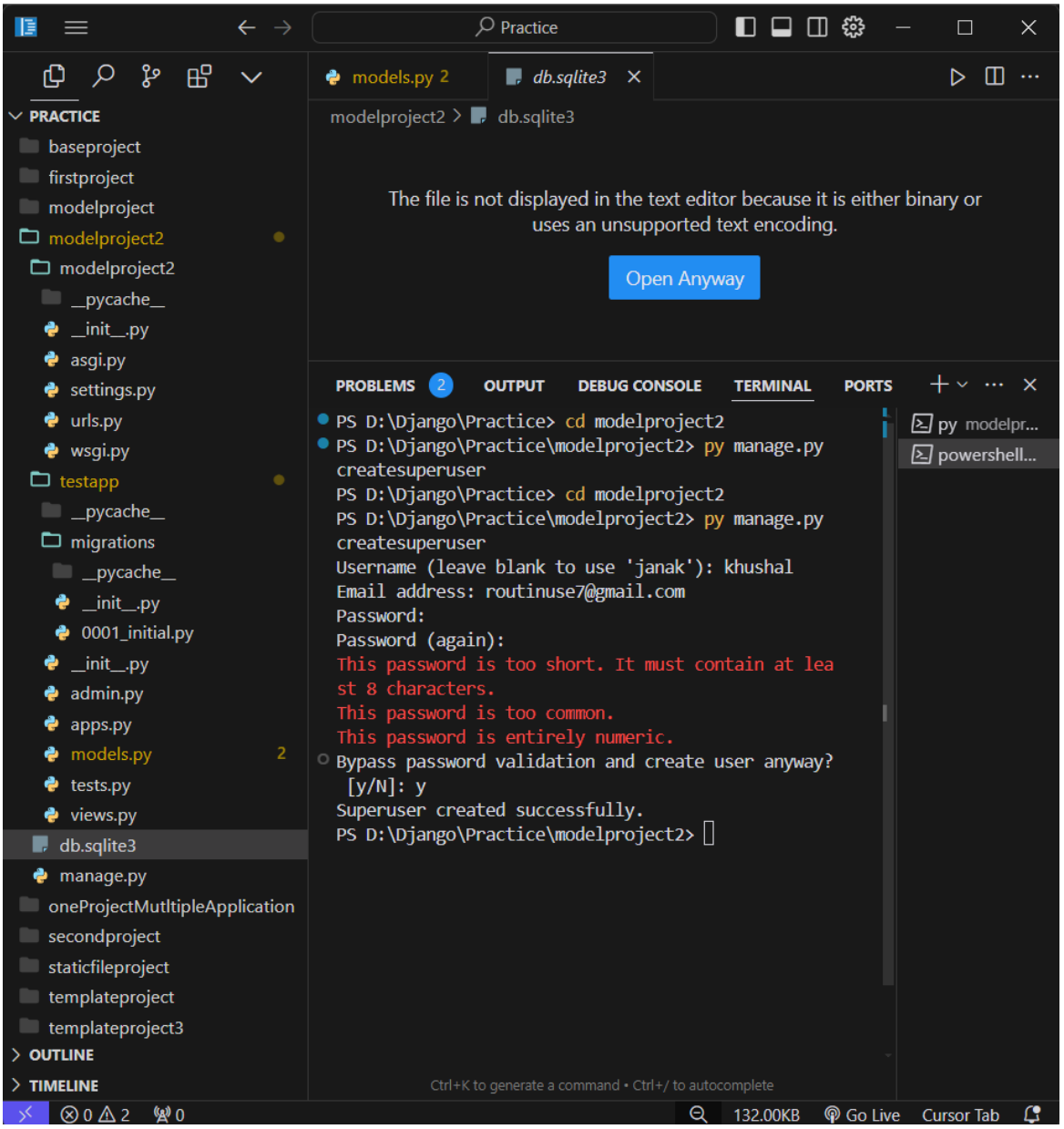
Khushal Savalakha

```
Summary:
--------
1. Perform database configurations inside settings.py
2. write model class inside models.py
3. py manage.py makemigrations
4. py manage.py migrate
```




```
PS D:\Django\Practice> cd modelproject2
PS D:\Django\Practice\modelproject2> py manage.py createsupe
Username (leave blank to use 'janak'): khushal
Email address: routinuse7@gmail.com
Password:
Password (again):
This password is too short. It must contain at least 8 chara
This password is too common.
This password is entirely numeric.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
PS D:\Django\Practice\modelproject2>
```

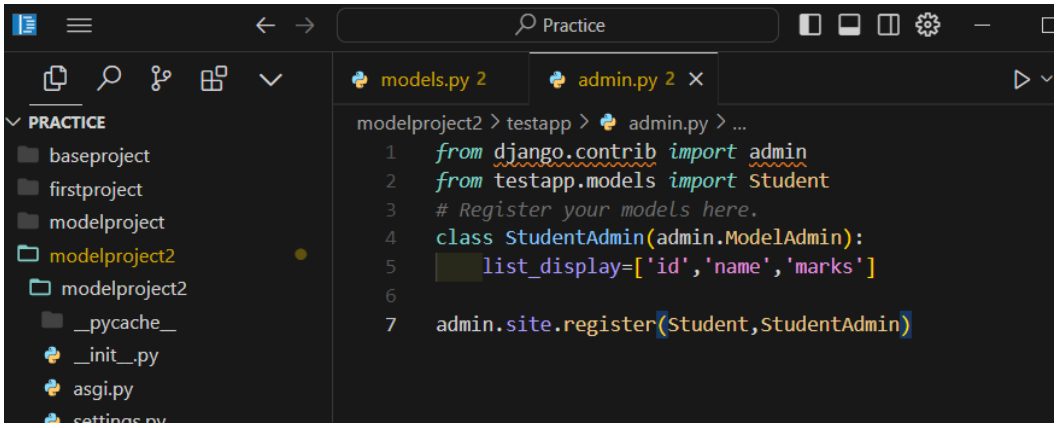

Khushal Savalakha

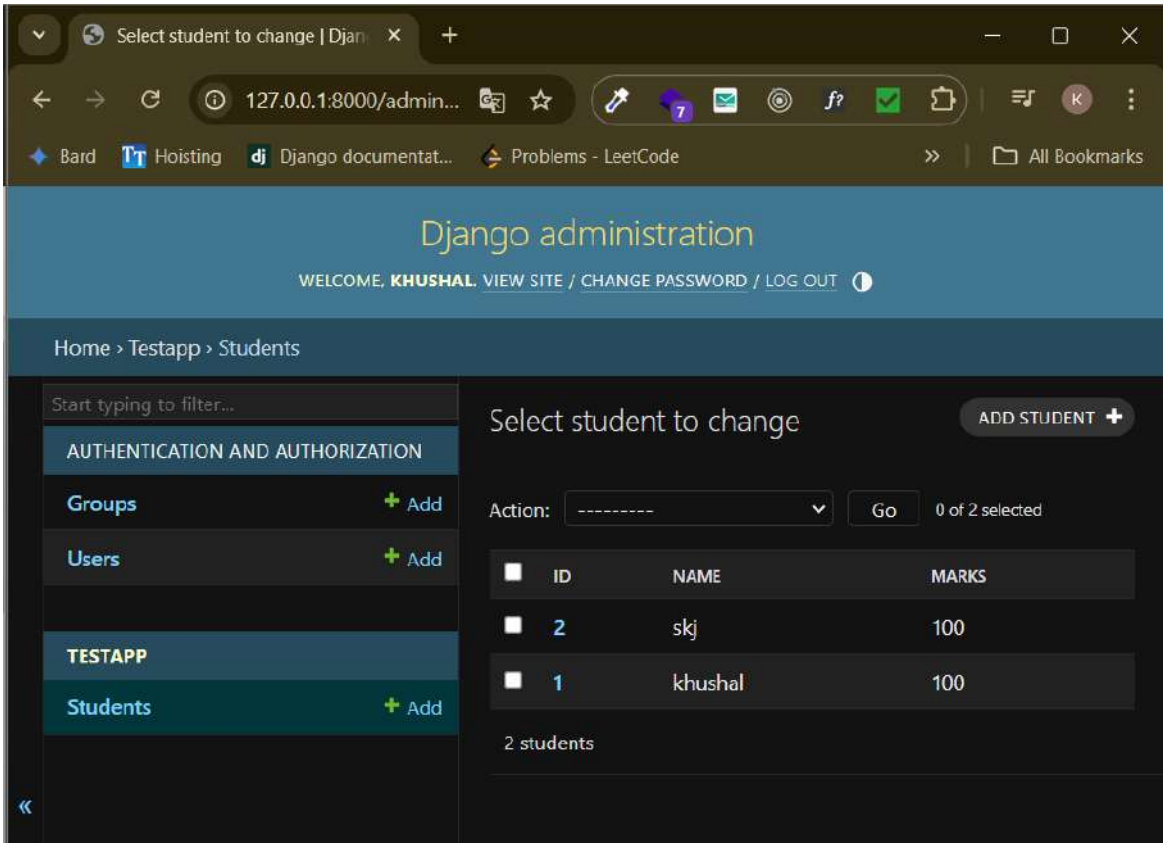



```
Email address: durgasoftonline@gmail.com
Password:
Password (again):
The password is too similar to the username.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.

***Note: By default our created tables are not visible in admin interface.
We have to register model inside admin interface then only it is visible.
We have to do registration inside admin.py file

admin.py:
---------
from django.contrib import admin
from testapp.models import Student
# Register your models here.
class StudentAdmin(admin.ModelAdmin):
    list_display = ['name','marks']

admin.site.register(Student,StudentAdmin)
```

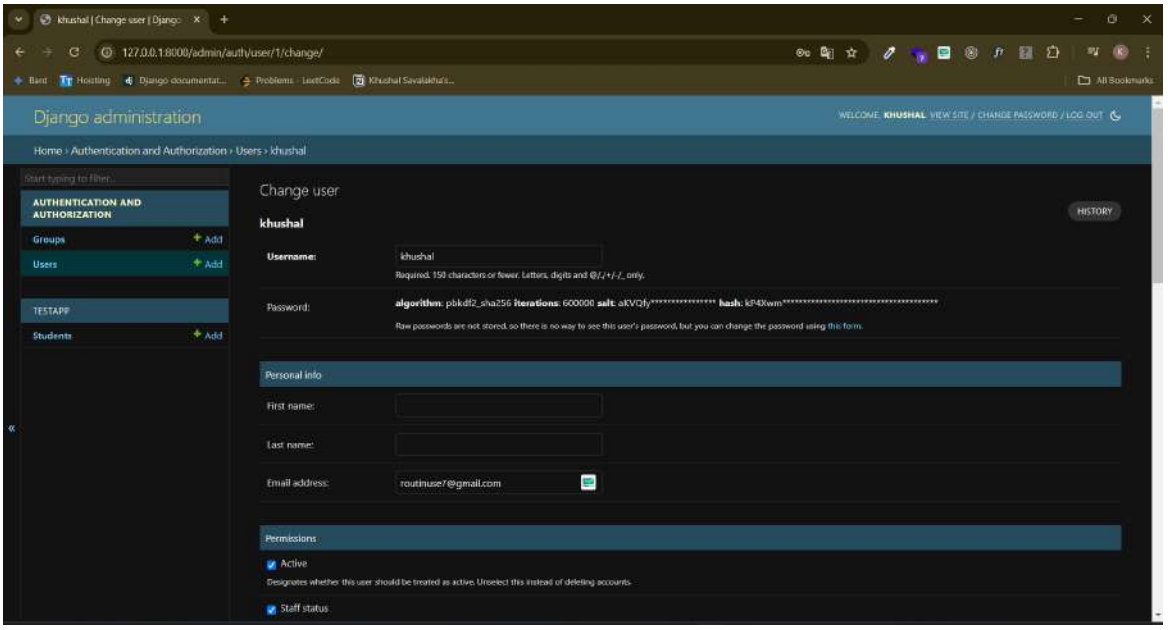
```
from django.contrib import admin
from testapp.models import Student
# Register your models here.
class StudentAdmin(admin.ModelAdmin):
    list_display=['id','name','marks']

admin.site.register(Student,StudentAdmin)
```

here list_display is predifined.





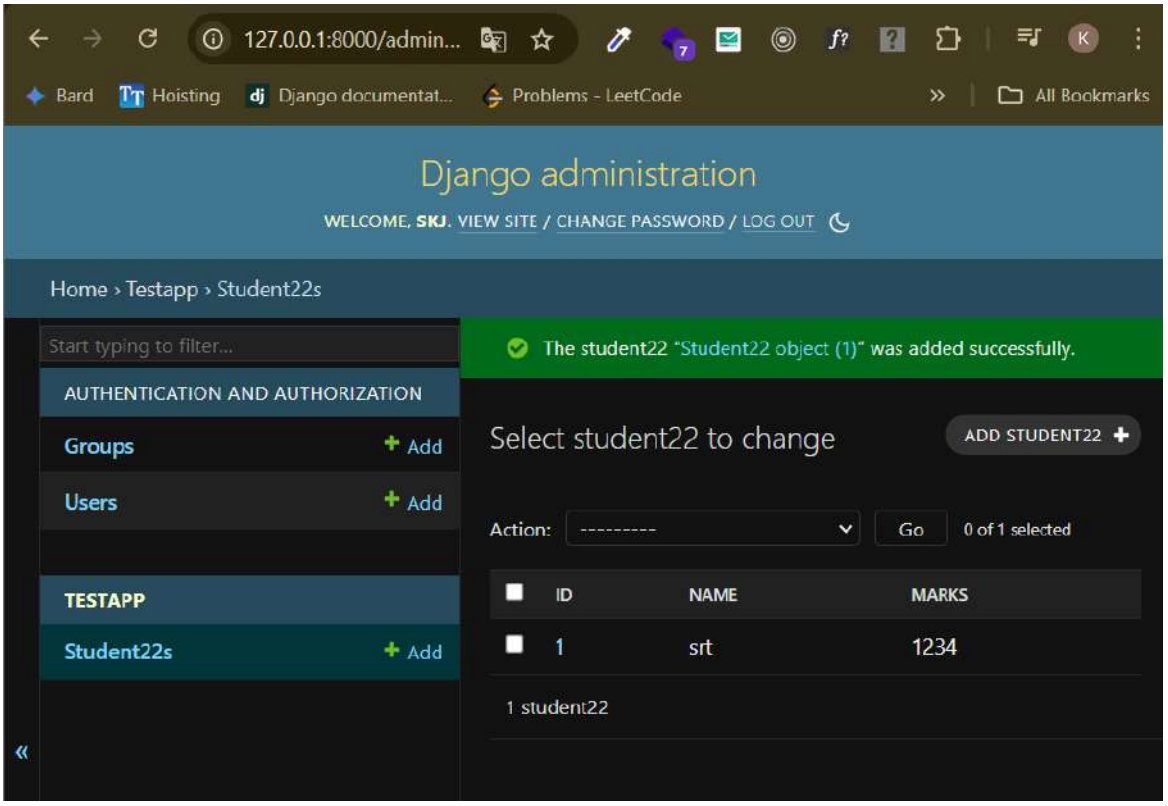
Khushal Savalakha

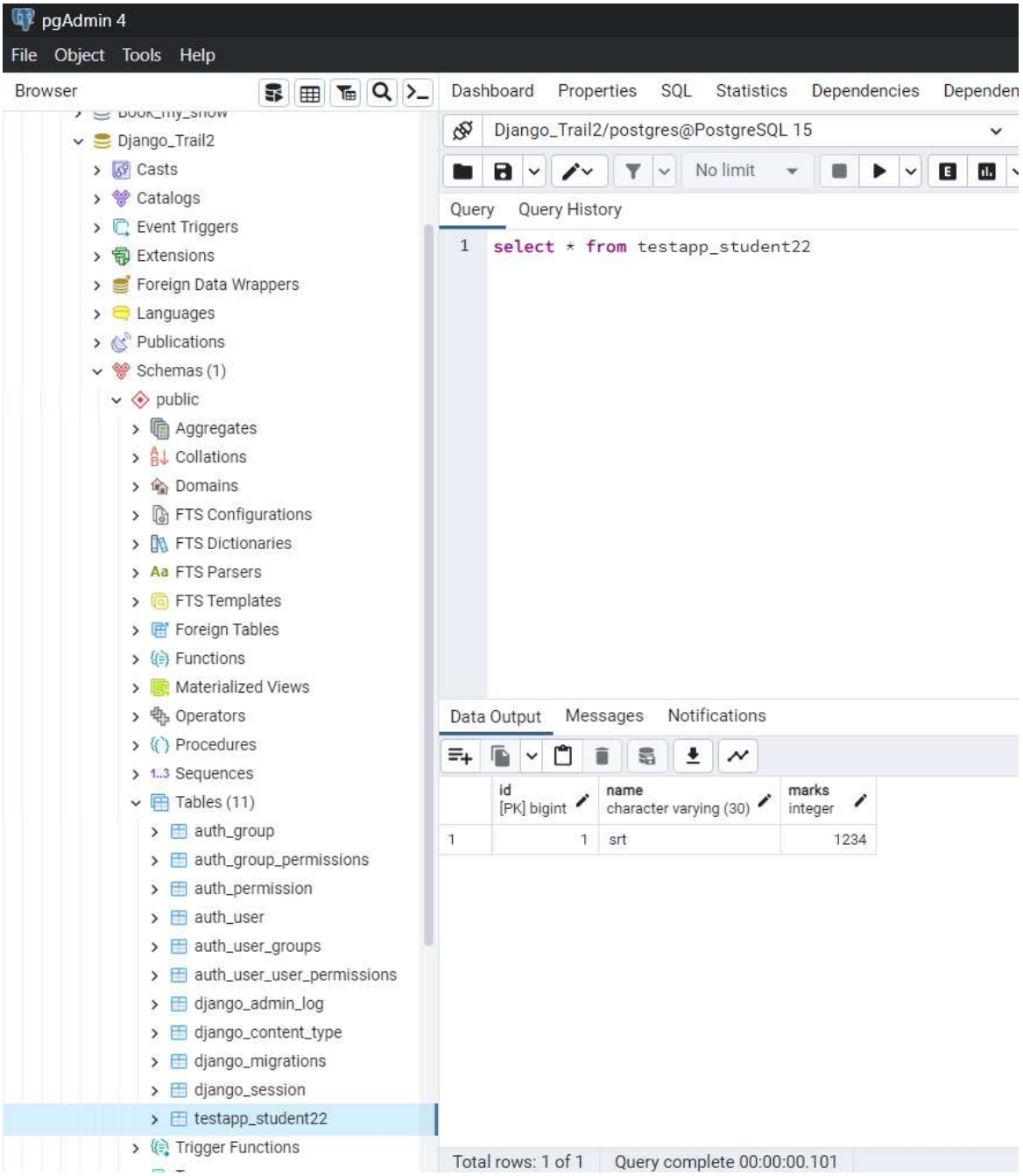



```
---------
from django.contrib import admin
from testapp.models import Student
# Register your models here.
class StudentAdmin(admin.ModelAdmin):
    list_display = ['name','marks']

admin.site.register(Student,StudentAdmin)

Note:
For every Model class we have to define separate Admin class. In that admin class
we have to specify which fields(columns) should required to display in admin
interface. Here we use list_display predefined variable.
```

Here we use list_display predefined varible .We have to register every Model and corresponding ModelAdmin class in admin.site.

```
PS D:\Django\Practice\modelproject3> py manage.py createsupe
Username (leave blank to use 'janak'): skj
Email address: skj@gmail.com
Password:
Password (again):
This password is too short. It must contain at least 8 chara
This password is too common.
This password is entirely numeric.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
PS D:\Django\Practice\modelproject3> py manage.py runserver
Watching for file changes with StatReloader
Performing system checks...
```


127.0.0.1:8000/admin...
Bard
Hoisting
Django documentat...
Problems - LeetCode
All Bookmarks
Django administration
WELCOME, SKJ. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Testapp › Student22s
Start typing to filter...
AUTHENTICATION AND AUTHORIZATION
Groups + Add
Users + Add
TESTAPP
Student22s + Add
The student22 "Student22 object (1)" was added successfully.
Select student22 to change
ADD STUDENT22 +
Action: ---------
Go
0 of 1 selected
ID
NAME
MARKS
1
srt
1234
1 student22



Khushal Savalakha

```
2143 sqllite3---->no configurations are required.
2144 mysql---->
2145   pip install mysqlclient
2146 oracle
2147   pip install cx_oracle
2148
2149 Read data from the database and display for the end user???
2150 1. startproject
2151 2. startapp
2152 3. add app in settings.py
2153 4. add database configurations in settings.py
2154 5. test database connection
2155 6. create model
2156 7. makemigrations and migrate
2157 8. register model and modeladmin inside admin.py
2158 9. createsuperuser
```

```
1  from django.db import models
2
3  # Create your models here.
4  class Employee(models.Model):
5      eno = models.IntegerField()
6      ename = models.CharField(max_length=30)
7      esal = models.FloatField()
8      eaddr = models.IntegerField()
9
```

💡 write all data like which types of field are there in Django how to define write all type of combinations.

In Django, models are used to define the structure of your database tables. Each attribute of a model represents a database field. Django provides a wide variety of field types, each corresponding to a different type of data. Below is an explanation of the common field types and some of their combinations.

Common Field Types in Django Models:

1. **IntegerField**: Stores integer values.

Khushal Savalakha

```
eno = models.IntegerField()
```

2. **CharField**: Stores strings with a maximum length.

```
ename = models.CharField(max_length=30)
```

3. **FloatField**: Stores floating-point numbers.

```
esal = models.FloatField()
```

4. **BooleanField**: Stores a True/False value.

```
is_active = models.BooleanField(default=True)
```

5. **DateField**: Stores date values.

```
hire_date = models.DateField()
```

6. **DateTimeField**: Stores date and time values.

```
created_at = models.DateTimeField(auto_now_add=True)
```

7. **EmailField**: Stores email addresses.

```
email = models.EmailField(max_length=254)
```

8. **TextField**: Stores large text data.

```
description = models.TextField()
```

9. **DecimalField**: Stores decimal numbers.

```
price = models.DecimalField(max_digits=10, decimal_pla
ces=2)
```

10. **FileField**: Handles file uploads.

Khushal Savalakha

```
file = models.FileField(upload_to='uploads/')
```

11. **ImageField**: Handles image uploads.

```
image = models.ImageField(upload_to='images/')
```

12. **URLField**: Stores URL strings.

```
website = models.URLField()
```

13. **ForeignKey**: Defines a many-to-one relationship.

```
department = models.ForeignKey(Department, on_delete=m
odels.CASCADE)
```

14. **ManyToManyField**: Defines a many-to-many relationship.

```
projects = models.ManyToManyField(Project)
```

15. **OneToOneField**: Defines a one-to-one relationship.

```
profile = models.OneToOneField(Profile, on_delete=mode
ls.CASCADE)
```

Examples of Field Combinations:

1. **Nullable Field**: A field that can have `NULL` values in the database.

```
middle_name = models.CharField(max_length=30, null=Tru
e)
```

2. **Unique Field**: A field that must have unique values across the table.

```
email = models.EmailField(unique=True)
```

3. **Default Value**: Assigns a default value to a field.

Khushal Savalakha

```
is_active = models.BooleanField(default=True)
```

4. **Auto-generated Fields**: Fields that are automatically generated.
   - `auto_now_add` sets the field to the current date/time when the object is first created.
   - `auto_now` updates the field to the current date/time every time the object is saved.

```
created_at = models.DateTimeField(auto_now_add=True)
updated_at = models.DateTimeField(auto_now=True)
```

5. **Choices**: Restrict a field to predefined choices.

```
STATUS_CHOICES = [
    ('draft', 'Draft'),
    ('published', 'Published'),
]
status = models.CharField(max_length=10, choices=STATU
S_CHOICES, default='draft')
```

6. **Custom Validation**: Validate field values using custom logic.

```
age = models.IntegerField(validators=[MinValueValidato
r(0), MaxValueValidator(120)])
```

Example Model:

```
from django.db import models
from django.core.validators import MinValueValidator, Max
ValueValidator

class Employee(models.Model):
    eno = models.IntegerField()
    ename = models.CharField(max_length=30)
    esal = models.FloatField()
    eaddr = models.IntegerField()
    email = models.EmailField(unique=True)
    hire_date = models.DateField()
    status = models.CharField(max_length=10, choices=[('a
```


Khushal Savalakha

```
ctive', 'Active'), ('inactive', 'Inactive')])
    age = models.IntegerField(validators=[MinValueValidat
or(18), MaxValueValidator(65)])
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

This example demonstrates a variety of fields, validations, and customizations you can define in your Django models. The `Employee` model, for instance, combines different fields with attributes like `unique`, `choices`, and `validators`.

> 💡 created_at = models.DateTimeField(auto_now_add=True)
> updated_at = models.DateTimeField(auto_now=True)
> it means it will save date and time of operation like insert and update am i true ?

Yes, you're absolutely correct!

- `created_at = models.DateTimeField(auto_now_add=True)`: This field automatically captures and stores the date and time when the object is **first created** (i.e., when an insert operation occurs). Once set, the value does not change.
- `updated_at = models.DateTimeField(auto_now=True)`: This field automatically updates to the current date and time **every time the object is saved** (i.e., during update operations).

In summary:

- `created_at` will store the timestamp of when the record was first created.
- `updated_at` will store the timestamp of the most recent update.

These fields are particularly useful for tracking when records are created and last modified.

> 💡 what is the reason behind projects = models.ManyToManyField(Project)
> using this explain ?

The line `projects = models.ManyToManyField(Project)` defines a **many-to-many relationship** between two models in Django. Let’s break down what this means and why you would use it.

What is a Many-to-Many Relationship?

Khushal Savalakha

In database terms, a **many-to-many** relationship means that each record in one table can relate to multiple records in another table, and vice versa. For example:

- An **employee** can work on multiple **projects**.
- A **project** can have multiple **employees** working on it.

In Django, this kind of relationship is represented using the `ManyToManyField`.

Explanation with Example:

Let's assume you have two models: `Employee` and `Project`.

```
from django.db import models

class Project(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()

class Employee(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField()
    projects = models.ManyToManyField(Project)
```

Here, the `projects` field in the `Employee` model is a `ManyToManyField`, which signifies that an employee can be associated with multiple projects, and each project can have multiple employees.

Behind the Scenes:

When you define a `ManyToManyField` in Django, it automatically creates an additional **junction table** (also known as a **through table** or **association table**) in the database. This table will store the relationships between the `Employee` and `Project` models. For example, it will have two columns:

- `employee_id` (foreign key referencing the `Employee` model)
- `project_id` (foreign key referencing the `Project` model)

This junction table allows for multiple records in the `Employee` table to be linked to multiple records in the `Project` table, and vice versa.

Example Use Case:

Khushal Savalakha

Imagine you are managing an organization where employees are assigned to different projects. You want to keep track of which employees are working on which projects. In this scenario:

- One **Employee** might be working on several **Projects**.
- One **Project** might have multiple **Employees** working on it.

Example Code Usage:

```
# Creating instances of Project
project1 = Project.objects.create(name="Project A", descr
iption="Description of Project A")
project2 = Project.objects.create(name="Project B", descr
iption="Description of Project B")

# Creating an instance of Employee
employee1 = Employee.objects.create(name="John Doe", emai
l="john@example.com")

# Assigning projects to the employee
employee1.projects.add(project1, project2)  # Adds both P
roject A and Project B to John Doe

# Retrieving projects for an employee
john_projects = employee1.projects.all()

# Retrieving employees for a project
projectA_employees = project1.employee_set.all()
```

Why Use `ManyToManyField`?

1. **Efficient Data Representation**: Instead of creating multiple fields or separate tables manually, Django handles the many-to-many relationship automatically, saving you from writing complex database logic.
2. **Flexibility**: It allows you to easily assign and query multiple relationships between two models.
3. **Ease of Use**: Django provides simple methods (`add`, `remove`, `clear`, etc.) to manage these relationships without needing to deal with the underlying junction table directly.

Khushal Savalakha

Real-World Example:

- **Tags on a Blog Post**: A post can have multiple tags, and each tag can be associated with multiple posts.
- **Courses and Students**: A student can enroll in multiple courses, and each course can have multiple students.

In summary, `ManyToManyField` is a powerful way to represent complex relationships in Django, where each entity can be related to multiple entities of another type.

↳ **Efficient Data Representation**: Instead of creating multiple fields or separate tables manually, Django handles the many-to-many relationship automatically, saving you from writing complex database logic.

but tell me reason why we write projects = models.ManyToManyField(Project)
here we just don't write to look good tell me which type of complex logic i don't have to write because of this ?

Great question! The reason for using `projects = models.ManyToManyField(Project)` goes beyond just looking good—it simplifies the complexity of managing relationships between records in two tables. Without this, you would have to write more complex logic and handle the relationship manually. Let's dive into what you'd have to do if you didn't use `ManyToManyField` and how it simplifies things.

1. Avoiding Manual Creation of a Junction Table:

Without Django's `ManyToManyField`, you would need to manually create a separate table (often called a junction table or a join table) to store the relationships between employees and projects. This table would need to contain foreign keys to both the `Employee` and `Project` models.

For example, you might have to create a table like this:

```
class EmployeeProject(models.Model):
    employee = models.ForeignKey(Employee, on_delete=mode
ls.CASCADE)
    project = models.ForeignKey(Project, on_delete=model
s.CASCADE)
```

Now, every time you wanted to assign an employee to a project, you would need to manually insert a row into this `EmployeeProject` table. Managing these inserts, updates, and deletions would require additional logic in your code.

Khushal Savalakha

2. Simplifying Queries and Data Management:

With a manually created junction table, retrieving the projects an employee is working on would require a complex query:

```
# Without ManyToManyField
employee_projects = EmployeeProject.objects.filter(employ
ee=some_employee)
projects = [relation.project for relation in employee_pro
jects]
```

With `ManyToManyField`, you can simply do:

```
projects = some_employee.projects.all()
```

This is much simpler, more readable, and less error-prone.

Similarly, adding a new project to an employee would require additional logic:

```
# Without ManyToManyField
EmployeeProject.objects.create(employee=some_employee, pr
oject=some_project)
```

With `ManyToManyField`, you can just do:

```
some_employee.projects.add(some_project)
```

3. Automated Integrity Management:

When you use `ManyToManyField`, Django automatically handles referential integrity. For example, if you delete an `Employee`, Django will automatically manage what happens to the related projects based on the `on_delete` behavior. Without this field, you would need to manually ensure that all related records in the `EmployeeProject` table are correctly updated or deleted.

4. Avoiding Duplicate Logic and Handling Edge Cases:

Managing a many-to-many relationship without `ManyToManyField` requires custom logic to prevent duplicates in the join table. You'd need to check if a relationship already exists before adding a new one. For example:

Khushal Savalakha

```
# Without ManyToManyField
if not EmployeeProject.objects.filter(employee=some_emplo
yee, project=some_project).exists():
    EmployeeProject.objects.create(employee=some_employe
e, project=some_project)
```

With `ManyToManyField`, this is automatically handled by Django:

```
some_employee.projects.add(some_project)
```

5. Effortless Data Migration and Schema Evolution:

When you define a `ManyToManyField`, Django handles the database schema creation and updates for you. This includes creating the junction table, adding/removing fields, and generating migrations. Without this, you'd have to manually manage the database schema, which could be prone to errors and require more effort to maintain over time.

6. Access to Django's ORM Features:

By using `ManyToManyField`, you get access to Django’s powerful ORM features. You can perform complex queries, filtering, and operations (like counting related objects) directly through the ORM, without needing to write raw SQL.

For example:

- Counting the number of projects an employee is working on:

  ```
  project_count = some_employee.projects.count()
  ```

- Finding employees who are working on a specific project:

  ```
  employees = Employee.objects.filter(projects=some_proj
  ect)
  ```

Without `ManyToManyField`, you'd need to write these queries manually using Django’s `QuerySet` API or raw SQL, which would be more complicated and error-prone.

Summary:

Using `projects = models.ManyToManyField(Project)` saves you from:

- Manually creating and managing a junction table.

Khushal Savalakha

- Writing complex queries to handle relationships.
- Manually ensuring referential integrity and handling edge cases like duplicates.
- Managing schema evolution and data migrations.
- Missing out on the convenience and power of Django's ORM features.

In short, `ManyToManyField` simplifies your code, reduces the potential for bugs, and allows you to focus on the business logic rather than low-level database management.

> 💡 it means when relation is Many to Many then every thing is connected to each other on that time instead of writing extra code like checking wheather this data is also inserted in this relation or not or it is duplicated or if it is deleted then we also have to delete from other relation that types of conditions we don't have to handle because we define Many to Many relation then backend automatically checks that types of things am i true ?

Yes, you're absolutely right!

When you define a **Many-to-Many** relationship using `ManyToManyField` in Django, the framework handles all of the complex operations for you in the backend. This includes:

1. **Avoiding Duplicates:** Django ensures that the same relationship (e.g., the same employee assigned to the same project) isn't duplicated in the database. You don't have to write extra code to check if a relationship already exists before adding it.
2. **Automatic Insertions:** When you add a new relationship (e.g., assigning an employee to a project), Django automatically inserts the appropriate entries into the junction table, without requiring you to manage this manually.
3. **Automatic Deletions:** If you delete an object (e.g., an employee or a project), Django will handle the removal of the corresponding relationships from the junction table, depending on the `on_delete` behavior you've specified. This means you don't have to manually delete related entries.
4. **Referential Integrity:** Django maintains the integrity of the relationships between your models. If an object is removed, Django ensures that the database remains consistent by updating or removing related records as necessary.
5. **Efficient Querying:** You can query relationships easily (e.g., finding all projects an employee is working on) without needing to write complex SQL queries or worry about joins and filtering.

Khushal Savalakha

In summary:

- **No need to manually check for duplicates.**
- **No need to manually insert relationships into a junction table.**
- **No need to manually delete relationships from a junction table.**
- **Automatic management of referential integrity.**
- **Simplified querying and data access.**

Django's `ManyToManyField` abstracts away all of these complexities, allowing you to work at a higher level and focus on your application's business logic rather than low-level database operations. So yes, you're absolutely correct in saying that Django's `ManyToManyField` automates these tasks and spares you from having to write that extra code.

> 💡 give me one example with making table and junction and do operations mentioned above so how Django is working i can understand by proper table visualization and change explain simply and depthly. here i don't want to query in postgres you have to draw a table .

Let's break down the concept of a **Many-to-Many** relationship with a clear, step-by-step explanation. We'll visualize how Django manages the relationship between two tables using an **Employee** and **Project** example.

Step 1: Initial Tables Without Many-to-Many Field

Employee Table:

id (PK)	name	email
1	John Doe	john@example.com
2	Jane Smith	jane@example.com

Project Table:

id (PK)	name	description
1	Project A	Description A
2	Project B	Description B

Step 2: The Junction Table (Automatically Created by Django)

Khushal Savalakha

When you define a `ManyToManyField`, Django automatically creates a **junction table** (also called a **through table**). This table holds the relationships between `Employee` and `Project`.

EmployeeProject (Junction Table):

id (PK)	employee_id (FK)	project_id (FK)
1	1	1
2	1	2
3	2	1

Step 3: Operations and How Django Manages Them

Operation 1: Adding a Relationship (Assigning John Doe to Project A and Project B)

When you do:

```
john_doe.projects.add(project_a, project_b)
```

Junction Table Update:

- Django automatically inserts records into the **EmployeeProject** table:
  - (employee_id=1, project_id=1)
  - (employee_id=1, project_id=2)

Junction Table After the Operation:

id (PK)	employee_id (FK)	project_id (FK)
1	1	1
2	1	2

Operation 2: Avoiding Duplicates

When you try to add the same relationship again:

```
john_doe.projects.add(project_a)
```

Django checks if `(employee_id=1, project_id=1)` already exists in the junction table. Since it does, it **does not** add a duplicate entry.

Junction Table After the Operation:

Khushal Savalakha

id (PK)	employee_id (FK)	project_id (FK)
1	1	1
2	1	2

Operation 3: Removing a Relationship (Removing John Doe from Project A)

When you do:

```
john_doe.projects.remove(project_a)
```

Junction Table Update:

- Django automatically deletes the record `(employee_id=1, project_id=1)` from the **EmployeeProject** table.

Junction Table After the Operation:

id (PK)	employee_id (FK)	project_id (FK)
2	1	2

Operation 4: Deleting an Employee or Project

If you delete John Doe from the `Employee` table:

```
john_doe.delete()
```

Django automatically deletes all related records in the junction table. So, the entry `(employee_id=1)` is removed from **EmployeeProject**.

Junction Table After the Operation:

id (PK)	employee_id (FK)	project_id (FK)
(Empty)		

Step 4: How Django Manages It

1. **Adding Records:**
    - When you add a project to an employee, Django inserts a new row into the junction table.
    - If you try to add a duplicate, Django prevents it automatically.

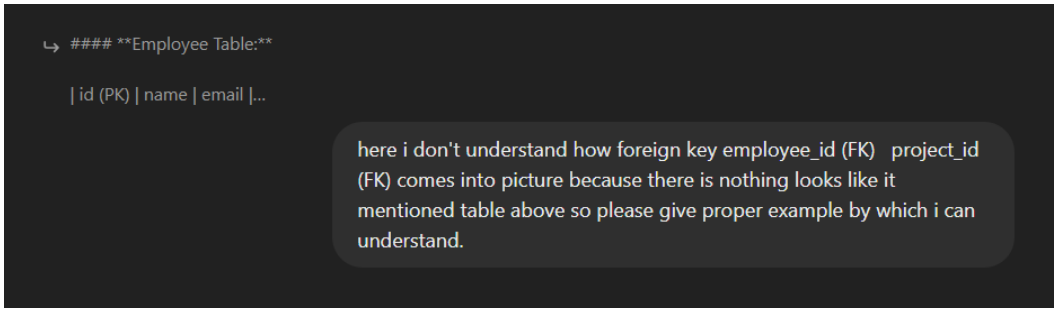
Khushal Savalakha

2. **Removing Records:**
    - When you remove a project from an employee, Django deletes the corresponding row from the junction table.
3. **Referential Integrity:**
    - If an employee or project is deleted, Django automatically removes the related rows in the junction table, ensuring the database remains consistent.

Summary:

- The **Employee** and **Project** tables remain clean, containing only the core data for employees and projects.
- The **EmployeeProject (Junction Table)** manages the many-to-many relationship between them.
- Django handles the complexity of inserting, updating, deleting, and avoiding duplicates within the junction table.
- This means you **don't need to manually** write complex SQL queries or handle all of the potential edge cases.

This visualization should help you understand how Django simplifies managing many-to-many relationships by automatically creating and maintaining a junction table in the background. You just interact with the higher-level model objects, and Django takes care of the rest.



Let me explain the concept of **Foreign Keys (FK)** in the context of the Many-to-Many relationship between the `Employee` and `Project` tables.

Understanding Foreign Keys in a Junction Table

In a relational database, **Foreign Keys (FK)** are used to link rows in one table to rows in another table. When you have a **Many-to-Many** relationship between two tables, a third

Khushal Savalakha

table—often called a **junction table** or **through table**—is automatically created by Django to manage the relationship between these two tables.

The junction table will contain **Foreign Keys** that point to the primary keys (PK) of the `Employee` and `Project` tables. This is how the connection between the two tables is maintained.

Let's go step by step.

Step 1: The Employee Table

id (PK)	name	email
1	John Doe	john@example.com
2	Jane Smith	jane@example.com

Step 2: The Project Table

id (PK)	name	description
1	Project A	Description A
2	Project B	Description B

Step 3: The Junction Table (EmployeeProject)

When you define a Many-to-Many relationship using `ManyToManyField` in Django, the framework automatically creates a **junction table**. This table contains two columns that are **Foreign Keys (FK)** pointing to the `Employee` and `Project` tables.

This is how the junction table might look:

employee_id (FK)	project_id (FK)
1	1
1	2
2	1

Explanation of the Junction Table:

- **employee_id (FK):** This is a foreign key that refers to the **id** in the `Employee` table. For example, `1` in this column refers to "John Doe."
- **project_id (FK):** This is a foreign key that refers to the **id** in the `Project` table. For example, `1` in this column refers to "Project A."

Example Operations:

Khushal Savalakha

1. **Assigning John Doe to Project A and Project B:**
   - Django inserts the following rows into the junction table:
     - `(employee_id=1, project_id=1)` for John Doe → Project A
     - `(employee_id=1, project_id=2)` for John Doe → Project B
2. **Assigning Jane Smith to Project A:**
   - Django inserts the following row into the junction table:
     - `(employee_id=2, project_id=1)` for Jane Smith → Project A
3. **Avoiding Duplicates:**
   - If you try to assign John Doe to Project A again, Django checks the junction table and sees that `(employee_id=1, project_id=1)` already exists. Therefore, it does not insert a duplicate row.
4. **Removing Relationships:**
   - If you remove John Doe from Project A, Django deletes the row `(employee_id=1, project_id=1)` from the junction table.
5. **Referential Integrity:**
   - If you delete John Doe from the `Employee` table, Django automatically removes all rows in the junction table where `employee_id=1`.

Why Foreign Keys Are Important:

- **Foreign Keys** ensure that the relationships between tables are maintained correctly. For example, if you try to insert a row into the junction table with `employee_id=3`, but there is no `id=3` in the `Employee` table, the database will reject the insertion because it violates the foreign key constraint.
- They also help with cascading operations, like automatically deleting related rows in the junction table when an employee or project is deleted.

Summary:

- The **junction table** is automatically created by Django when you define a `ManyToManyField`.
- This table contains two **Foreign Keys**—one pointing to the `Employee` table and one pointing to the `Project` table.

Khushal Savalakha

- This setup allows you to manage the many-to-many relationship efficiently, without manually handling inserts, deletes, or referential integrity checks.

By using `ManyToManyField`, Django abstracts all this complexity away, allowing you to work with these relationships in a simple and intuitive manner.

```
<!DOCTYPE html>
<html lang="en">
{% load static %}
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initi
    <title>Employee Data</title>
    <link rel="stylesheet" href="{% static 'css/emp1.css' %}
</head>
<body>
    <h1>Hello</h1>
    <p>DUBAI:
        <img src="{% static 'images\dubai_palm_jumeirah_isla
        Natherland:
        <img src="{% static 'images\Natherland.jpg' %}" alt=
    </p>
</body>
</html>
```

```
"""
Django settings for modelproject3 project.

Generated by 'django-admin startproject' using Django 5.0.7.

For more information on this file, see
https://docs.djangoproject.com/en/5.0/topics/settings/

For the full list of settings and their values, see
https://docs.djangoproject.com/en/5.0/ref/settings/
"""

from pathlib import Path
```


Khushal Savalakha

```
# Build paths inside the project like this: BASE_DIR / 'subd
BASE_DIR = Path(__file__).resolve().parent.parent
TEMPLATE_DIR=BASE_DIR/'templates'
STATIC_DIR=BASE_DIR/'static'

# Quick-start development settings - unsuitable for producti
# See https://docs.djangoproject.com/en/5.0/howto/deployment

# SECURITY WARNING: keep the secret key used in production s
SECRET_KEY = 'django-insecure-jbuw5=usl0f889_0!mh%)oq!t@3=u_

# SECURITY WARNING: don't run with debug turned on in produc
DEBUG = True

ALLOWED_HOSTS = []


# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'testapp'
]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware'
]
```


Khushal Savalakha

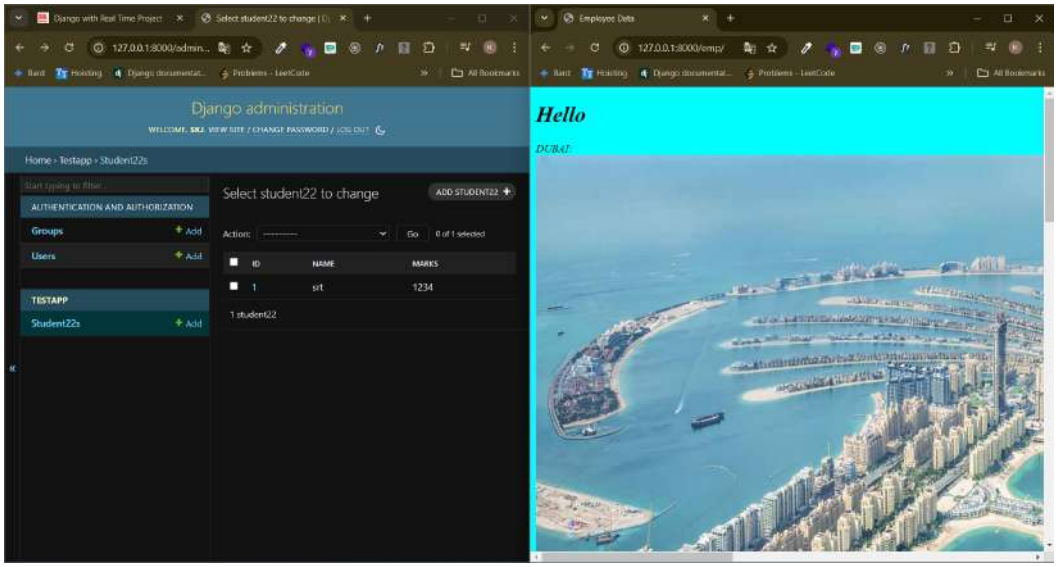
```
ROOT_URLCONF = 'modelproject3.urls'

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTe
        'DIRS': [TEMPLATE_DIR],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request'
                'django.contrib.auth.context_processors.auth
                'django.contrib.messages.context_processors.
            ],
        },
    },
]

WSGI_APPLICATION = 'modelproject3.wsgi.application'


# Database
# https://docs.djangoproject.com/en/5.0/ref/settings/#databa

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',  # Postgr
        'NAME': 'Django_Trail2',               # Replace wit
        'USER': 'postgres',
        'PASSWORD': 'khushal',
        'HOST': 'localhost',
        'PORT': '5432',                            # Defaul
    }
}


# Password validation
# https://docs.djangoproject.com/en/5.0/ref/settings/#auth-p
```

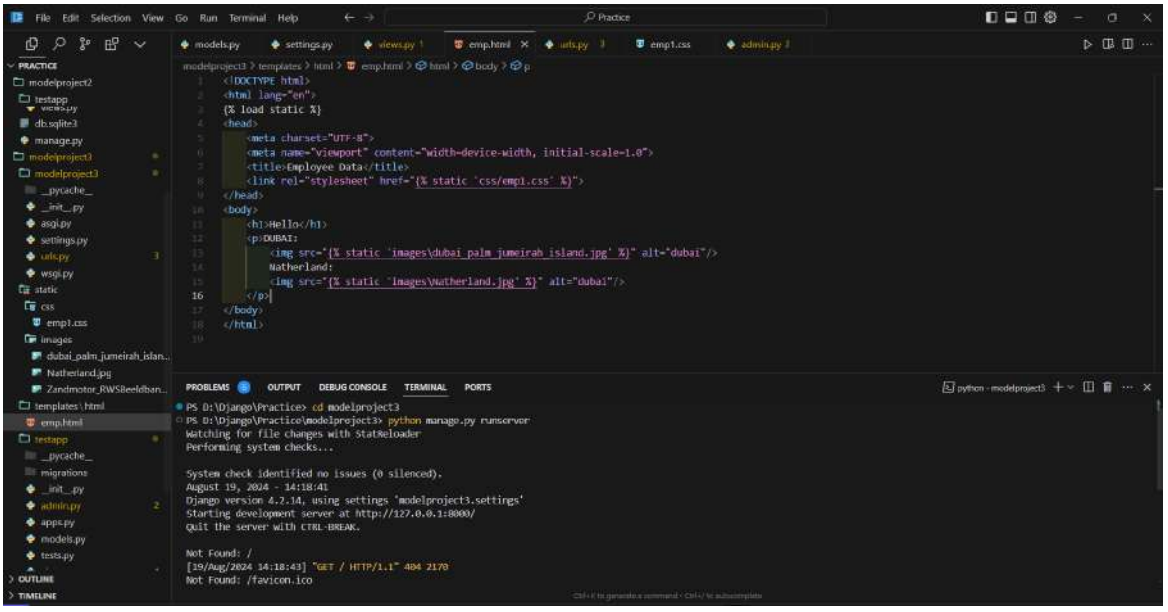

Khushal Savalakha

```
AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME': 'django.contrib.auth.password_validation.Use
    },
    {
        'NAME': 'django.contrib.auth.password_validation.Mir
    },
    {
        'NAME': 'django.contrib.auth.password_validation.Com
    },
    {
        'NAME': 'django.contrib.auth.password_validation.Num
    },
]


# Internationalization
# https://docs.djangoproject.com/en/5.0/topics/i18n/

LANGUAGE_CODE = 'en-us'

TIME_ZONE = 'UTC'

USE_I18N = True

USE_TZ = True


# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/5.0/howto/static-files/

STATIC_URL = 'static/'
STATICFILES_DIRS=[
    STATIC_DIR,
]
# Default primary key field type
# https://docs.djangoproject.com/en/5.0/ref/settings/#defaul
```


Khushal Savalakha

```
DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'
```





```python
from django.db import models

# Create your models here.
class Employee(models.Model):
    eno = models.IntegerField()
    ename = models.CharField(max_length=30)
    esal = models.FloatField()
    eaddr = models.CharField(max_length=30)

```

In Django, it is necessary to specify the `max_length` attribute for fields like `CharField` because it defines the maximum number of characters that can be stored in that field. The `max_length` parameter is required for `CharField` and `TextField` to ensure the proper storage size in the database. Without specifying `max_length`, Django will raise a validation error.

For other field types like `IntegerField` or `FloatField`, `max_length` is not required because these fields inherently have a defined size based on their data type.

```html
<!DOCTYPE html>
<html lang="en">
  {% load static %}
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initi
    <title>Student List</title>
    <link rel="stylesheet" href="{% static 'css/emp1.css' %}
  </head>
  <body>
    <h1>Student List</h1>

    {% if student_list %}
    <table>
      <thead>
        <th>ID</th>
        <th>NAME</th>
```


Khushal Savalakha

```
        <th>Marks</th>
      </thead>
      {% for student in student_list %}
      <tr>
        <td>{{student.id}}</td>
        <td>{{student.name}}</td>
        <td>{{student.marks}}</td>
      </tr>
      {% endfor %}
    </table>
    {% else %}
    <p>No data found!!!</p>
    {% endif %}
    <p> No records found </p>
  </body>
</html>
```

In Django templates, you can use `{% elif %}` for else-if logic within an `{% if %}` block. However, this is generally used for checking additional conditions rather than for loops like the one in your code. In your specific case, if you want to add an `elif` condition to check something specific (for example, checking if `student_list` is empty but another condition is true), you can do something like this:

Khushal Savalakha

```
<!DOCTYPE html>
<html lang="en">
  {% load static %}
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, in
itial-scale=1.0" />
    <title>Student List</title>
    <link rel="stylesheet" href="{% static 'css/emp1.css'
%}" />
  </head>
  <body>
    <h1>Student List</h1>

    {% if student_list %}
    <table>
      <thead>
        <th>ID</th>
        <th>NAME</th>
        <th>Marks</th>
      </thead>
      {% for student in student_list %}
      <tr>
        <td>{{ student.id }}</td>
        <td>{{ student.name }}</td>
        <td>{{ student.marks }}</td>
      </tr>
      {% endfor %}
    </table>
    {% elif some_other_condition %}
    <p>Some other condition met, but no students found.</
p>
    {% else %}
    <p>No data found!!!</p>
    {% endif %}

    <p> No records found </p>
```


Khushal Savalakha

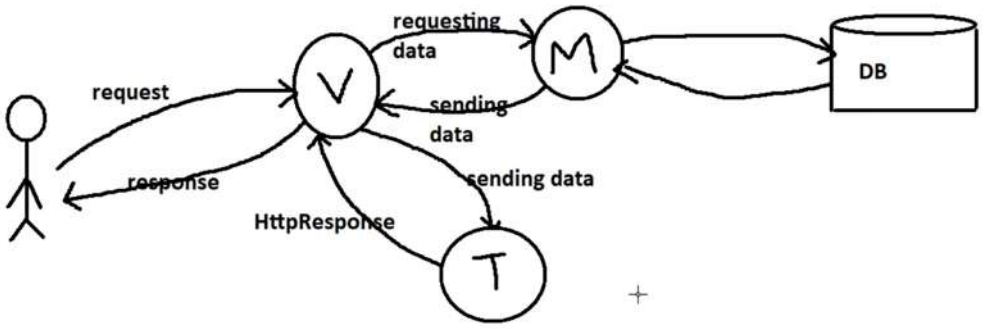
```
  </body>
</html>
```

Explanation:

1. `{% elif some_other_condition %}` **:** This is where you can check another condition. If `student_list` is empty but `some_other_condition` is true, this block will be executed.
2. **Final** `else` **block:** This will execute if `student_list` is empty and `some_other_condition` is not true.

In your original code, you have both a `{% if student_list %}` block and an unconditional `<p> No records found </p>` at the bottom. If you only want one of those messages to show based on conditions, remove the unconditional `<p> No records found </p>` or integrate its logic into the `{% if %}` structure.

```
System check identified no issues (0 silenced).
August 20, 2024 - 20:20:05
Django version 4.2.14, using settings 'modelproject3.setting
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.

<QuerySet [<Student22: Student22 object (1)>, <Student22: St
```



> 💡 is it proper flow in Django that first user send request to view then view request to database then database sending data to Templates and templates response to view in http response and then view response back to end user am i true ?

The flow you described has the right components but is slightly mixed up in terms of the order and interactions. Here's the correct flow in a Django application:

Khushal Savalakha

1. **User Sends Request**: The user sends an HTTP request to the Django server, which is routed to the appropriate **view** based on the URL.
2. **View Handles Request**: The **view** function processes the request. If it needs data, it will query the **database** using Django's ORM (Object-Relational Mapping).
3. **Database Query**: The **database** returns the requested data to the **view**.
4. **View Prepares Data for Template**: The **view** passes the data to a **template**, typically as a context dictionary.
5. **Template Renders HTML**: The **template** uses the provided data to generate an HTML page or other content (e.g., JSON for APIs).
6. **View Returns Response**: The **view** then returns an HTTP response to the user, containing the rendered content.

So, the correct flow is:

1. **User Request** → 2. **View** → 3. **Database** → 4. **View** → 5. **Template** → 6. **View Response to User**

Would you like to dive deeper into any of these steps?

```
PS D:\Django\Practice\modelproject3> python manage.py help

Type 'manage.py help <subcommand>' for help on a specific su

Available subcommands:

[auth]
    changepassword
    createsuperuser

[contenttypes]
    remove_stale_contenttypes

[django]
    check
    compilemessages
    createcachetable
    dbshell
    diffsettings
    dumpdata
```


Khushal Savalakha

```
    flush
    inspectdb
    loaddata
    makemessages
    makemigrations
    migrate
    optimizemigration
    sendtestemail
    shell
    showmigrations
    sqlflush
    sqlmigrate
    sqlsequencereset
    squashmigrations
    startapp
    startproject
    test
    testserver

[sessions]
    clearsessions

[staticfiles]
    collectstatic
    findstatic
    runserver
PS D:\Django\Practice\modelproject3>
```

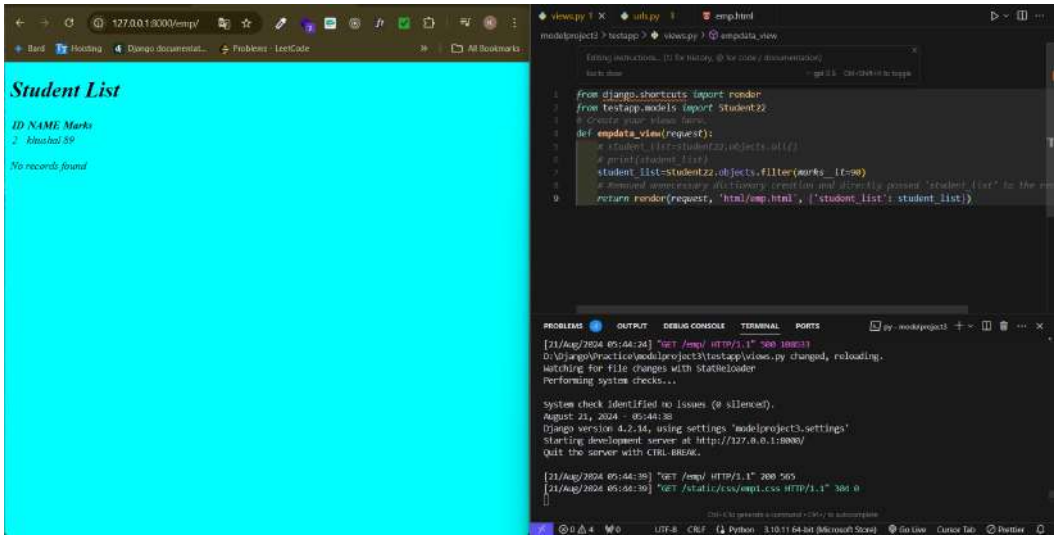

Khushal Savalakha

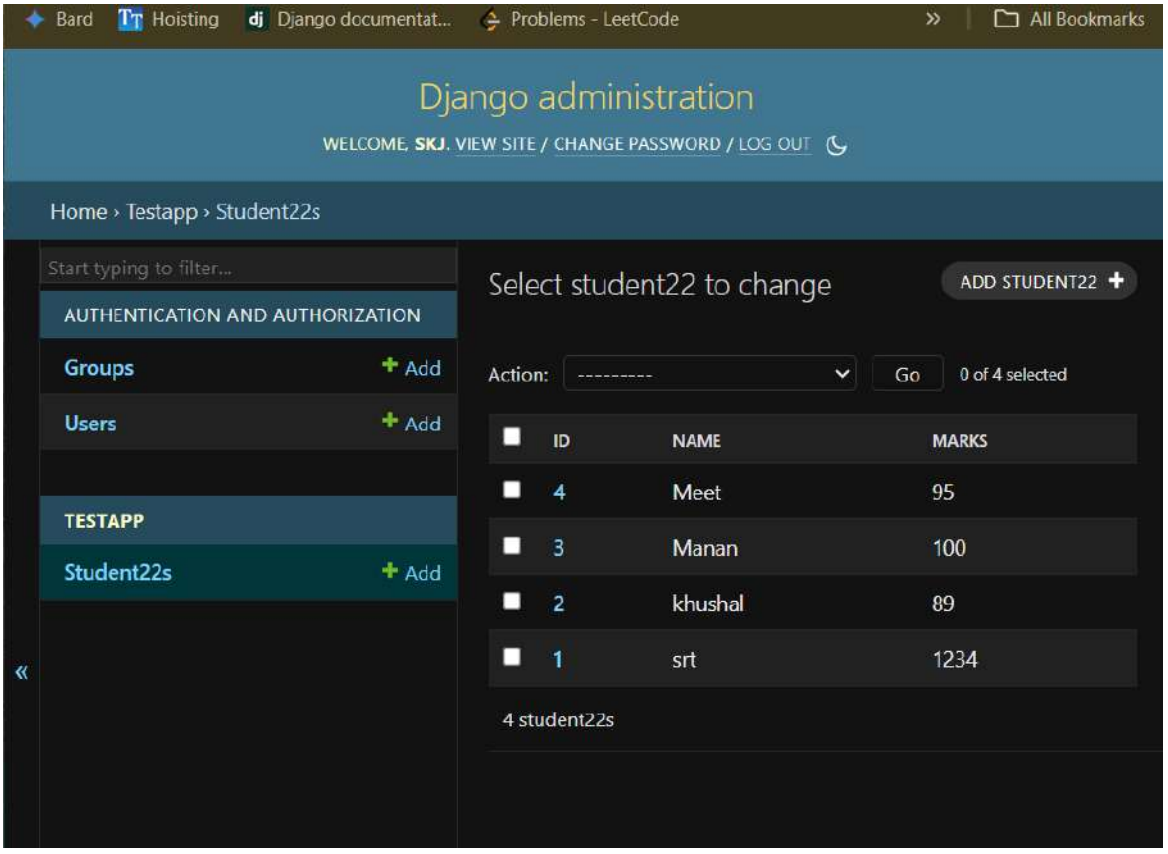
```python
from faker import Faker
fakegen = Faker()
name = fakegen.name()
print(name)
fname = fakegen.first_name()
print(fname)
lname = fakegen.last_name()
print(lname)
date1 = fakegen.date()
print(date1)
number = fakegen.random_number(5)
print(number)
email1 = fakegen.email()
print(email1)
```

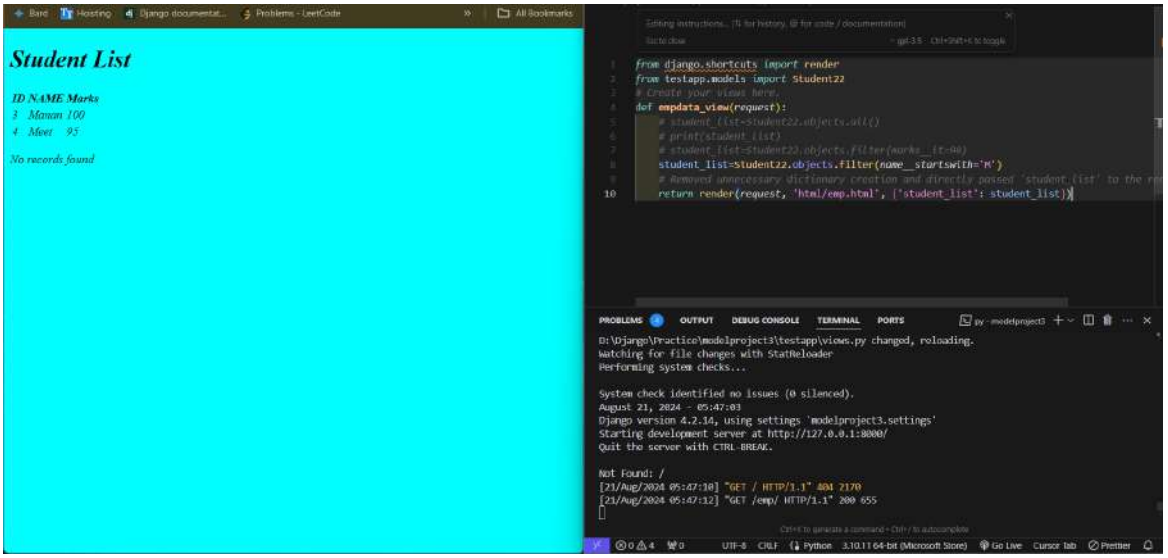

Khushal Savalakha

```
D:\durgaclasses>py test.py
Lawrence Smith
Ronald
Conway
1974-07-17
44408
john31@spencer.com

D:\durgaclasses>py test.py
Ashley Rodriguez
Amanda
Williams
1987-12-23
21858
gabrielle91@gmail.com
```

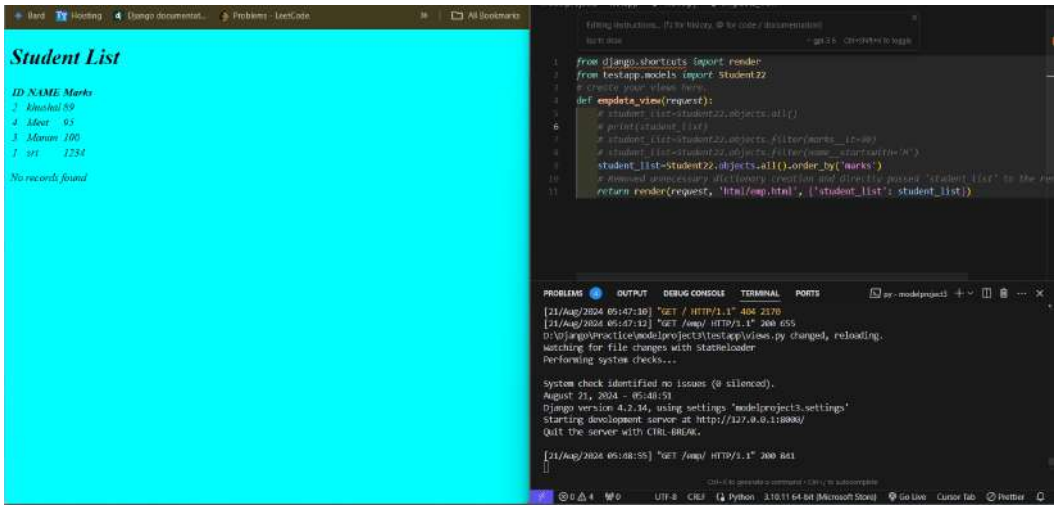
```
from django.shortcuts import render
from testapp.models import Student22
# Create your views here.
def empdata_view(request):
    # student_list=Student22.objects.all()
    # print(student_list)
    student_list=Student22.objects.filter(marks__lt=90)#ORM
    # Removed unnecessary dictionary creation and directly p
    return render(request, 'html/emp.html', {'student_list':
```

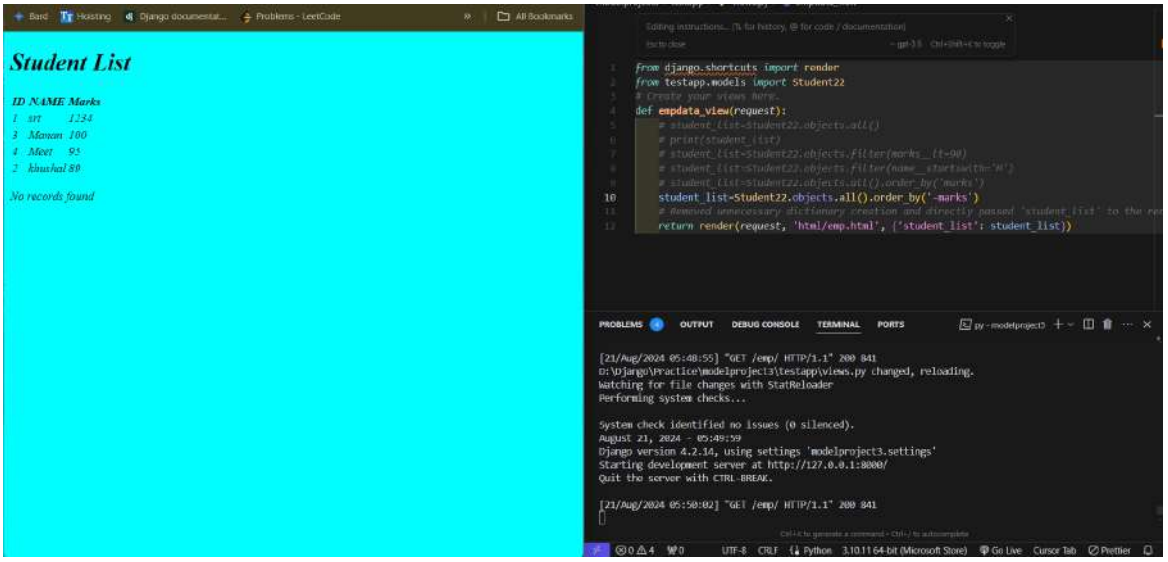

Student List
ID NAME Marks
2 khushal 89
No records found
from django.shortcuts import render
from testapp.models import Student22
def empdata_view(request):
student_list=Student22.objects.filter(marks__lt=90)
return render(request, 'html/emp.html', {'student_list': student_list})
[21/Aug/2024 05:44:34] "GET /emp/ HTTP/1.1" 200 565


Bard
Hoisting
Django documentat...
Problems - LeetCode
All Bookmarks
Django administration
WELCOME, SKJ. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Testapp › Student22s
Start typing to filter...
AUTHENTICATION AND AUTHORIZATION
Groups + Add
Users + Add
TESTAPP
Student22s + Add
Select student22 to change
ADD STUDENT22 +
Action: ---------
Go
0 of 4 selected
ID NAME MARKS
4 Meet 95
3 Manan 100
2 khushal 89
1 srt 1234
4 student22s



It is case sensitive.





```
from django.shortcuts import render
from testapp.models import Student22
# Create your views here.
def empdata_view(request):
    # student_list=Student22.objects.all()
    # print(student_list)
    # student_list=Student22.objects.filter(marks__lt=90)
    # student_list=Student22.objects.filter(name__startswith
    # student_list=Student22.objects.all().order_by('marks')
    student_list=Student22.objects.all().order_by('-marks')
    # Removed unnecessary dictionary creation and directly p
    return render(request, 'html/emp.html', {'student_list':
```

▼ Jobs Project

```
from django.db import models

# Create your models here.
class HydJobs(models.Model):
    date=models.DateField()
    company=models.CharField(max_length=100)
    title=models.CharField(max_length=30)
    eligibility=models.CharField(max_length=30)
    address=models.CharField(max_length=30)
    email=models.EmailField()
    phonenumber=models.BigIntegerField()

class BangloreJobs(models.Model):
    date=models.DateField()
    company=models.CharField(max_length=100)
    title=models.CharField(max_length=30)
    eligibility=models.CharField(max_length=30)
    address=models.CharField(max_length=30)
    email=models.EmailField()
    phonenumber=models.BigIntegerField()

class PuneJobs(models.Model):
    date=models.DateField()
```


Khushal Savalakha

```
    company=models.CharField(max_length=100)
    title=models.CharField(max_length=30)
    eligibility=models.CharField(max_length=30)
    address=models.CharField(max_length=30)
    email=models.EmailField()
    phonenumber=models.BigIntegerField()
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, in
    <title>Document</title>
</head>
<body>
    <h1>Hyderabad Job Info</h1>
    {% if jobs_list %}
    <table>
        <thead>
            <th>Date</th>
            <th>Company</th>
            <th>Title</th>
            <th>Eligibily</th>
            <th>Address</th>
            <th>Email</th>
            <th>Phone Number</th>
        </thead>
        {% for job in jobs_list %}
        <tr>
            <td>{{job.date}}</td>
            <td>{{job.Company}}</td>
            <td>{{job.title}}</td>
            <td>{{job.eligibility}}</td>
            <td>{{job.address}}</td>
            <td>{{job.email}}</td>
            <td>{{job.phonenumber}}</td>
        </tr>
        {% endfor %}
```


Khushal Savalakha

```
    {% else %}
    <p>No  data found!!</p>
    {% endif %}
    </table>
</body>
</html>
```

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, in
    <title>Durga Jobs</title>
    <link rel="stylesheet" href='{% static "css/index.css
</head>
<body>
    <h1>Home Page</h1>
    <a href="{% url 'hyderabad' %}"><button>Hyderabad Job
    <a href="#"><button>Bangalore Jobs</button></a>
    <a href="#"><button>Pune Jobs</button></a>
</body>
</html>
```

```
from django.shortcuts import render
from testapp.models import HydJobs,BangloreJobs,PuneJobs
# Create your views here.

def home_views(request):
    return render(request,'testapp/index.html')

def hyd_views(request):
    jobs_list=HydJobs.objects.all()
    return render(request,'testapp/hyd.html',{'jobs_list'
```

here whatever name we specified to usrl we have to write it in url

Khushal Savalakha

```
"""
URL configuration for Jobs project.

The `urlpatterns` list routes URLs to views. For more inf
    https://docs.djangoproject.com/en/5.0/topics/http/url
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, na
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view()
Including another URLconf
    1. Import the include() function: from django.urls im
    2. Add a URL to urlpatterns:  path('blog/', include('
"""
from django.contrib import admin
from django.urls import path
from testapp import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('',views.home_views),
    path('hyd/', views.hyd_views, name='hyderabad'),
]
```

💡 what is the logic behind name and url how they are connected ?

The connection between the `name` parameter in Django's URL patterns and the actual URL is established through Django's URL routing system. Here's how it works:

1. URL Configuration (`urls.py`):

When you define a URL pattern in Django, you can give it a `name`:

```
from django.urls import path
from . import views
```


Khushal Savalakha

```
urlpatterns = [
    path('hyd/', views.hyd, name='hyd'),
]
```

In this example, the URL pattern is for the path `hyd/`, and it's associated with the `views.hyd` view function. The `name='hyd'` is a label for this URL pattern.

2. Connecting `name` and URL:

- **URL Pattern**: The string `'hyd/'` represents the actual URL path the user would navigate to in the browser (`http://localhost:8000/hyd/`).
- **Name**: The `name='hyd'` allows you to refer to this URL pattern elsewhere in your Django project without needing to use the actual URL string `'hyd/'`.

3. Using `name` in Templates:

In your templates, you can use the `{% url %}` template tag to generate URLs based on the `name`:

```
<a href="{% url 'hyd' %}">Hyderabad Jobs</a>
```

- **How It Works**: When Django renders this template, it looks for the URL pattern with `name='hyd'`. It then generates the appropriate URL path (`hyd/`) and substitutes it in the `href` attribute of the `<a>` tag.
- **Output**: The HTML rendered would be:

  ```
  <a href="/hyd/">Hyderabad Jobs</a>
  ```

4. Using `name` in Views:

You can also use the `name` in your views, for example, when redirecting:

```
from django.shortcuts import redirect

def some_view(request):
    return redirect('hyd')
```

- **How It Works**: The `redirect('hyd')` call uses the `name='hyd'` to find the corresponding URL pattern (`hyd/`) and redirects the user to that path.

Khushal Savalakha

5. Reverse URL Lookup:

Django provides the `reverse()` function, which allows you to get the actual URL path from the `name`:

```
from django.urls import reverse

url = reverse('hyd')
print(url)  # Output: /hyd/
```

- **How It Works**: The `reverse('hyd')` function looks up the URL pattern with `name='hyd'` and returns the corresponding URL path (`'/hyd/'`).

6. Why This Matters:

- **Flexibility**: If you decide to change the URL from `hyd/` to `hyderabad-jobs/`, you only need to update the URL pattern in `urls.py`:

  ```
  urlpatterns = [
      path('hyderabad-jobs/', views.hyd, name='hyd'),
  ]
  ```

  Now, everywhere you used `{% url 'hyd' %}` or `reverse('hyd')`, Django will automatically generate the new URL path `/hyderabad-jobs/`.

- **Avoiding Hardcoding**: This approach reduces the risk of bugs and inconsistencies that could arise from hardcoding URLs throughout your project.

Summary:

The `name` in Django's URL patterns acts as a label or identifier for a specific URL path. The `{% url %}` template tag, `reverse()` function, and `redirect()` function use this `name` to dynamically generate the correct URL, providing flexibility and maintainability across your Django project. This connection allows you to change URLs easily without modifying code throughout your project.

💡 it means name stores url am i true ?

Not exactly. The `name` in Django's URL patterns does not directly store the URL; rather, it serves as a unique identifier or label that refers to a specific URL pattern defined in your `urls.py` file.

Khushal Savalakha

How It Works:

- **URL Pattern**: When you define a URL pattern in Django, you're specifying both a URL path and a view function that should handle requests to that path.

  ```
  urlpatterns = [
      path('hyd/', views.hyd, name='hyd'),
  ]
  ```

  - `'hyd/'`: This is the actual URL path.
  - `views.hyd`: This is the view function that will be called when someone navigates to the `'hyd/'` URL.
  - `name='hyd'`: This is a label that you can use to refer to this URL pattern elsewhere in your Django project.

Key Points:

- **The `name` is an identifier**: It's a name that you assign to the URL pattern. It doesn't store the URL itself but serves as a key or reference to it.
- **Django's Routing System**: Django internally maps the `name` to the corresponding URL pattern. When you use `{% url 'hyd' %}` in a template or `reverse('hyd')` in your views, Django looks up the URL pattern associated with the `name` and then generates the corresponding URL.
- **Flexibility and Abstraction**: By using `name`, you abstract away the actual URL string. This means that you can change the URL path in one place (in `urls.py`), and all references using the `name` will automatically point to the new URL.

Analogy:

Think of the `name` as a nickname or alias for the URL pattern. The actual URL is like someone's full name, while the `name` is a nickname that you can use to refer to the URL pattern in a simpler, more maintainable way.

Example Recap:

- **URL pattern**: `'hyd/'`
- **Name**: `'hyd'`
- **Usage in a template**:
  Here,

Khushal Savalakha

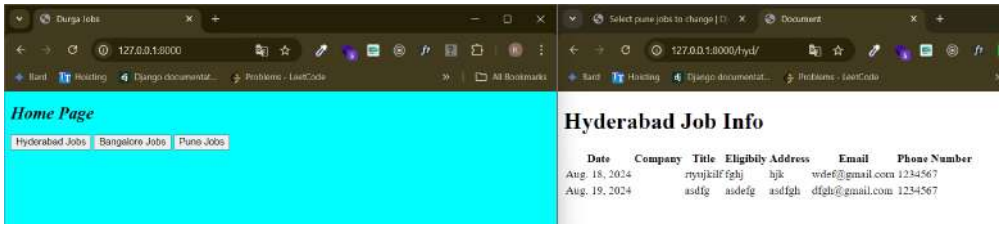
`{% url 'hyd' %}` doesn't store the URL `'hyd/'`, but it uses the `name` `'hyd'` to look up and generate the corresponding URL.

```
<a href="{% url 'hyd' %}">Hyderabad Jobs</a>
```

Summary:

The `name` in Django's URL patterns doesn't store the URL but acts as an identifier that refers to the URL pattern. This identifier allows you to easily reference and manage URLs throughout your Django project, providing flexibility and reducing the need to hardcode URLs in your templates and views.

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, in
    <title>Durga Jobs</title>
    <link rel="stylesheet" href='{% static "css/index.css
</head>
<body>
    <h1>Home Page</h1>
    <!-- <a href="{% url 'hyderabad' %}"><button>Hyderaba
    <a href="/hyd/"><button>Hyderabad Jobs</button></a>
    <a href="#"><button>Bangalore Jobs</button></a>
    <a href="#"><button>Pune Jobs</button></a>
</body>
</html>
```



Khushal Savalakha

💡 tell me when to use makemigrations and migrate in django like when we use form,Model tell me all types of combinations and thier work flow depthly in Django with example.

In Django, `makemigrations` and `migrate` are commands used to manage database schema changes based on your Django models. Here's a detailed explanation of when and why to use these commands, along with examples to illustrate their workflow.

Overview of Commands

- `makemigrations`: This command is used to create new migration files based on the changes made to your Django models. Migrations are essentially a way to record changes to the database schema, so Django knows how to apply them to the database.
- `migrate`: This command applies the migrations to your database. It updates the database schema to match your current models as described in the migration files.

When to Use `makemigrations` and `migrate`

1. **Creating or Modifying Models:**
   - **When to Use**: Anytime you create a new model or modify an existing model (e.g., adding or removing fields, changing field types, etc.), you need to generate a migration to reflect these changes in the database.
   - **Example**:

     ```
     # models.py
     from django.db import models

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

     - Run `python manage.py makemigrations` to create a migration file for this new model.
     - Run `python manage.py migrate` to apply this migration and create the `Book` table in the database.
2. **Changing Field Types or Adding Constraints:**

Khushal Savalakha

- **When to Use**: When you change the type of a field or add constraints like `unique` or `null`.
- **Example**:

```
# Changing a field type in models.py
class Book(models.Model):
    title = models.TextField()  # Changed from CharField to TextField
    author = models.CharField(max_length=100)
```

  - Run `python manage.py makemigrations` to generate the migration for this change.
  - Run `python manage.py migrate` to apply the migration.

3. **Renaming or Removing Fields:**
   - **When to Use**: When you rename or remove fields in your models.
   - **Example**:

```
# Renaming a field in models.py
class Book(models.Model):
    name = models.CharField(max_length=100)  # Renamed from 'title'
    author = models.CharField(max_length=100)
```

  - Run `python manage.py makemigrations` to create a migration that reflects this change.
  - Run `python manage.py migrate` to apply the changes.

4. **Creating or Modifying Database Tables:**
   - **When to Use**: When you need to create new tables or modify existing ones based on model changes.
   - **Example**:

```
# models.py
class Publisher(models.Model):
    name = models.CharField(max_length=100)
    established = models.DateField()
```


Khushal Savalakha

- Run `python manage.py makemigrations` to create the migration for the `Publisher` model.
- Run `python manage.py migrate` to create the `Publisher` table.

Workflow Example

1. **Define Models:**

```
# models.py
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)
    birth_date = models.DateField()
```

2. **Generate Migrations:**
   - Run `python manage.py makemigrations` to create a migration file that includes the schema changes for the `Author` model.
3. **Apply Migrations:**
   - Run `python manage.py migrate` to apply the migration and create the `Author` table in the database.
4. **Modify Models:**

```
# Adding a new field
class Author(models.Model):
    name = models.CharField(max_length=100)
    birth_date = models.DateField()
    nationality = models.CharField(max_length=50)  #
New field
```

5. **Generate and Apply New Migrations:**
   - Run `python manage.py makemigrations` to generate a migration file for the new `nationality` field.
   - Run `python manage.py migrate` to apply this migration and update the `Author` table schema.

Combining with Forms

Khushal Savalakha

When using forms, changes in models often necessitate migrations:

- **Example**: If you add a new field to a model that is used in a form, you'll need to make and apply migrations to update the database schema before using the form.

```
# forms.py
from django import forms
from .models import Author

class AuthorForm(forms.ModelForm):
    class Meta:
        model = Author
        fields = ['name', 'birth_date', 'nationalit
y']  # Include the new field
```

Summary

- `makemigrations`: Create migration files that record changes to your models.
- `migrate`: Apply migration files to update the database schema.

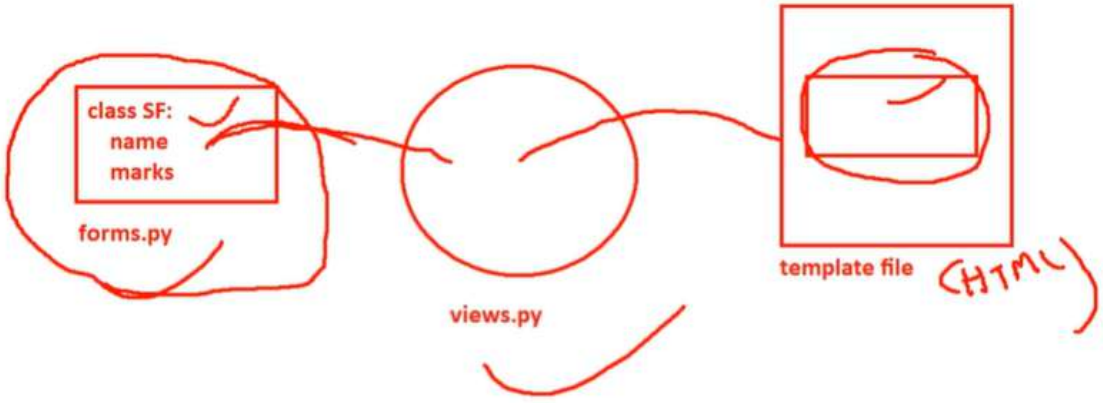
These commands are essential for keeping your database schema in sync with your Django models, ensuring that your application runs smoothly with the expected data structure.

▼ Working With Django Forms:

Khushal Savalakha

```
model class--->converted into database table
form class--->converted into html form

Process to generate Django forms:
---------------------------------

models.py-->
class Employee(models.Model):
    name = models.CharField(max_length=30)
    marks = models.IntegerField()
```

```
models.py-->
class Employee(models.Model):
    name = models.CharField(max_length=30)
    marks = models.IntegerField()

1. Creation of forms.py file in our application folder with our required fields.


from django import forms
class StudentForm(forms.Form):
    name = forms.CharField()
    marks = forms.IntegerField()
```



1.Creation of forms.py i n our application folder with our required fields.

```
forms.py:
---------
from django import forms
class StudentForm(forms.Form):
    name = forms.CharField()
    marks = forms.IntegerField()

Note: name,marks are the field names which will be available in html form.
```


Khushal Savalakha

```
step-2: usage of forms.py inside views.py file:
-----------------------------------------------
views.py is responsible to send this form to the template html file.

def studentinputview(request):
    form = StudentForm()
    my_dict = {'form': form}
    return render(request,'testapp/input.html',context=my_dict)
    #return render(request,'testapp/input.html',my_dict)
    #return render(request,'testapp/input.html',{'form': form})
```

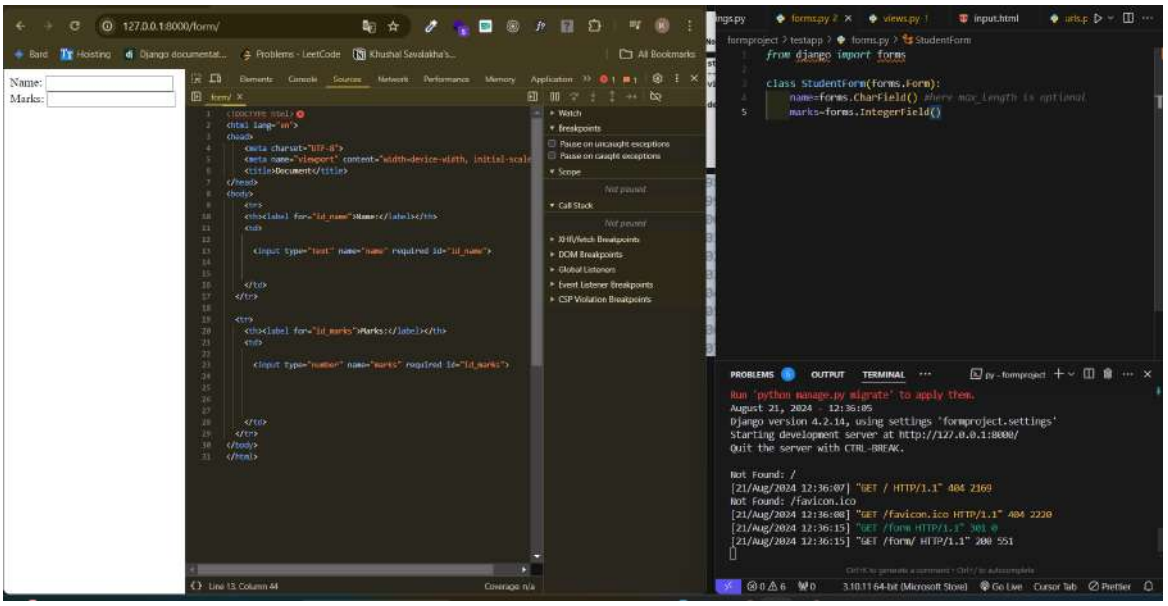
```
step-3:
-------
input.html:
-----------

<form method='post'>
    {{form}}
</form>
```

Here below code is automatically generated

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initi
    <title>Document</title>
</head>
<body>
    <tr>
    <th><label for="id_name">Name:</label></th>
    <td>

      <input type="text" name="name" required id="id_name">



    </td>
  </tr>
```


Khushal Savalakha

```
  <tr>
    <th><label for="id_marks">Marks:</label></th>
    <td>

      <input type="number" name="marks" required id="id_mark

    </td>
  </tr>
</body>
</html>
```



Instes of writing {{form}}

```
<!DOCTYPE html>
{%  load static %}
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initi
    <title>Document</title>
    <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dis
      rel="stylesheet"
      integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr
```

```
      crossorigin="anonymous"
    />
    <link rel="stylesheet" href="{% static 'css/input.css' %
  </head>
  <body>
    <h1>Student Input Form</h1>
    <div class="container" align="center" >
        <form method="post">
             {{form.as_p}}
            <input type="submit" name="" class="btn btn-lg b
        </form>
    </div>
  </body>
</html>
```

```
<!DOCTYPE html>

<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initi
    <title>Document</title>
    <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dis
      rel="stylesheet"
      integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr
      crossorigin="anonymous"
    />
    <link rel="stylesheet" href="/static/css/input.css">
  </head>
  <body>
    <h1>Student Input Form</h1>
    <div class="container" align="center" >
        <form method="post">
             <p>
    <label for="id_name">Name:</label>
    <input type="text" name="name" required id="id_name">
```


Khushal Savalakha

```
  </p>


  <p>
    <label for="id_marks">Marks:</label>
    <input type="number" name="marks" required id="id_marks'
  </p>
            <input type="submit" name="" class="btn btn-lg b
        </form>
    </div>
  </body>
</html>
```

💡 Cross Site Request Forgery:

Cross-Site Request Forgery (CSRF) is a type of security vulnerability where an attacker tricks a user into performing actions on a web application in which they are authenticated, without their knowledge or consent. These actions are typically executed in the context of the victim's session with the vulnerable web application.

How CSRF Works:

1. **Victim Authentication:**
   - The victim logs into a web application (e.g., a banking site) and their browser receives an authentication cookie.
2. **Malicious Request:**
   - The attacker crafts a malicious request that appears to be a legitimate request to the web application. This could be something like a request to transfer money, change account details, etc.
3. **Tricking the Victim:**
   - The attacker tricks the victim into making the request. This could happen if the victim clicks on a malicious link, visits a malicious website, or is tricked into submitting a form with the crafted request.
4. **Automatic Request Execution:**

Khushal Savalakha

- Since the victim is already authenticated (due to the existing session cookie), the web application processes the request as if it were legitimate, without the victim's knowledge.

5. **Unintended Action:**
   - The action is performed in the context of the victim's session, causing unintended effects, such as transferring money to the attacker's account.

CSRF Example:

Let's say you are logged into your bank's website (e.g., `bank.com`) and your session is active.

1. **Crafted Malicious Request:**
   - The attacker creates a malicious website with a hidden form that submits a request to `bank.com` to transfer money:

```
<form action="<https://bank.com/transfer>" method="POS
T">
    <input type="hidden" name="account" value="attacke
r_account">
    <input type="hidden" name="amount" value="1000">
</form>

<script>
    document.forms[0].submit(); // Automatically submi
ts the form
</script>
```

2. **Victim Visits Malicious Site:**
   - You (the victim) visit this malicious site while still logged into your bank account in another browser tab.
3. **Request Sent to Bank:**
   - The form is automatically submitted by the malicious site, sending the request to `bank.com`.
4. **Bank Processes the Request:**
   - Since your session is active, the bank processes the transfer as if you had initiated it, transferring money to the attacker's account.

Khushal Savalakha

CSRF Protection in Django:

Django includes built-in protection against CSRF attacks using CSRF tokens.

- **CSRF Tokens:** When a form is rendered in a Django template, Django automatically generates a CSRF token, which is a unique value for that session. This token is included in the form as a hidden field:

```
<form method="POST">
    {% csrf_token %}
    <!-- Other form fields -->
</form>
```

- **Validation:** When the form is submitted, Django checks that the submitted CSRF token matches the one that was generated and stored in the user's session. If the tokens don't match, the request is denied.
- **CSRF Middleware:** Django's CSRF protection is implemented via middleware that checks the presence and validity of CSRF tokens in all POST requests.

How CSRF Protection Works:

1. **User Requests a Form:** The user accesses a page with a form. Django includes a hidden CSRF token in the form.
2. **Form Submission:** When the user submits the form, the CSRF token is sent back to the server with the form data.
3. **Validation:** Django compares the received CSRF token with the token stored in the session. If they match, the request is allowed; if not, it is rejected.

Summary:

- **CSRF** is a type of attack where an attacker tricks a user into making a request on a web application where they are authenticated, causing unintended actions.
- **Django's CSRF protection** uses tokens to ensure that the request is genuinely coming from the authenticated user, preventing such attacks.

By using Django's built-in CSRF protection (enabled by default), you can protect your web application from these kinds of attacks.

```
<!DOCTYPE html>
{%  load static %}
<html lang="en">
```


Khushal Savalakha

```
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initi
    <title>Document</title>
    <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dis
      rel="stylesheet"
      integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr
      crossorigin="anonymous"
    />
    <link rel="stylesheet" href="{% static 'css/input.css' %
  </head>
  <body>
    <h1>Student Input Form</h1>
    <div class="container" align="center" >
        <form method="post">
             {{form.as_p}}
             {% csrf_token %}
            <input type="submit" name="" class="btn btn-lg b
        </form>
    </div>
  </body>
</html>
```

💡 below is pagesource:

```
<!DOCTYPE html>

<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initi
    <title>Document</title>
    <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dis
```


Khushal Savalakha

```
      rel="stylesheet"
      integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr
      crossorigin="anonymous"
    />
    <link rel="stylesheet" href="/static/css/input.css">
  </head>
  <body>
    <h1>Student Input Form</h1>
    <div class="container" align="center" >
        <form method="post">
             <p>
    <label for="id_name">Name:</label>
    <input type="text" name="name" required id="id_name">

  </p>

  <p>
    <label for="id_marks">Marks:</label>
    <input type="number" name="marks" required id="id_marks'
  </p>
  <p>
    <label for="id_age">Age:</label>
    <input type="number" name="age" required id="id_age">
  </p>
             <input type="hidden" name="csrfmiddlewaretoken'
            <input type="submit" name="" class="btn btn-lg b
        </form>
    </div>
  </body>
</html>
```

💡 it means when some malicious request comes to user site if it will got to that site and so some actions like transfer many if there is a form with Django frame work or other framework CSRF token can not be same if any malicious activity is present am i true ?

Yes, you're on the right track. Here's a more detailed explanation:

Khushal Savalakha

How CSRF Tokens Protect Against Malicious Requests:

1. **Unique Token for Each User Session:**
   - When a user visits a site built with Django (or another framework with CSRF protection), a unique CSRF token is generated and stored in the user's session. This token is also included in any form that the server renders for the user.
2. **Token Verification on Form Submission:**
   - When the user submits a form, this CSRF token is sent along with the form data. The server then checks if the token received in the form submission matches the token stored in the user's session.
3. **Blocking Malicious Requests:**
   - **Malicious Request Without a Valid Token:** If an attacker tries to trick the user into making a request to the site (like transferring money), the request will likely not have the correct CSRF token, or no token at all.
   - **Token Mismatch:** Even if the malicious site attempts to guess the token or somehow include one, it won’t match the unique token stored in the user’s session on the server.
   - **Request Denied:** When Django checks the token and sees that it doesn't match (or is missing), it will reject the request, preventing the malicious action from being executed.

Summary:

- **CSRF Token Mismatch:** If a malicious site tries to perform an action on behalf of the user, it will not have the correct CSRF token.
- **Request Rejection:** Since the CSRF token won’t match the one stored in the user’s session, Django will reject the request, preventing any unauthorized or harmful actions.

Example in Action:

- **Legitimate Action:**
  - You log into your banking site, and a form to transfer money includes a hidden CSRF token.
  - When you submit the form, the CSRF token is sent and verified by the server, allowing the transaction to proceed.
- **Malicious Attempt:**

Khushal Savalakha

- An attacker tricks you into visiting a malicious site that tries to submit a money transfer request to your bank.
- The request from the malicious site does not have the correct CSRF token, or no token at all.
- Django detects this and rejects the request, ensuring that the malicious action cannot be performed.

So yes, if any malicious activity tries to send a forged request, the CSRF token in Django (or other frameworks with CSRF protection) ensures that the request will be blocked if the token is incorrect or missing. This is a key defense mechanism against CSRF attacks.

> 💡 CSRF token is generated and stored in the user's session.
> but if this session data has CSRF token then some one can also access then how it will safe ?

Your concern about the security of the CSRF token stored in the session is valid, but it's important to understand how web browsers and Django manage sessions and CSRF tokens to ensure security.

Key Points of CSRF Token Security:

1. **Same-Origin Policy:**
   - Web browsers enforce the "Same-Origin Policy," which means that a web page can only make requests and access resources from the same domain that served it. This policy prevents a malicious site from reading or tampering with data from another site, including session data and CSRF tokens.
2. **CSRF Tokens and Session Security:**
   - The CSRF token is stored on the server side in the user’s session data, and the browser doesn't directly store or access it. When a form is rendered, the CSRF token is included as a hidden input in the form or as a header in AJAX requests. The token is then submitted back to the server, where it is validated against the token stored in the session.
3. **Session Hijacking Prevention:**
   - **Cookies and Secure Flags:** Cookies used to manage sessions (like the session cookie that stores the session ID) can be marked as `HttpOnly` and `Secure`. This prevents client-side scripts from accessing the cookie (`HttpOnly`) and ensures that

Khushal Savalakha

cookies are only sent over HTTPS (`Secure`), reducing the risk of session hijacking.

- **SameSite Attribute:** Cookies can also be marked with the `SameSite` attribute, which controls whether cookies are sent with cross-site requests. Setting `SameSite=Lax` or `SameSite=Strict` helps prevent the CSRF token from being sent with cross-origin requests, further mitigating the risk of CSRF attacks.

4. **Encrypted Tokens and Sessions:**
   - **Encryption:** Django can encrypt session data, including the CSRF token, using its built-in session framework. This adds another layer of security, making it difficult for attackers to tamper with or read session data.
   - **Django Middleware:** The CSRF middleware in Django automatically handles the generation and verification of CSRF tokens, ensuring that tokens are securely managed and validated.
5. **Regular Token Regeneration:**
   - **Token Expiry:** CSRF tokens are typically regenerated on a regular basis or on certain actions (like login) to minimize the window of time in which an attacker could potentially exploit a leaked token.

Addressing the Concern:

- **Attacker Access to CSRF Token:**
  - For an attacker to exploit a CSRF vulnerability, they would need to trick the user into performing an action without the user's knowledge. However, they wouldn't be able to access the CSRF token directly due to the Same-Origin Policy, which prevents a malicious site from accessing data on another domain.
  - If an attacker somehow obtains the CSRF token (which is difficult), they still need to be able to access the session data and have the user's session active, which is usually protected by the session management strategies mentioned above.
- **Session Management and Secure Storage:**
  - Django securely stores session data server-side and manages the CSRF tokens using best practices, which makes it difficult for an attacker to tamper with or access them.

Summary:

The CSRF token is safe because:

Khushal Savalakha

1. **Same-Origin Policy** prevents malicious sites from accessing it directly.
2. **Session Management** ensures that the token is stored securely and that sessions are protected.
3. **Encryption and Secure Cookies** make it difficult for an attacker to hijack sessions or tamper with CSRF tokens.
4. **Django's CSRF Middleware** handles tokens securely and ensures they are used correctly.

By following these security practices, Django makes it very challenging for an attacker to exploit CSRF vulnerabilities, even if they try to access or manipulate session data.

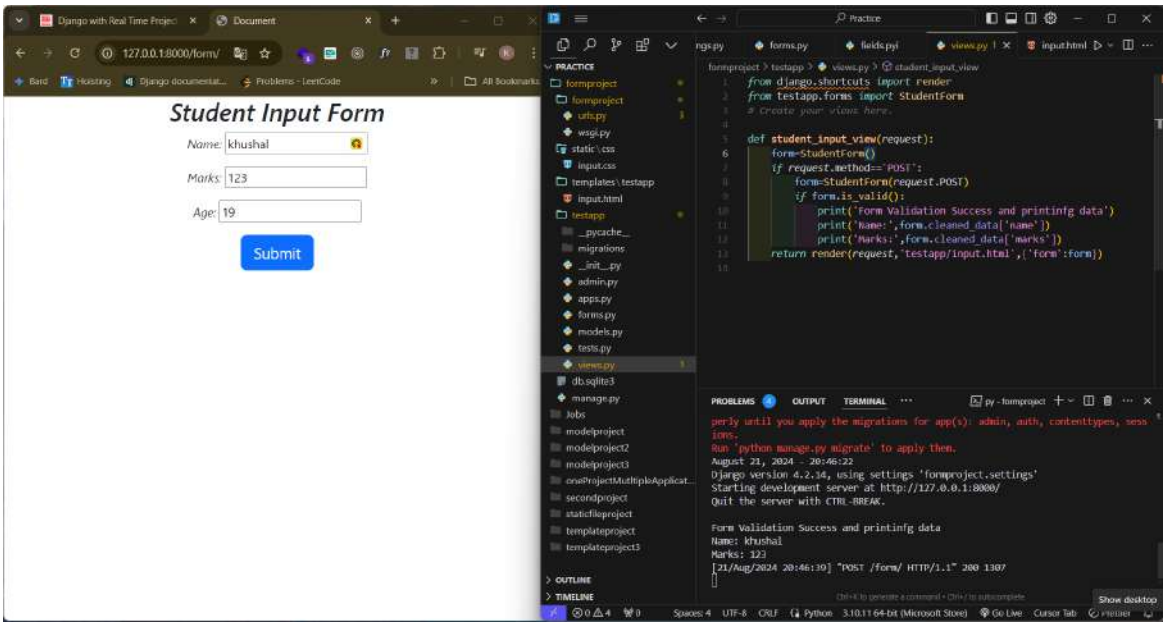
```
<input type="hidden" name="csrfmiddlewaretoken"
value="NylUTqgvW745N7DgQQXymv6YldLjIY0LySdfVUX5ta43rZuhvdEW
```

```
3101 ----------------------------------------------
3102 How to generate django form?
3103 forms.py--->views.py--->template file
3104
3105 What is the need of csrf_token?
3106   web security purposes
3107
3108 csrf_token is mandatory.
3109
```

```
3117 If we add csrf_token then in the generated form the following hidden field will
     added, which makes our post request secure.
3118
3119 <input type="hidden" name="csrfmiddlewaretoken"
     value="HdOq7HC7WNJI4KKEgvu48wpl6Lw8B41q6m8Zol4meHR22YJnfROpHgKhLmAxDgn3">
3120
3121 The value of this hidden field is keep on changing from request to request. Hence
     it is impossible to forgery our request.
3122
3123 If we configure csrf_token then only django accepts our form.
```

```
3125 How to process input data from the form inside views.py file:
3126 ------------------------------------------------------------
3127 Inside views.py, we have to read data provided by end user and we have to use that
     data based on our requirement.
3128
3129 form = StudentForm()--->empty form object to display form to the end user
3130 form = StudentForm(request.POST)--->This form object contains end user provided
     data
3131
```

```
cleaned_data --->dictionary which contains end user provided data
form.cleaned_data['name']--->The name entered by end user
form.cleaned_data['marks']--->The name entered by end user

form.is_valid() --->To check whether validations are successful or not
```




```
<!DOCTYPE html>
{%  load static %}
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initi
    <title>Document</title>
    <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dis
      rel="stylesheet"
      integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr
      crossorigin="anonymous"
    />
    <link rel="stylesheet" href="{% static 'css/input.css' %
  </head>
  <body>
    <h1>Student Input Form</h1>
    <div class="container" align="center" >
        <form method="post">
            {{form.as_p}}
```


Khushal Savalakha

```
                {% csrf_token %}
               <input type="submit" name="" class="btn btn-lg b
          </form>
      </div>
    </body>
</html>
```

```
from django import forms

class StudentForm(forms.Form):
    name=forms.CharField() #here max_length is optional
    marks=forms.IntegerField()
    age=forms.IntegerField()
```

```
from django.shortcuts import render
from testapp.forms import StudentForm
# Create your views here.

def student_input_view(request):
    form=StudentForm()
    if request.method=='POST':
        form=StudentForm(request.POST)
        if form.is_valid():
            print('Form Validation Success and printinfg dat
            print('Name:',form.cleaned_data['name'])
            print('Marks:',form.cleaned_data['marks'])
    return render(request,'testapp/input.html',{'form':form}
```

💡 i am asking that first when we request student_input_view(request) on that time it will assign value to form = StudentForm() then it goes to if condition here when request comes it is in the form of get so if condition is false so it will move to return statement in which we are sending input.html so we are out of student_input_view function so how it show me printed data in console i don't understand this please explain this.

Understanding the Flow

Khushal Savalakha

1. **Initial Request (GET Request):**
   - When you first access the page that triggers `student_input_view(request)`, it's a GET request.
   - The view creates an empty form instance with `form = StudentForm()`.
   - Since this is a GET request, the condition `if request.method == 'POST':` is **not** met.
   - The view then directly proceeds to the `return render(request, 'testapp/input.html', {'form': form})` line.
   - The `input.html` template is rendered with the empty form and returned to the browser.
   - No data is printed to the console because the code inside the `if form.is_valid()` block is never executed during this GET request.
2. **Form Submission (POST Request):**
   - After the user fills out the form and submits it, the browser sends a POST request to the same view (`student_input_view(request)`).
   - The view now re-assigns the `form` variable with `form = StudentForm(request.POST)`, which populates the form with the submitted data.
   - This time, the condition `if request.method == 'POST':` is met because it's a POST request.
   - The view then checks if the form is valid with `if form.is_valid():`. If the form is valid, it executes the block of code inside this condition.
   - **This is where the data is printed to the console**:
     - `print('Name:', form.cleaned_data['name'])`
     - `print('Marks:', form.cleaned_data['marks'])`
   - After printing, the view continues to the `return render(request, 'testapp/input.html', {'form': form})` statement, rendering the `input.html` template again. This time, the form is pre-filled with the submitted data.

Key Point

- **Printed Data on Console:** The `print()` statements only execute when the form is submitted via a POST request and the form is valid. They do not execute during the initial GET request.

Khushal Savalakha

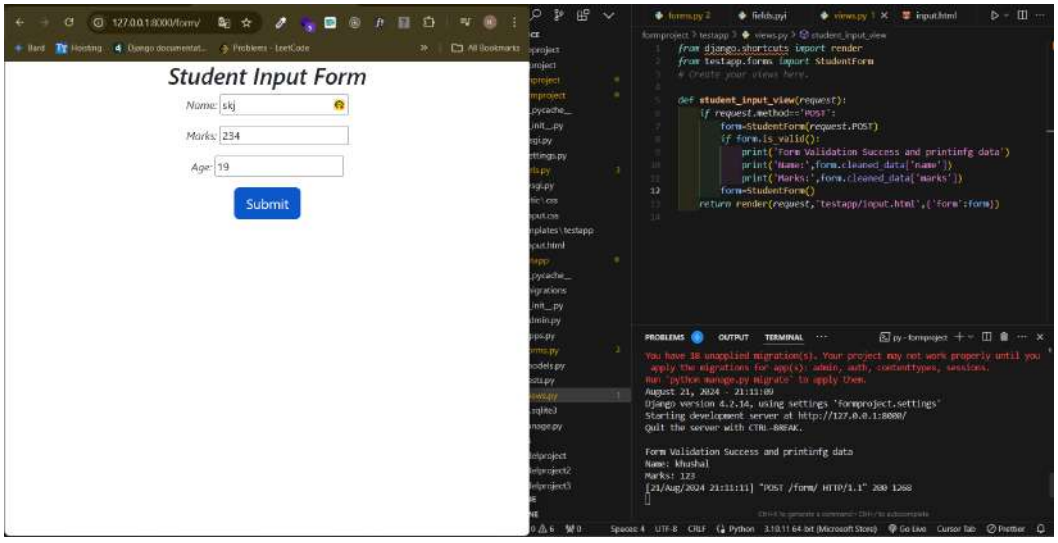
- **Request Flow:** The request handling is sequential. The initial GET request does not trigger the `print()` statements because the form validation code is not reached. The printed data appears in the console only after the form is submitted and the view processes the POST request.

Recap

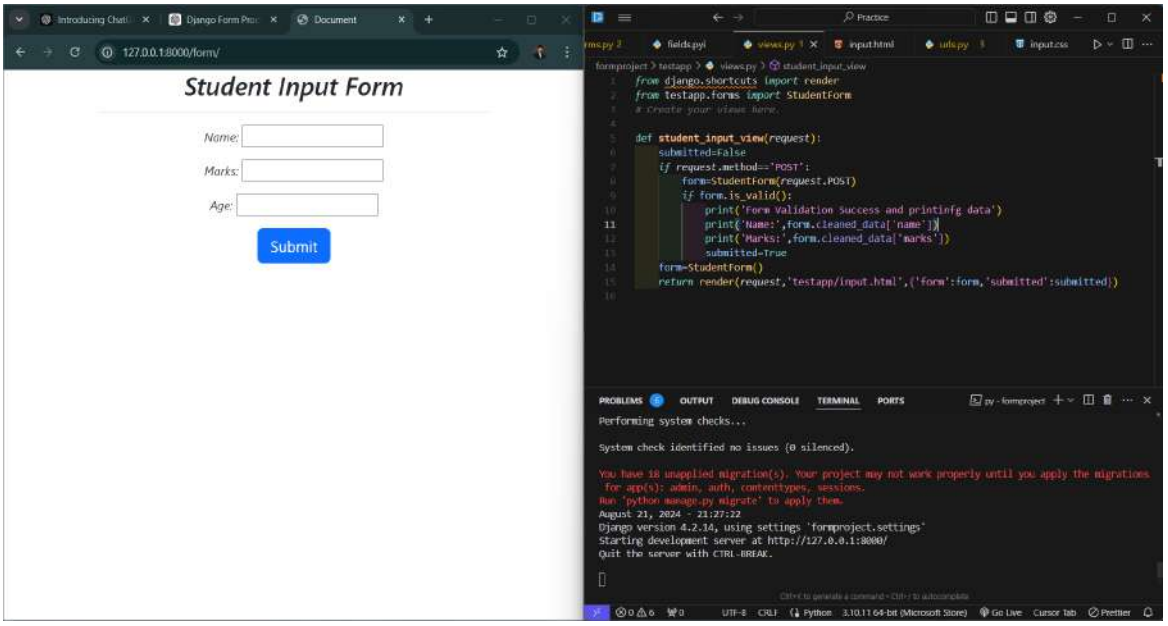
- **GET Request:** Renders the form without any console output.
- **POST Request:** Processes the submitted form, validates it, and if valid, prints data to the console.

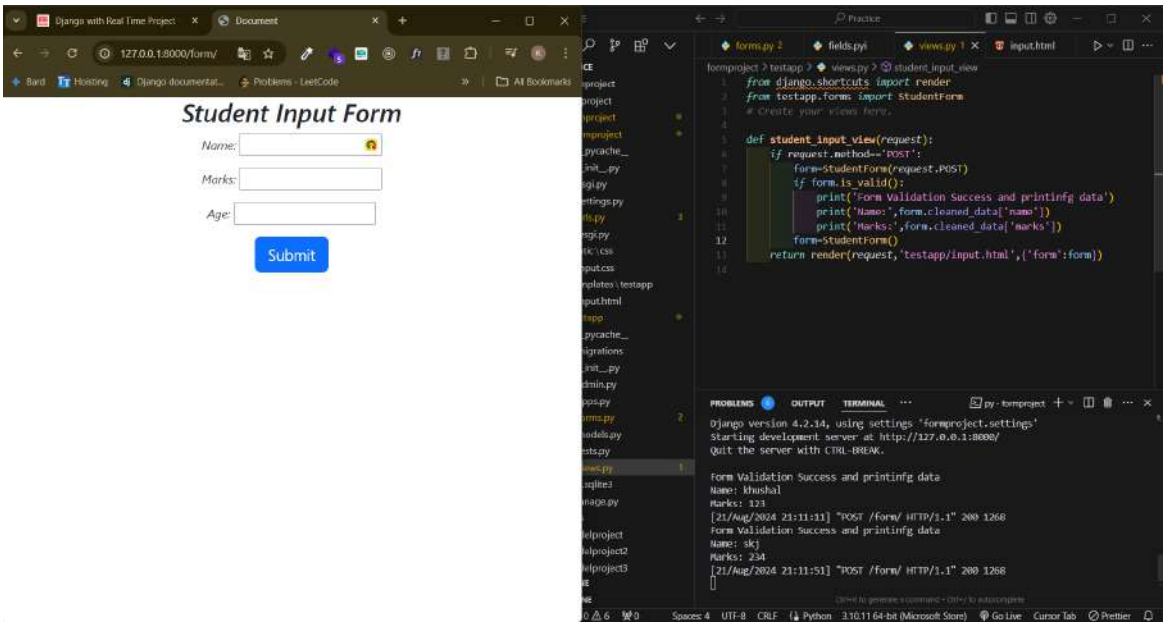
I hope this clarifies how the `print()` statements are executed and why you see the printed data in the console only after submitting the form.

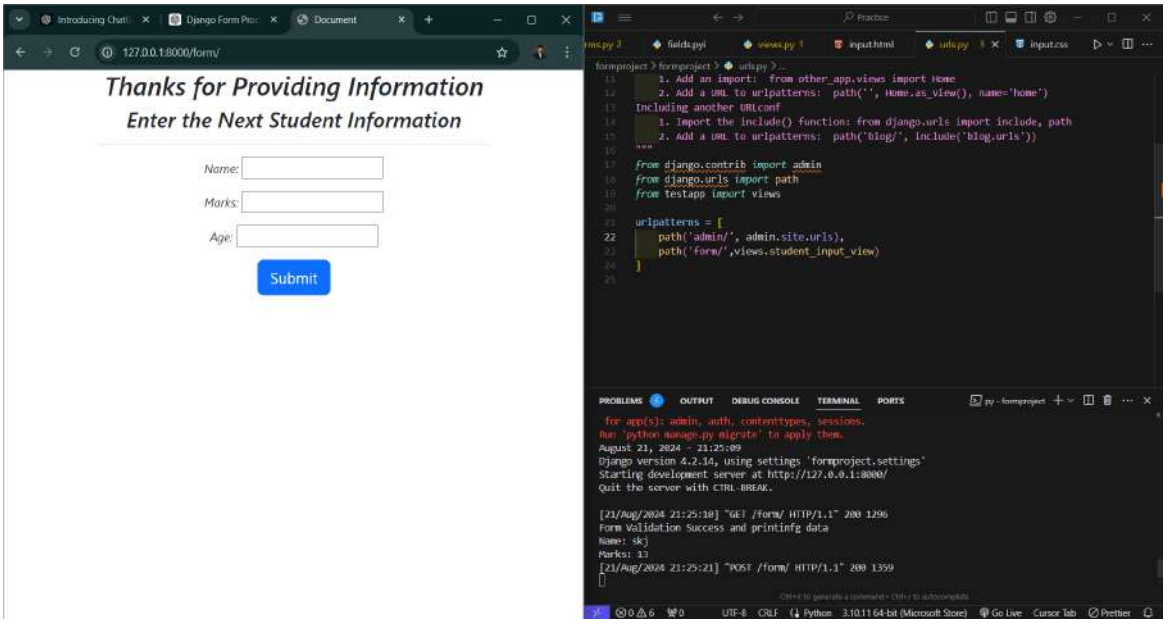
```
August 21, 2024 - 20:46:22
Django version 4.2.14, using settings 'formproject.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.

Form Validation Success and printinfg data
Name: khushal
Marks: 123
[21/Aug/2024 20:46:39] "POST /form/ HTTP/1.1" 200 1307
```

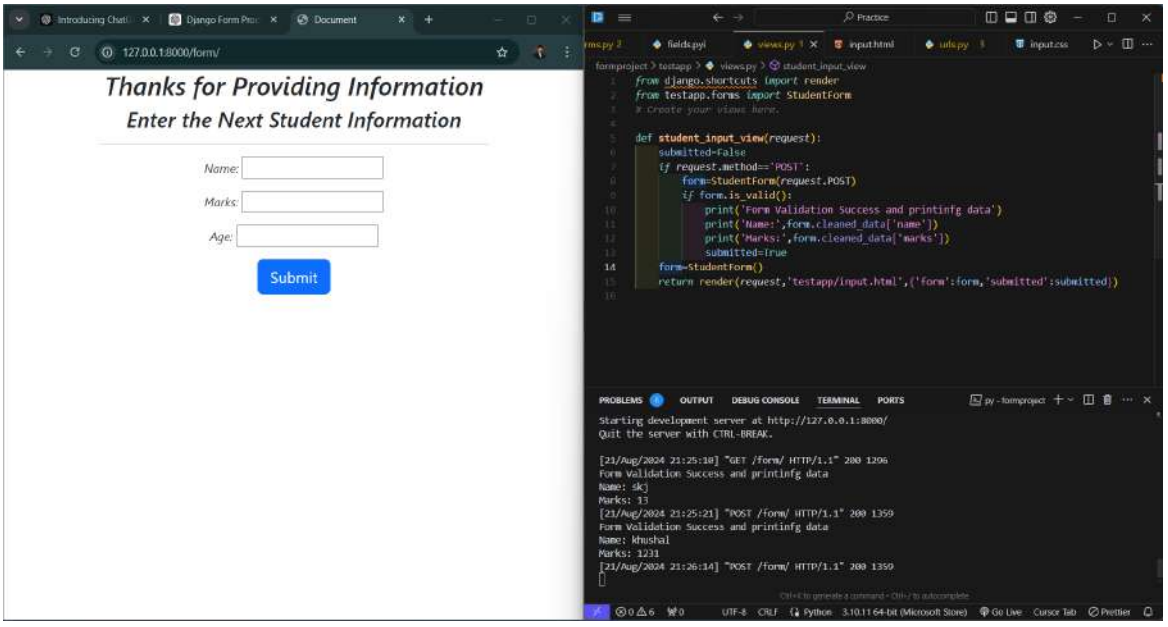


after submitting it does not show old data


Introducing ChatG
Django Form Proc
Document
127.0.0.1:8000/form/
Student Input Form
Name:
Marks:
Age:
Submit
Practice
fields.pyi
views.py 1
input.html
urls.py
input.css
formproject > testapp > views.py > student_input_view
from django.shortcuts import render
from testapp.forms import StudentForm
# Create your views here.
def student_input_view(request):
submitted=False
if request.method=='POST':
form=StudentForm(request.POST)
if form.is_valid():
print('Form Validation Success and printinfg data')
print('Name:',form.cleaned_data['name'])
print('Marks:',form.cleaned_data['marks'])
submitted=True
form=StudentForm()
return render(request,'testapp/input.html',{'form':form,'submitted':submitted})
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
Performing system checks...
System check identified no issues (0 silenced).
You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations
for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
August 21, 2024 - 21:27:22
Django version 4.2.14, using settings 'formproject.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.


Django with Real Time Project
Document
127.0.0.1:8000/form/
Student Input Form
Name:
Marks:
Age:
Submit
forms.py 2
fields.pyi
views.py 1
input.html
formproject > testapp > views.py > student_input_view
from django.shortcuts import render
from testapp.forms import StudentForm
# Create your views here.
def student_input_view(request):
if request.method=='POST':
form=StudentForm(request.POST)
if form.is_valid():
print('Form Validation Success and printinfg data')
print('Name:',form.cleaned_data['name'])
print('Marks:',form.cleaned_data['marks'])
form=StudentForm()
return render(request,'testapp/input.html',{'form':form})
PROBLEMS
OUTPUT
TERMINAL
Django version 4.2.14, using settings 'formproject.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
Form Validation Success and printinfg data
Name: khushal
Marks: 123
[21/Aug/2024 21:11:11] "POST /form/ HTTP/1.1" 200 1268
Form Validation Success and printinfg data
Name: skj
Marks: 234
[21/Aug/2024 21:11:51] "POST /form/ HTTP/1.1" 200 1268






```
<!DOCTYPE html>
{%  load static %}
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initi
    <title>Document</title>
    <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dis
      rel="stylesheet"
```

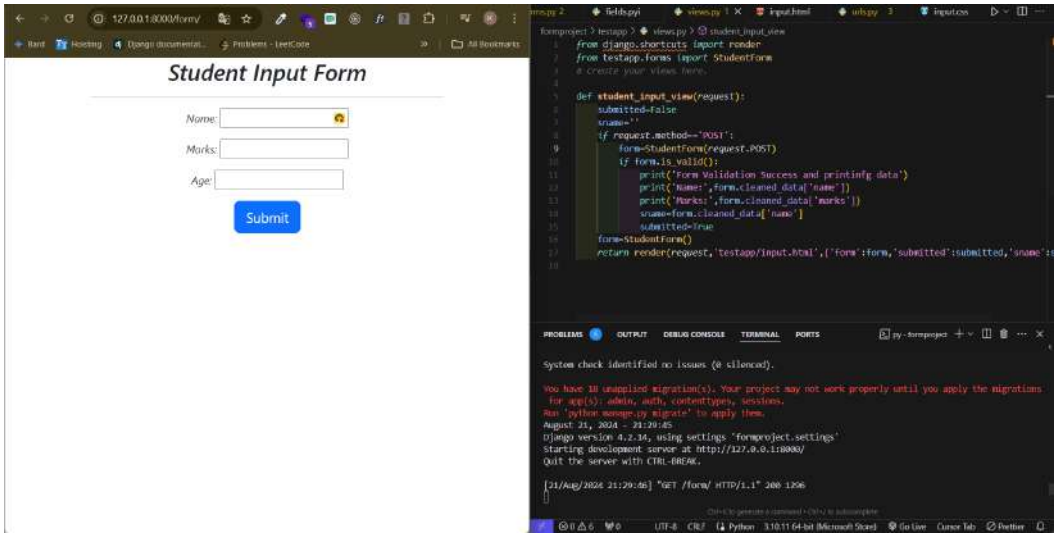
```
      integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr
      crossorigin="anonymous"
    />
    <link rel="stylesheet" href="{% static 'css/input.css' %
  </head>
  <body>
    <div class="container" align="center" >
      {% if submitted %}
      <h1>Thanks for Providing Information of {{sname}}</h1>
      <h2>Enter the Next Student Information</h2>
      {% else %}
       <h1>Student Input Form</h1>
      {% endif %}
      <hr>
        <form method="post">
             {{form.as_p}}
             {% csrf_token %}
            <input type="submit" name="" class="btn btn-lg b
        </form>
    </div>
  </body>
</html>
```

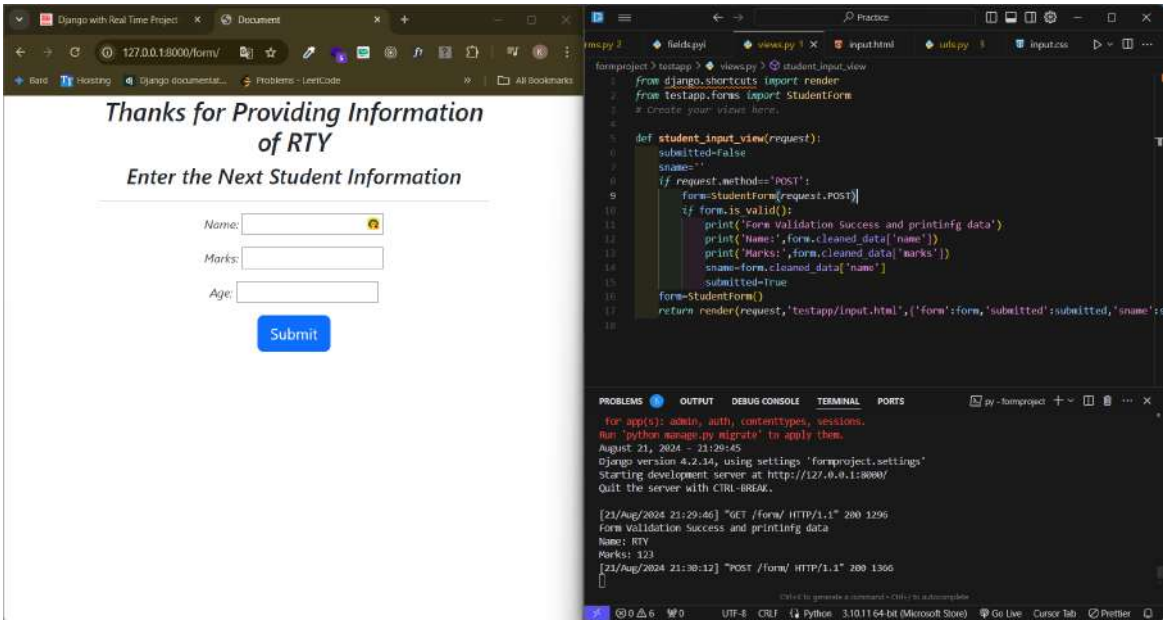
```
from django.shortcuts import render
from testapp.forms import StudentForm
# Create your views here.

def student_input_view(request):
    submitted=False
    sname=''
    if request.method=='POST':
        form=StudentForm(request.POST)
        if form.is_valid():
            print('Form Validation Success and printinfg dat
            print('Name:',form.cleaned_data['name'])
            print('Marks:',form.cleaned_data['marks'])
            sname=form.cleaned_data['name']
            submitted=True
```


Khushal Savalakha

```
    form=StudentForm()
    return render(request,'testapp/input.html',{'form':form,
```

By below example we can see when we submit the data post reuest send to student_input_view by which function is again executed by this we are assining form=StudentForm(request.POST) so we are sending data of previous data .





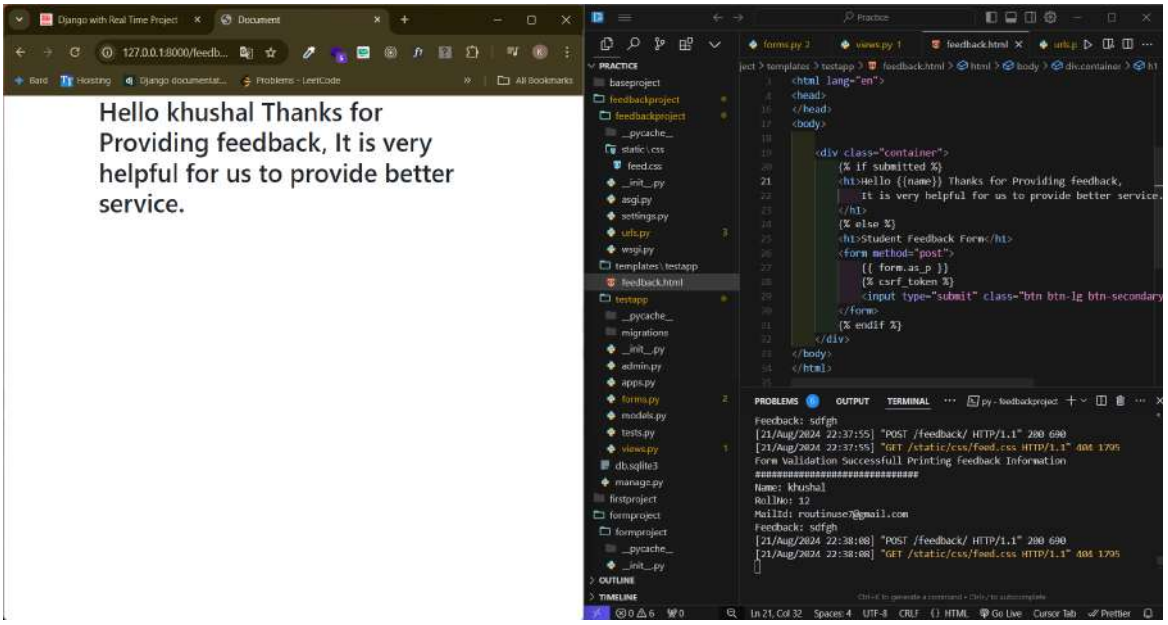

```
from django.shortcuts import render
from testapp.forms import FeedBackForm
# Create your views here.

def feedback_view(request):
```

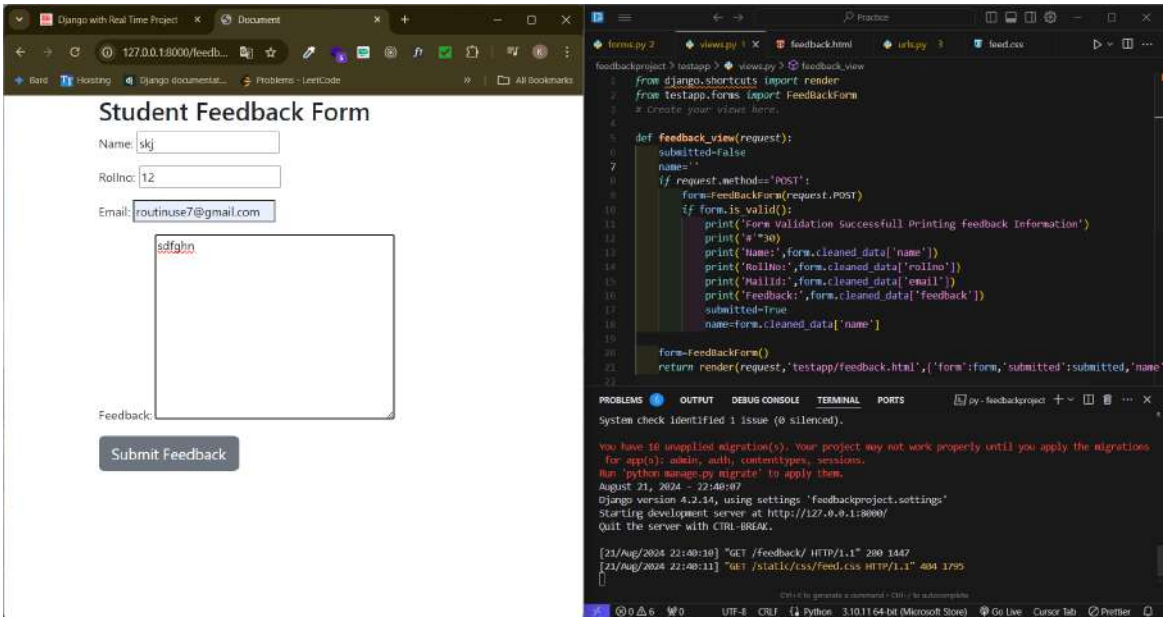
```
    submitted=False
    name=''
    if request.method=='POST':
        form=FeedBackForm(request.POST)
        if form.is_valid():
            print('Form Validation Successfull Printing feed
            print('#'*30)
            print('Name:',form.cleaned_data['name'])
            print('RollNo:',form.cleaned_data['rollno'])
            print('MailId:',form.cleaned_data['email'])
            print('Feedback:',form.cleaned_data['feedback'])
            submitted=True
            name=form.cleaned_data['name']

    form=FeedBackForm()
    return render(request,'testapp/feedback.html',{'form':fo
```

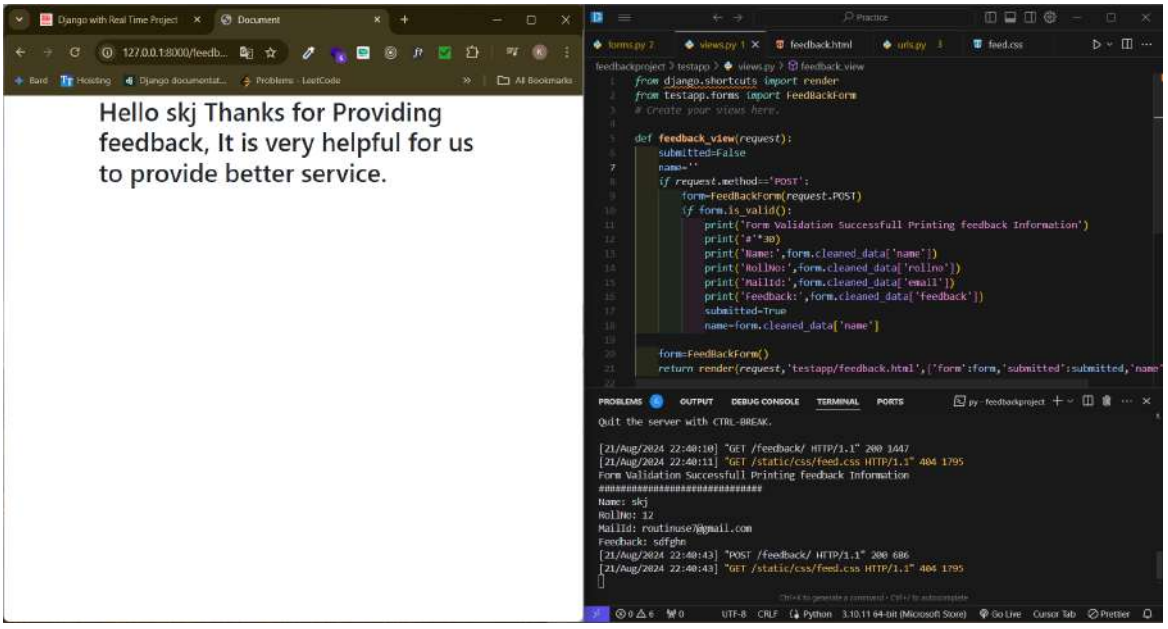
```
<!DOCTYPE html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initi
    <title>Document</title>
    <link
    href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/
    rel="stylesheet"
    integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5p
    crossorigin="anonymous"
  />

  <link rel="stylesheet" href="{% static 'css/feed.css' %}">
</head>
<body>

    <div class="container">
        {% if submitted %}
        <h1>Hello {{name}} Thanks for Providing feedback,
```


Khushal Savalakha

```
            It is very helpful for us to provide better serv
        </h1>
        {% else %}
        <h1>Student Feedback Form</h1>
        <form method="post">
            {{ form.as_p }}
            {% csrf_token %}
            <input type="submit" class="btn btn-lg btn-secon
        </form>
        {% endif %}
    </div>
</body>
</html>
```








```
Form Validations:
-----------------
It is very common requirement to implement validations for form paramters like

1. The feedback minimum length should be 200 characters
2. The password should contains minimum 8 characters
3. user email should be only gmail etc
etc
```

```
We can implement validation logic by using 2 ways:

1. Explicitly by the programmer by using clean methods
2. By using django inbuilt validators

*** Total validation logic should be written inside forms.py
```

here 2 ways is easy but 1st is more powerful.

```python
class Parent:
    def __init__(self):
        self.x=9999

    def property(self):
        print('gold+land+cash')

class Child(Parent):
    def education(self):
        print('B.Tech qualification+job')

c = Child()
c.education()
c.property()
print(c.x)
```

how Form stores the data.

```python
class Form:
    def __init__(self):
        self.cleaned_data={'name':'durga'}

class FeedBackForm(forms.Form):
    name = forms.CharField()
    rollno = forms.IntegerField()
    email = forms.EmailField()
    feedback = forms.CharField(widget=forms.Textarea)

form = FeedBackForm()
form.cleaned_data['name']
```

forms is a module name and Form is class.

Khushal Savalakha

```python
class Parent:
    def marry(self):
        print('marray appalamma')

class Child(Parent):
    def marry(self):
        print('marry katrina kaif')
        super().marry()

c = Child()
c.marry()
```

```
1. Explicitly by the programmer by using clean methods:
-------------------------------------------------------
syntax of clean method: clean_fieldname(self)

In the form class, for any field, if we define clean method, then at the time of
submit this form, django will call this method automcatically to perform
validations. If the clean method won't raise any error then only request will be
processed.
```

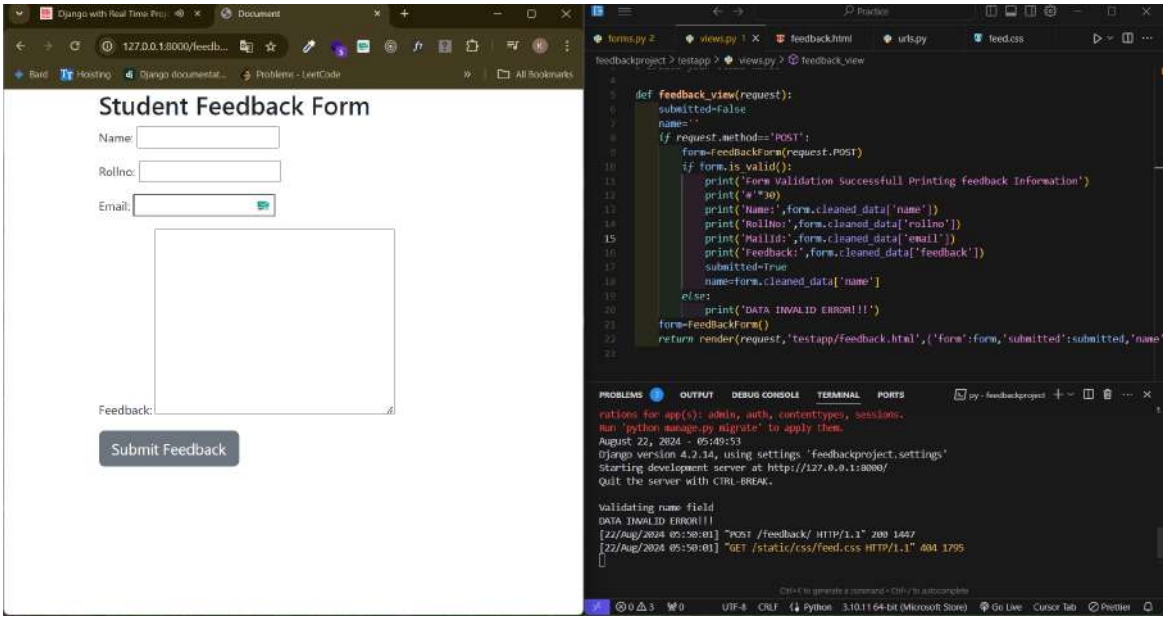
```python
class TooYoungException(BaseException):
    def __init__(self,msg):
        self.msg=msg

class TooOldException(BaseException):
    def __init__(self,msg):
        self.msg=msg

age = int(input('Enter Your Age:'))
if age >60:
    raise TooYoungException('Please wait some more time,definitely you will get best
elif age<18:
    raise TooOldException('Your age already crossed marriage age, no chance of gettin
else:
    print('Thanks for uploading your profile, will send match information soon')
```


```
1. Explicitly by the programmer by using clean methods:
-------------------------------------------------------
syntax of clean method: clean_fieldname(self)

In the form class, for any field, if we define clean method, then at the time of
submit this form, django will call this method automcatically to perform
validations. If the clean method won't raise any error then only request will be
processed.
```


```
from django import forms
```

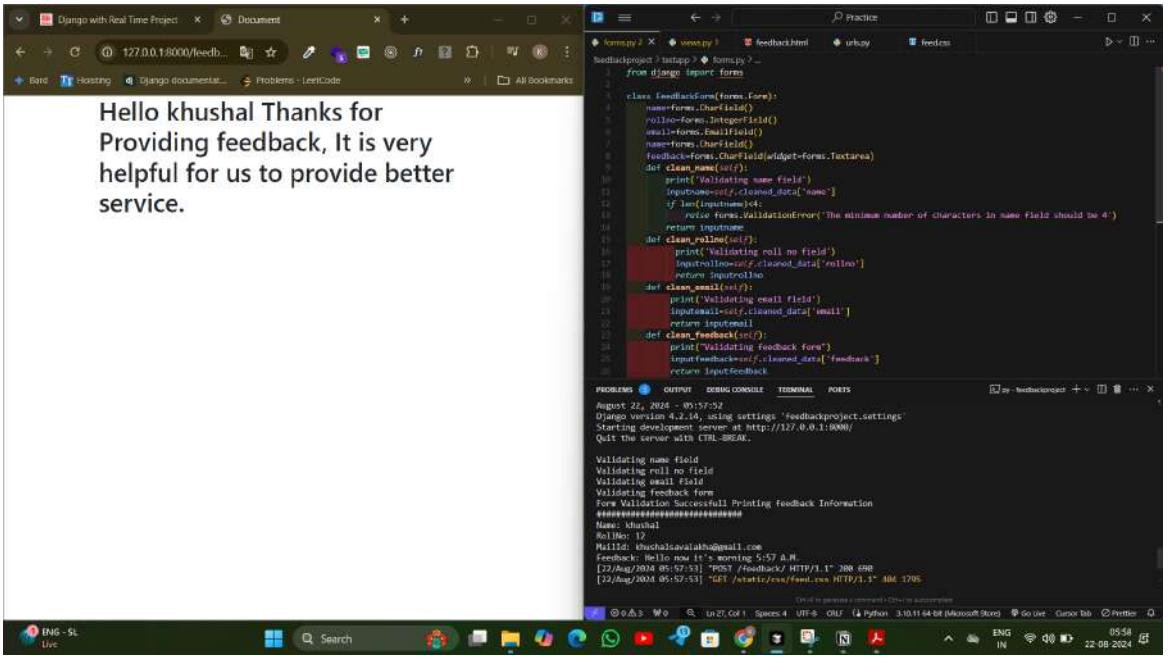

Khushal Savalakha

```
class FeedBackForm(forms.Form):
    name=forms.CharField()
    rollno=forms.IntegerField()
    email=forms.EmailField()
    name=forms.CharField()
    feedback=forms.CharField(widget=forms.Textarea)
    def clean_name(self):
        print('Validating name field')
        inputname=self.cleaned_data['name']
        if len(inputname)<4:
            raise forms.ValidationError('The minimum number
        return inputname
```




```
from django import forms

class FeedBackForm(forms.Form):
    name=forms.CharField()
    rollno=forms.IntegerField()
    email=forms.EmailField()
    name=forms.CharField()
    feedback=forms.CharField(widget=forms.Textarea)
    def clean_name(self):
        print('Validating name field')
        inputname=self.cleaned_data['name']
```

```
        if len(inputname)<4:
            raise forms.ValidationError('The minimum number
        return inputname
    def clean_rollno(self):
          print('Validating roll no field')
          inputrollno=self.cleaned_data['rollno']
          return inputrollno
    def clean_email(self):
         print('Validating email field')
         inputemail=self.cleaned_data['email']
         return inputemail
    def clean_feedback(self):
         print("Validating feedback form")
         inputfeedback=self.cleaned_data['feedback']
         return inputfeedback
```

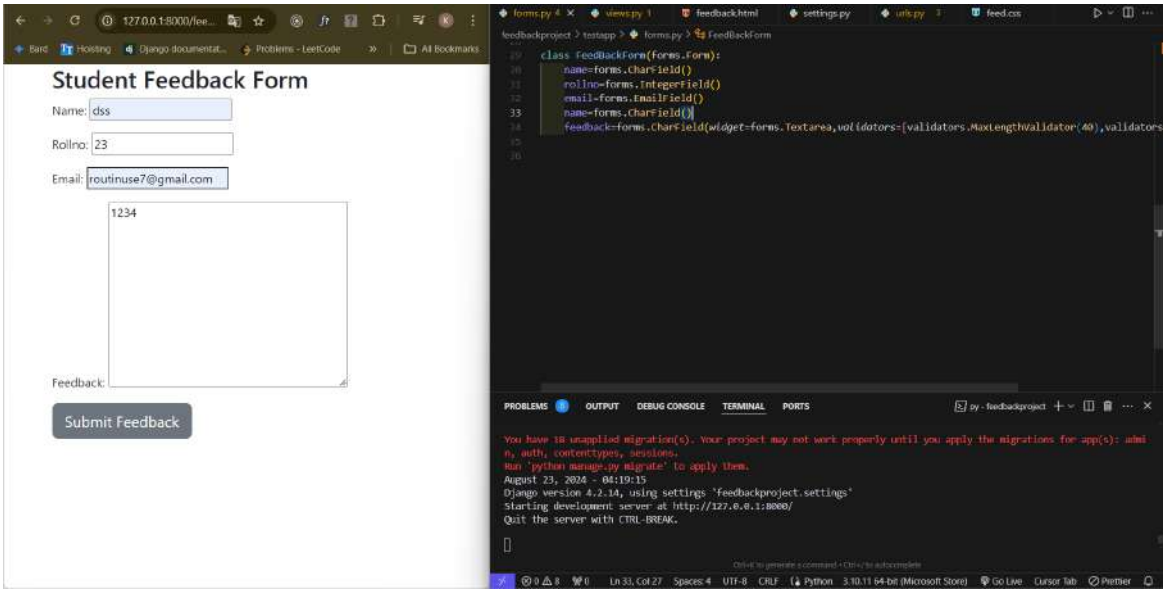



```
You have 18 unapplied migration(s). Your project may not wor
Run 'python manage.py migrate' to apply them.
August 22, 2024 - 05:57:52
Django version 4.2.14, using settings 'feedbackproject.setti
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

```
Validating name field
Validating roll no field
Validating email field
Validating feedback form
Form Validation Successfull Printing feedback Information
##############################
Name: khushal
RollNo: 12
MailId: khushalsavalakha@gmail.com
Feedback: Hello now it's morning 5:57 A.M.
[22/Aug/2024 05:57:53] "POST /feedback/ HTTP/1.1" 200 690
[22/Aug/2024 05:57:53] "GET /static/css/feed.css HTTP/1.1" 4
```

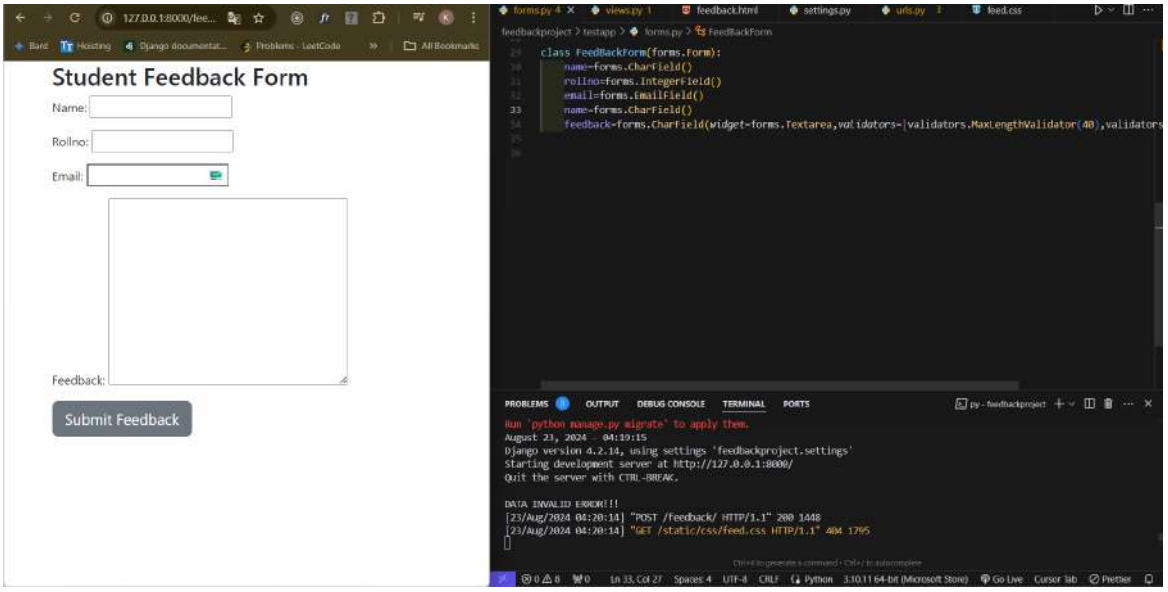
```

Note: Django will call these field level clean methods automatically and we are
    not required to call these methods explicitly.

```

```
2. By using django inbuilt validators
-------------------------------------
Django provides several inbuilt validators to perform  very common validations.
We can use directly and we are not responsible to implement those.
All inbuilt validators present in django.core module


from django import forms
from django.core import validators

class FeedBackForm(forms.Form):
    name = forms.CharField()
    rollno = forms.IntegerField()
    email = forms.EmailField()
    feedback = forms.CharField(widget=forms.Textarea,
            validators=[validators.MaxLengthValidator(40)])
```



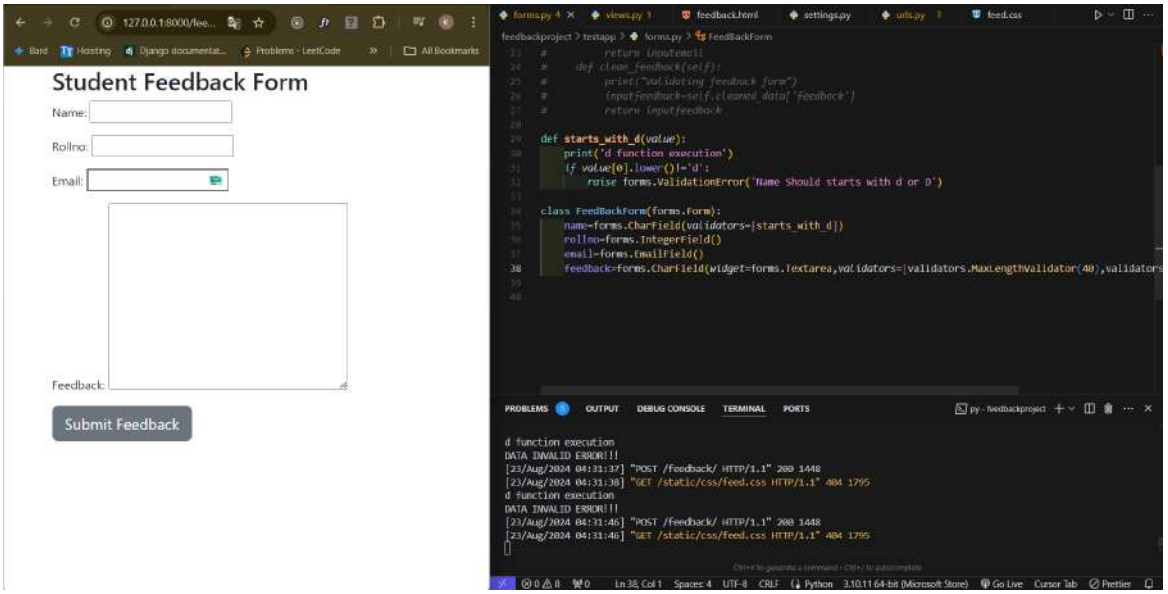
Khushal Savalakha

```
class FeedBackForm(forms.Form):
    name=forms.CharField()
    rollno=forms.IntegerField()
    email=forms.EmailField()
    name=forms.CharField()
    feedback=forms.CharField(widget=forms.Textarea
    ,validators=[validators.MaxLengthValidator(40),
    validators.MinLengthValidator(10)])
```



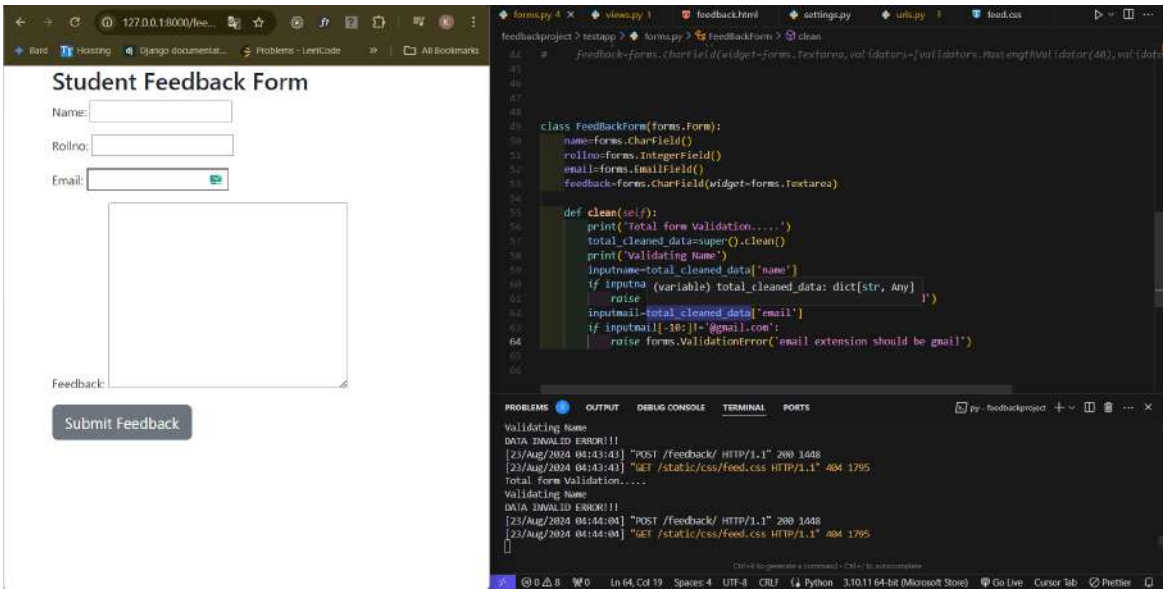

```
def starts_with_d(value):
    print('d function execution')
    if value[0].lower()!='d':
        raise forms.ValidationError('Name Should starts with

def gmail_check(value):
    if value[-10:]!='@gmail.com':
        raise forms.ValidationError('Mail extension should b

class FeedBackForm(forms.Form):
    name=forms.CharField(validators=[starts_with_d])
    rollno=forms.IntegerField()
    email=forms.EmailField()
```

```
    feedback=forms.CharField(widget=forms.Textarea,validator
```

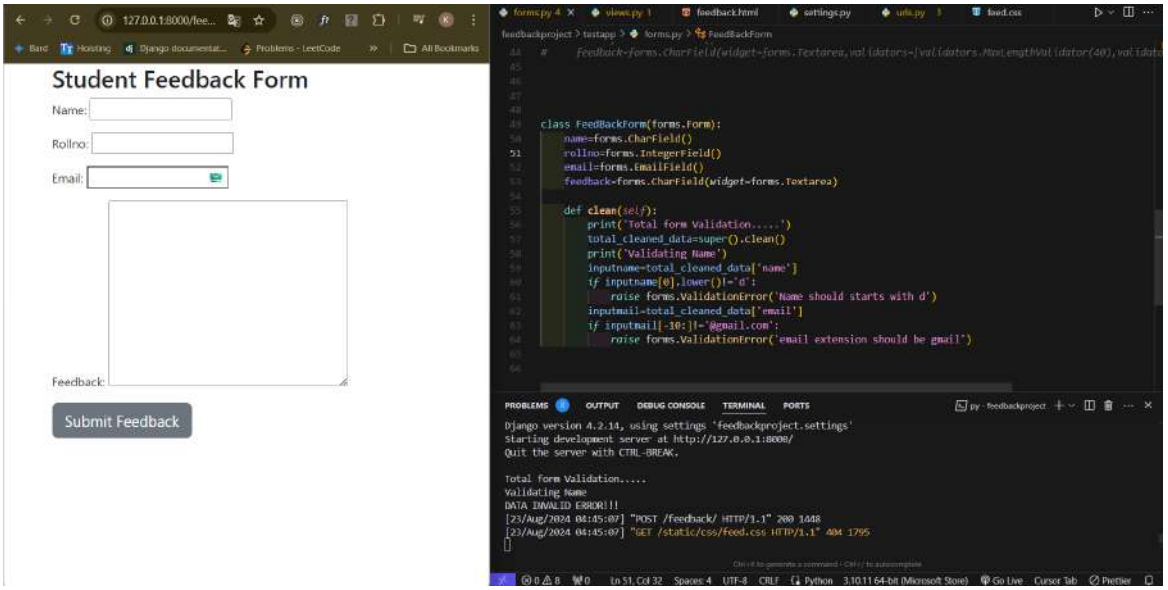


here form does not accept data which name is not started with ‘d’ or ‘D’.

if i send khushal as a name and gmail :khushal@yahoo.com if i send.



gmail :khushal@yahoo.com if i send.




```
def feedback_view(request):
    submitted=False
    name=''
    if request.method=='POST':
        form=FeedBackForm(request.POST)
        if form.is_valid():
            print('Form Validation Successfull Printing feed
            print('#'*30)
            print('Name:',form.cleaned_data['name'])
            print('RollNo:',form.cleaned_data['rollno'])
            print('MailId:',form.cleaned_data['email'])
            print('Feedback:',form.cleaned_data['feedback'])
            submitted=True
            name=form.cleaned_data['name']
        else:
            print('DATA INVALID ERROR!!!')
    form=FeedBackForm()
    return render(request,'testapp/feedback.html',{'form':fo
```

means all function which are present in class are executed as custom validator and we can give any name to function there is no any syntax am i true ?

Yes, you're on the right track, but let's clarify a few key points:

1. Custom Validation in Django Forms:

- In Django, you can create custom validation methods inside your form class to enforce specific validation rules that go beyond Django's built-in field validators.

2. Custom `clean_<fieldname>` Methods:

- If you want to validate a specific field individually, you can create a method in the form class named `clean_<fieldname>`. For example, if you want to validate the `name` field, you can create a method named `clean_name`:

```
class FeedBackForm(forms.Form):
    name = forms.CharField()

    def clean_name(self):
        name = self.cleaned_data['name']
        if name[0].lower() != 'd':
            raise forms.ValidationError('Name should s
tart with d')
        return name
```

- This method is automatically called by Django when validating the `name` field.

3. `clean()` Method for Form-Wide Validation:

- The `clean()` method in the form class is used for validating multiple fields together or for performing additional validation that involves more than one field. This method doesn't have to follow any specific naming convention—just `clean()`.

```
class FeedBackForm(forms.Form):
    name = forms.CharField()
    email = forms.EmailField()

    def clean(self):
        cleaned_data = super().clean()
        name = cleaned_data.get('name')
        email = cleaned_data.get('email')

        if name and name[0].lower() != 'd':
            raise forms.ValidationError('Name should s
tart with d')
```


Khushal Savalakha

```
        if email and not email.endswith('@gmail.com'):
            raise forms.ValidationError('Email should
be a Gmail address')
```

4. No Specific Syntax for Naming Custom Validators:

- You can name your custom validation functions anything you like as long as they aren't special methods (like `clean_<fieldname>` or `clean`) that Django expects.
- If you create a function outside of the form class to use as a validator, you can name it anything you want. For example:

```
def starts_with_d(value):
    if value[0].lower() != 'd':
        raise forms.ValidationError('Name should start
with d')

class FeedBackForm(forms.Form):
    name = forms.CharField(validators=[starts_with_d])
```

Summary:

- **Custom validation functions** in Django forms can be named anything, and you can define them either inside or outside the form class.
- **Specific naming conventions** like `clean_<fieldname>` and `clean()` are used when you want Django to automatically call these methods during the validation process.
- **General functions** (like `starts_with_d`) can be created anywhere and then attached to fields using the `validators` parameter.

This flexibility allows you to enforce custom validation rules in a way that best suits your form's requirements.

> 💡 it is necessory to do makemigrations and migrate in when we create class in forms.py ?

No, it is **not necessary** to run `makemigrations` and `migrate` when you create or modify a class in `forms.py`.

Khushal Savalakha

Explanation:

- `forms.py` is used for defining forms in Django, which handle the presentation and validation of data in your web application. The classes you define in `forms.py` (such as `FeedBackForm`) are purely for handling user input and validating it. These classes are not directly related to the database schema.
- `makemigrations` **and** `migrate` are commands that are used to manage your database schema. Specifically:
  - `makemigrations`: This command creates new migration files based on the changes you've made to your models (in `models.py`). These migrations describe the changes to your database schema (like creating new tables or fields).
  - `migrate`: This command applies the migrations to your database, making the changes described in the migration files.

When to Use `makemigrations` and `migrate`:

- **Use** `makemigrations` **and** `migrate` when you create or modify models in `models.py`. This is because models directly represent database tables, and any changes to them may require updates to the database schema.

When You Don't Need `makemigrations` and `migrate`:

- **Do not use** `makemigrations` **and** `migrate` when you are only working with forms in `forms.py`. Since forms don't affect the database schema, there's no need to generate migrations for them.

Example Scenario:

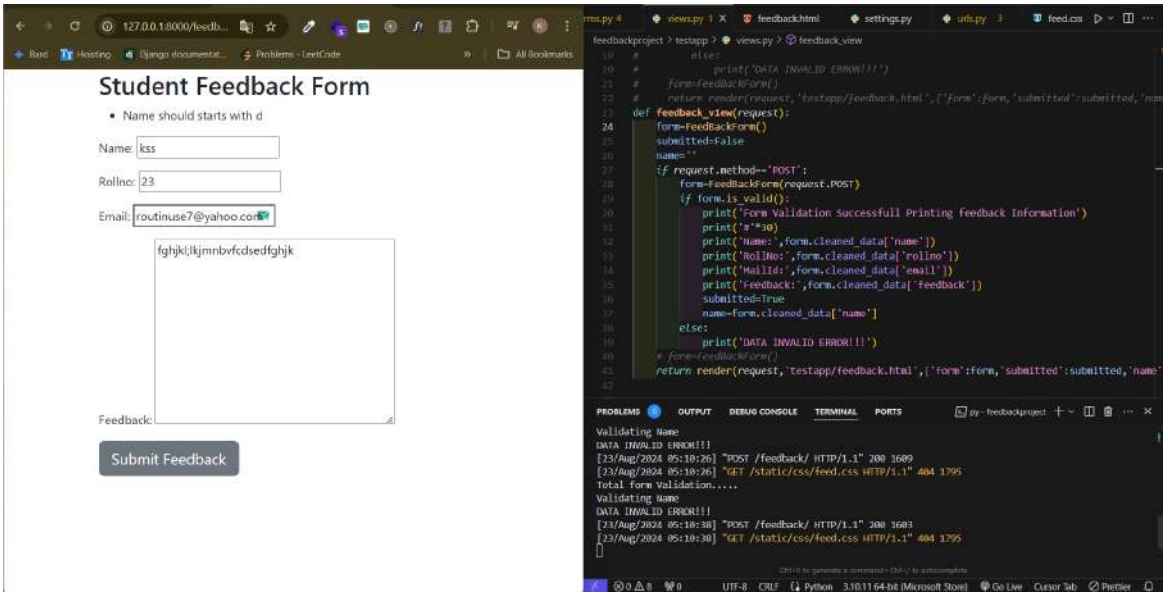
- **Creating or Modifying a Form:**
  - You define a new form class in `forms.py` or modify an existing one.
  - No database changes are involved, so you don't need to run `makemigrations` or `migrate`.
- **Creating or Modifying a Model:**
  - You define a new model class in `models.py` or modify an existing one (e.g., adding a new field).
  - You need to run `makemigrations` to generate the migration files, and then `migrate` to apply those changes to the database.

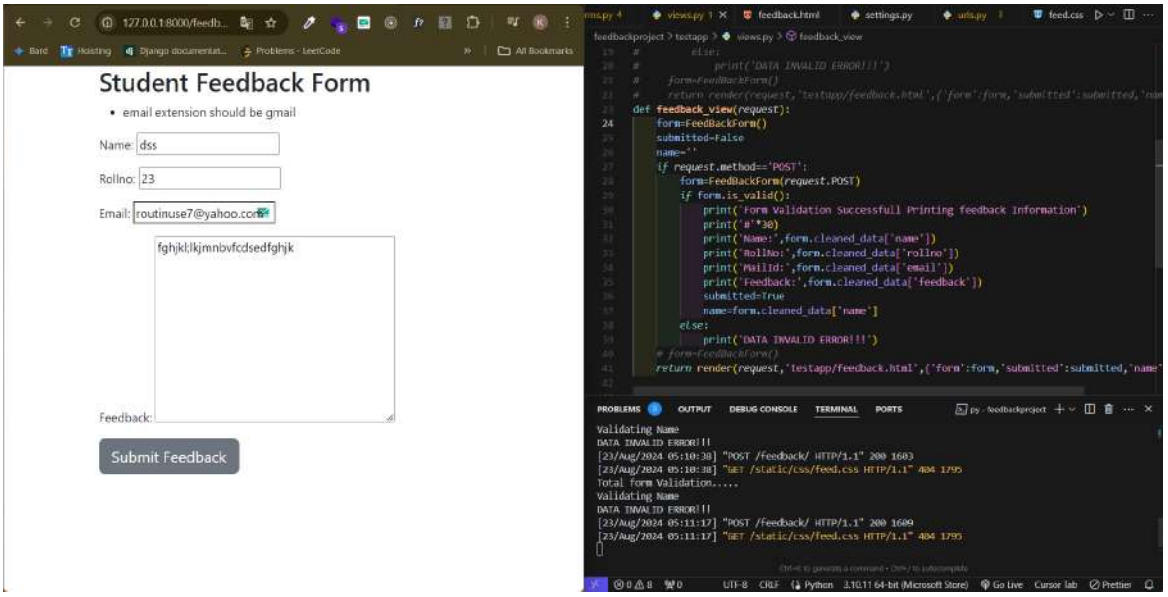
Summary:

Khushal Savalakha

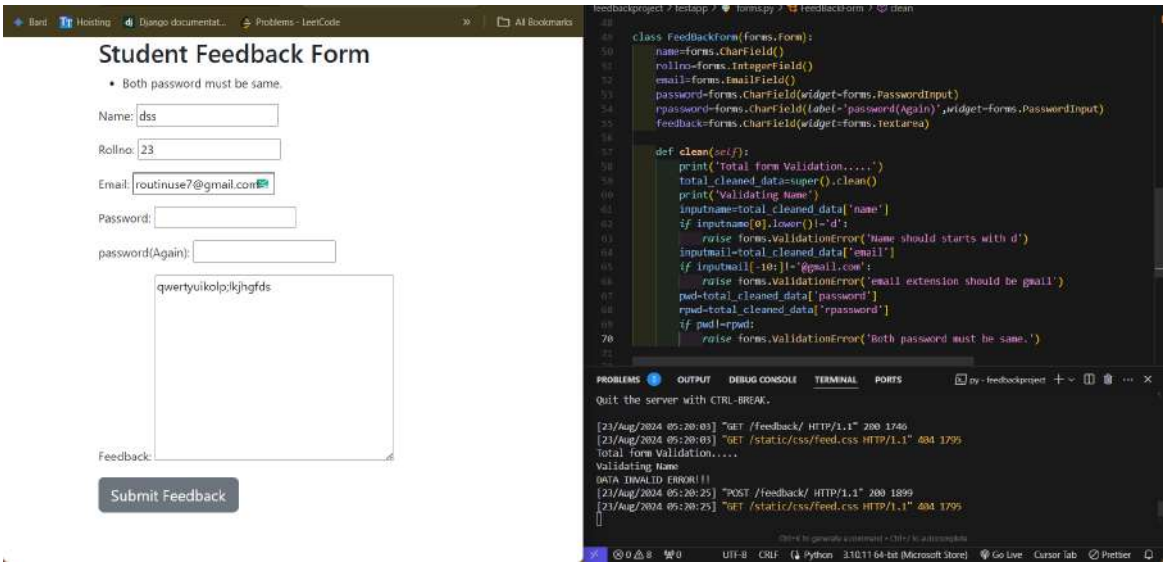
- `forms.py`: No need for `makemigrations` or `migrate`.
- `models.py`: Use `makemigrations` and `migrate` when you make changes.

```
def feedback_view(request):
    form=FeedBackForm()
    submitted=False
    name=''
    if request.method=='POST':
        form=FeedBackForm(request.POST)
        if form.is_valid():
            print('Form Validation Successfull Printing feed
            print('#'*30)
            print('Name:',form.cleaned_data['name'])
            print('RollNo:',form.cleaned_data['rollno'])
            print('MailId:',form.cleaned_data['email'])
            print('Feedback:',form.cleaned_data['feedback'])
            submitted=True
            name=form.cleaned_data['name']
        else:
            print('DATA INVALID ERROR!!!')
    # form=FeedBackForm()
    return render(request,'testapp/feedback.html',{'form':fo
```






```
class FeedBackForm(forms.Form):
    name=forms.CharField()
    rollno=forms.IntegerField()
    email=forms.EmailField()
    password=forms.CharField(widget=forms.PasswordInput)
    rpassword=forms.CharField(label='password(Again)',widget
    feedback=forms.CharField(widget=forms.Textarea)

    def clean(self):
        print('Total form Validation.....')
        total_cleaned_data=super().clean()
        print('Validating Name')
        inputname=total_cleaned_data['name']
        if inputname[0].lower()!='d':
            raise forms.ValidationError('Name should starts
        inputmail=total_cleaned_data['email']
        if inputmail[-10:]!='@gmail.com':
            raise forms.ValidationError('email extension sho
        pwd=total_cleaned_data['password']
        rpwd=total_cleaned_data['rpassword']
        if pwd!=rpwd:
            raise forms.ValidationError('Both password must
```



Explain what is the meaning of BOT request and also explains how it creates heavy traffic to the website ?with examples and also tell me which type of BOT we are talking about here and how they present on any site ?

What is a BOT Request?

A **BOT request** refers to an automated request made to a website or web service by a software application known as a "bot" (short for "robot"). These bots are programmed to perform repetitive tasks, such as scraping content, submitting forms, or clicking on links, without human intervention.

Types of Bots

There are several types of bots, but for the context of your question, we'll focus on the following:

1. **Good Bots**:
    - **Search Engine Crawlers**: Bots like Google's crawler (Googlebot) index websites to make them searchable.
    - **Monitoring Bots**: Bots used by services to monitor the uptime or performance of a website.
    - **Chatbots**: Bots that interact with users on a website to provide customer service.
2. **Malicious Bots**:
    - **Scraping Bots**: Bots that harvest content, prices, or other data from websites without permission.

- **Spam Bots**: Bots that submit forms, post comments, or send messages with spammy content.
- **DDoS Bots**: Bots that are part of a distributed denial-of-service (DDoS) attack, where they flood a website with traffic to overwhelm it.

How BOT Requests Create Heavy Traffic

1. **High Frequency of Requests**:
   - Bots can generate requests at a very high frequency, far beyond what a typical human user would do. For example, a scraping bot might request thousands of pages in a short amount of time to harvest all the data from a website.
2. **Parallel Requests**:
   - Bots can be programmed to make parallel requests, which means they can hit multiple pages or services on a website simultaneously, increasing the load on the server.
3. **Distributed Traffic (DDoS)**:
   - In a Distributed Denial of Service (DDoS) attack, a network of bots (often referred to as a botnet) sends a massive amount of traffic to a website. This can overwhelm the server's resources, making the website slow or even completely inaccessible to legitimate users.
4. **Continuous and Unpredictable Traffic**:
   - Unlike human users who visit a website during certain hours or days, bots can generate traffic continuously, including during off-peak hours, leading to consistent strain on the server.

Example Scenarios

1. **Search Engine Crawlers**:
   - **Scenario**: A search engine bot like Googlebot visits a website to index its pages.
   - **Impact**: Generally, this is a good thing, as it helps your site appear in search results. However, if the bot is not well-behaved or configured correctly, it can request pages too frequently, leading to excessive load on the server.
2. **Scraping Bots**:
   - **Scenario**: A competitor might deploy a bot to scrape your website’s prices or content.

Khushal Savalakha

- **Impact**: If the bot is not rate-limited or blocked, it can create heavy traffic, consuming bandwidth and server resources, and potentially making the website slow for legitimate users.

3. **DDoS Attack**:
   - **Scenario**: A botnet is used to launch a DDoS attack against a website.
   - **Impact**: The botnet floods the website with so much traffic that the server becomes overwhelmed, leading to outages or degraded performance, which affects all users.

How Bots Present on a Website

1. **User-Agent String**:
   - Bots often identify themselves through the `User-Agent` string in their HTTP requests. For example, Google's bot uses a `User-Agent` string like "Googlebot."
2. **IP Address Patterns**:
   - Some bots may come from known IP address ranges, which can be used to identify and filter them.
3. **Behavior Patterns**:
   - Bots often follow predictable patterns, such as visiting many pages very quickly, making requests at a regular interval, or accessing pages that a typical user wouldn't.
4. **Headers and Cookies**:
   - Bots may or may not use cookies, and they may include certain headers in their requests that differentiate them from typical browsers.

Preventing and Mitigating Bot Traffic

1. **Rate Limiting**:
   - Limit the number of requests that a single IP address or user-agent can make within a certain timeframe.
2. **CAPTCHA**:
   - Implement CAPTCHA challenges on forms or login pages to ensure the request is coming from a human.
3. **Bot Detection Services**:

Khushal Savalakha

- Use services like Cloudflare, Akamai, or others that offer bot detection and mitigation features.

4. **IP Blocking**:
    - Block known malicious IP addresses or IP ranges that are associated with bot traffic.

Summary

BOT requests are automated requests made by software applications, which can significantly increase traffic to a website. While some bots, like search engine crawlers, are beneficial, others, like scraping bots or DDoS bots, can create heavy and potentially harmful traffic. Managing bot traffic involves identifying bot behavior and implementing measures to limit their impact on the website's performance.

```
How to prevent requests from BOT?
---------------------------------
Automated program script is nothing but: BOT

problems with BOT requests:

1. To create unnecessary heavy traffic to the websites
2. To spread malware(viruses)

```

```
How to implement:
----------------
In the form, we will place one hidden field, which is not visible to the end user.
Hence there is no chance of providing value to this hidden field.

But BOT will send value for that hidden field also. If hidden field got some value
means it is the request from the bot and prevent that from submission.
```

```
class FeedBackForm(forms.Form):
    name=forms.CharField()
    rollno=forms.IntegerField()
    email=forms.EmailField()
    password=forms.CharField(widget=forms.PasswordInput)
    rpassword=forms.CharField(label='password(Again)',widget
    feedback=forms.CharField(widget=forms.Textarea)
    bot_handler=forms.CharField(required=False,widget=forms.

    def clean(self):
        print('Total form Validation.....')
```


Khushal Savalakha

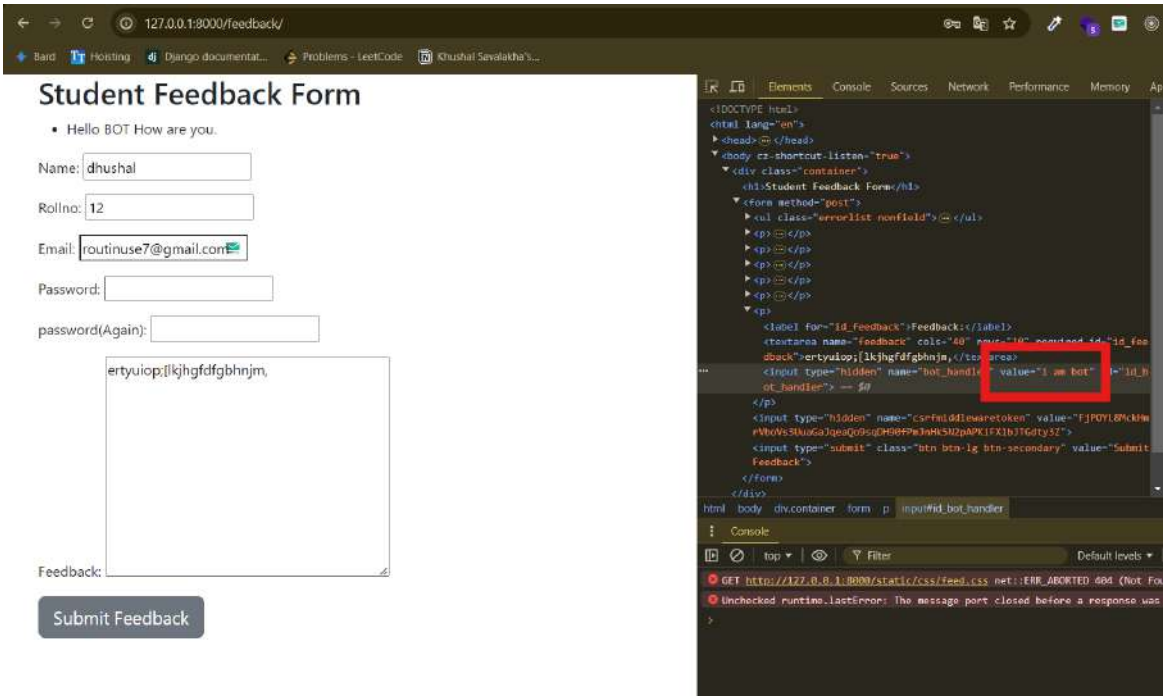
```
        total_cleaned_data=super().clean()
        print('Validating Name')
        inputname=total_cleaned_data['name']
        if inputname[0].lower()!='d':
            raise forms.ValidationError('Name should starts
        inputmail=total_cleaned_data['email']
        if inputmail[-10:]!='@gmail.com':
            raise forms.ValidationError('email extension sho
        pwd=total_cleaned_data['password']
        rpwd=total_cleaned_data['rpassword']
        if pwd!=rpwd:
            raise forms.ValidationError('Both password must
        bot_handler_value=total_cleaned_data['bot_handler']
        if len(bot_handler_value)>0:
            raise forms.ValidationError('Hello BOT How are y
```

```
<!DOCTYPE html>

<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initi
    <title>Document</title>
    <link
    href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/
    rel="stylesheet"
    integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5p
    crossorigin="anonymous"
  />

  <link rel="stylesheet" href='/static/css/feed.css''>
</head>
<body>

    <div class="container">

        <h1>Student Feedback Form</h1>
```


Khushal Savalakha

```
        <form method="post">
            <p>
    <label for="id_name">Name:</label>
    <input type="text" name="name" required id="id_name">
  </p>

  <p>
    <label for="id_rollno">Rollno:</label>
    <input type="number" name="rollno" required id="id_rollr
  </p>

  <p>
    <label for="id_email">Email:</label>
    <input type="email" name="email" maxlength="320" require
  </p>

  <p>
    <label for="id_password">Password:</label>
    <input type="password" name="password" required id="id_p
  </p>
  <p>
    <label for="id_rpassword">password(Again):</label>
    <input type="password" name="rpassword" required id="id_
  </p>

  <p>
    <label for="id_feedback">Feedback:</label>
    <textarea name="feedback" cols="40" rows="10" required i
</textarea>
      <input type="hidden" name="bot_handler" id="id_bot_har
  </p>
            <input type="hidden" name="csrfmiddlewaretoken"
            <input type="submit" class="btn btn-lg btn-secor
        </form>
    </div>
</body>
</html>
```


Khushal Savalakha



means BOT fill data by web scraping am i true ?

Yes, you are correct. **Bots** can indeed fill data by **web scraping**.

What is Web Scraping?

Web scraping is the process of automatically extracting data from websites. This involves making requests to web pages, retrieving the HTML content, and then parsing that content to extract specific information.

How Bots Use Web Scraping to Fill Data

1. **Automated Data Extraction**:
    - Bots can be programmed to visit web pages, extract specific pieces of data (like prices, product details, or contact information), and store this data in a structured format, such as a database or spreadsheet.
2. **Form Filling**:
    - Some bots are designed to automatically fill out forms on websites, submitting the required data to simulate human interaction. This can be used for purposes like automated registration, surveys, or spamming.
3. **Content Harvesting**:

- Scraping bots can gather large amounts of content from websites, such as articles, blog posts, images, or any other publicly available data. This data can then be repurposed or used for various applications, sometimes even without the website owner's permission.

Example Scenario

- **Price Aggregator Bot**:
  - A bot could be designed to scrape e-commerce websites to collect pricing information on specific products. It might visit multiple product pages, extract the price, and store that data in a database. This scraped data could then be used to create a price comparison website.
- **Lead Generation Bot**:
  - A bot could scrape websites for contact information such as email addresses or phone numbers, which could then be used for marketing or sales purposes.

Summary

Bots that perform web scraping automate the process of data extraction from websites. They can collect various types of data by parsing the HTML content of web pages and filling in forms to submit or retrieve information. While web scraping can be used for legitimate purposes, it can also create heavy traffic on websites and potentially violate terms of service or data privacy regulations if done improperly.

> 💡 it means when we use simple form then on that time we creating form.py file in which we are importing forms and then creating class with any name and in which parameter is forms.Form then we are defining fields which we want from user and also define their size(which is optional) then validation function or clean function etc then in view.py import this form class and making their instance and we are checking on this form which request is coming and then validation and other process happens am i true ?

Yes, your understanding is correct! Here's a breakdown of the process when you're using a simple form in Django:

1. Creating a Form (in `forms.py`):

- **Importing Forms:**

Khushal Savalakha

```
from django import forms
```

- **Creating a Form Class:**
  - You create a class that inherits from `forms.Form`.
  - Inside this class, you define the fields you want to capture from the user, such as `CharField`, `EmailField`, `IntegerField`, etc.
  - You can also specify additional attributes like `max_length`, `required`, `widget`, etc., to customize the fields.

```
class MyForm(forms.Form):
    name = forms.CharField(max_length=100)
    email = forms.EmailField()
    age = forms.IntegerField(required=False)
```

- **Validation (`clean` method):**
  - You can add custom validation by overriding the `clean` method or by defining `clean_<fieldname>` methods for field-specific validation.
  - The `clean` method is called during form validation to ensure that the data is clean before processing.

```
class MyForm(forms.Form):
    name = forms.CharField(max_length=100)
    email = forms.EmailField()
    age = forms.IntegerField(required=False)

    def clean_age(self):
        age = self.cleaned_data.get('age')
        if age is not None and age < 0:
            raise forms.ValidationError('Age cannot be
negative.')
        return age
```

2. Using the Form in a View (in `views.py`):

- **Importing the Form:**
  - In your `views.py`, you import the form class you created in `forms.py`.

Khushal Savalakha

```
from .forms import MyForm
```

- **Creating an Instance of the Form:**
  - In your view, you create an instance of the form. This can be an empty form (for GET requests) or a form populated with data (for POST requests).

```
def my_view(request):
    if request.method == 'POST':
        form = MyForm(request.POST)
        if form.is_valid():
            # Process the data, e.g., save it to the d
atabase
            name = form.cleaned_data['name']
            email = form.cleaned_data['email']
            age = form.cleaned_data['age']
            # ... do something with the data ...
    else:
        form = MyForm()

    return render(request, 'my_template.html', {'for
m': form})
```

- **Handling the Request:**
  - **GET Request:** When a GET request is made, you typically render an empty form, which is displayed to the user.
  - **POST Request:** When a POST request is made (e.g., when the form is submitted), the form is populated with the data from `request.POST`.
  - **Validation:** The `is_valid()` method checks if the data submitted is valid. If it is, you can access the cleaned data using `form.cleaned_data` and process it (e.g., save it to a database).

3. Rendering the Form in a Template:

- In your template, you render the form using the Django template language. Typically, you use something like `{{ form.as_p }}`, `{{ form.as_table }}`, or loop over the fields manually.

Khushal Savalakha

```
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Submit</button>
</form>
```

Summary:

- `forms.py`: Define the form, including the fields and any validation logic.
- `views.py`: Handle the form submission, validate it, and process the data.
- **Template:** Display the form and handle user interaction.

This workflow is commonly used for simple forms in Django, where you need to capture data from the user and process it in some way. Your understanding of the process is accurate!

▼ For Captcha

Step-1: pip install django-simple-captcha

```
Collecting Pillow>=6.2.0
  Downloading Pillow-8.3.1-cp38-cp38-win32.whl (2.8 MB)
     |[redacted]| 2.8 MB 2.2 MB/s
Collecting django-ranged-response==0.2.0
  Downloading django-ranged-response-0.2.0.tar.gz (3.0 kB)
Requirement already satisfied: sqlparse>=0.2.2 in c:\users\gouri\appdata\local\programs\python\python38-32\l
cha) (0.4.1)
Requirement already satisfied: pytz in c:\users\gouri\appdata\local\programs\python\python38-32\lib\site-pac
1)
Requirement already satisfied: asgiref<4,>=3.3.2 in c:\users\gouri\appdata\local\programs\python\python38-32
ptcha) (3.4.0)
Using legacy 'setup.py install' for django-simple-captcha, since package 'wheel' is not installed.
Using legacy 'setup.py install' for django-ranged-response, since package 'wheel' is not installed.
Installing collected packages: six, Pillow, django-ranged-response, django-simple-captcha
```

I could not capture full screenshot, but we don't need to do anything just type command (pip install django-simple-captcha) and hit enter. Installation will complete automatically.

Khushal Savalakha

```
settings.py
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'feedbackApp',
    'captcha'
]
```

Step-3: Run "python manage.py migrate" command


```
settings.py
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'feedbackApp',
    'captcha'
]
```

Step-3: Run "python manage.py migrate" command

```
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('feed/',views.feed_view),
    path('feedbackApp/',include('feedbackApp.urls')),
    path(r'captcha/', include('captcha.urls')),
]
```

Step-5: Update forms.py file.

Add following lines of code, see line no. 3 at below screenshot.

from captcha.fields import CaptchaField

Add following lines of code, see line no. 10 at below screenshot.

captcha = CaptchaField()

forms.py

```
1  from django import forms
2  from django.core import validators
3  from captcha.fields import CaptchaField
4
5  class Studfeed (forms.Form):
6      name = forms.CharField()
7      rollno = forms.IntegerField()
8      email = forms.EmailField()
9      feedback = forms.CharField(widget = forms.Textarea)
10     captcha = CaptchaField()
11
```

Everything is done for captcha, below part is for verifying it.

Everything is done for captcha, below part is for verifying it.

Save all and runserver using "py manage.py runserver" command.

Send request using your url pattern.

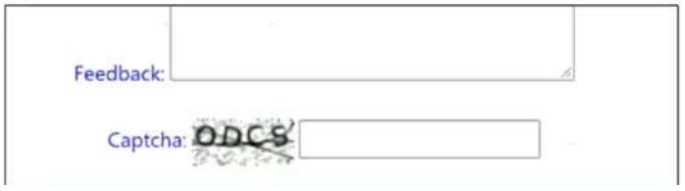
Here my application level url pattern as follows.

http://127.0.0.1:8000/feedbackApp/feed/

output for above url patten as follows.

We can see 'captcha' below feedback option in above screenshot.

I zoomed out for clear picture.



Khushal Savalakha

Student Feedback Form

Thank you Rakesh for submitting your feedback
All the best for your future

And output at server console as follows.

```
[11/Aug/2021 16:24:05] "GET /feedbackApp/feed/ HTTP/1.1" 200 1657
[11/Aug/2021 16:24:05] "GET /captcha/image/d9245291f0208a354d19fa18b5328e626f085d68/ HTTP/1.1" 200 4338

        Form Validation completed successfully

        Name: =  Rakesh

        Rollno: =  20

        Email: =  abcde@gmail.com

        Feedback: =  Trying  Captcha Code
[11/Aug/2021 16:26:31] "POST /feedbackApp/feed/ HTTP/1.1" 200 640
```

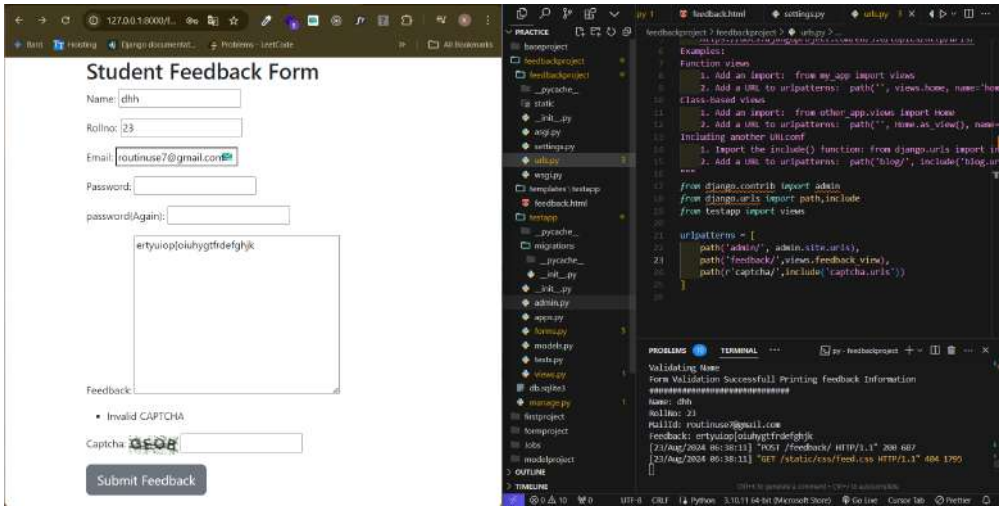
Now again I refresh the screen for new feedback form.

```
from django import forms
from django.core import validators
from captcha.fields import CaptchaField

class FeedBackForm(forms.Form):
    name=forms.CharField()
    rollno=forms.IntegerField()
    email=forms.EmailField()
    password=forms.CharField(widget=forms.PasswordInput)
    rpassword=forms.CharField(label='password(Again)',wid
    feedback=forms.CharField(widget=forms.Textarea)
    bot_handler=forms.CharField(required=False,widget=for
    captcha=CaptchaField()

    def clean(self):
        print('Total form Validation.....')
        total_cleaned_data=super().clean()
        print('Validating Name')
        inputname=total_cleaned_data['name']
        if inputname[0].lower()!='d':
            raise forms.ValidationError('Name should star
        inputmail=total_cleaned_data['email']
        if inputmail[-10:]!='@gmail.com':
```


Khushal Savalakha

```
            raise forms.ValidationError('email extension
        pwd=total_cleaned_data['password']
        rpwd=total_cleaned_data['rpassword']
        if pwd!=rpwd:
            raise forms.ValidationError('Both password mu
        bot_handler_value=total_cleaned_data['bot_handler
        if len(bot_handler_value)>0:
            raise forms.ValidationError('Hello BOT How ar
```

```
from django.contrib import admin
from django.urls import path,include
from testapp import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('feedback/',views.feedback_view),
    path(r'captcha/',include('captcha.urls'))
]
```



Khushal Savalakha

▼ Working With Django Model Forms:

```
How to develop model based forms:
--------------------------------
1. While developing form class, we have to inherit from forms.ModelForm class instead of forms.Form class.

   class RegisterForm(forms.Form) -->Normal form
   class RegisterForm(forms.ModelForm) -->Model form
```

```
2.

class RegisterForm(forms.ModelForm):
      class Meta:
          model = Student
          fields = '__all__'
```

```
case-1: Instead of all fields, if we require particular fields
class RegisterForm(forms.ModelForm):
      class Meta:
          model = Student
          fields = ('field1','field2','field3')

case-2: Instead of all fields, if we want to exclude certain fields
class RegisterForm(forms.ModelForm):
      class Meta:
          model = Student
          exclude = ['field1','field2']
```

```
How to save user's input data to database in model based forms:
---------------------------------------------------------------
def student_view(request):
    if request.method == 'POST':
              form = StudentForm(request.POST)
              if form.is_valid():
                  form.save()
```

```
Note:
class RegisterForm(forms.ModelForm):
     field declarations for validation
     class Meta:
         model=Student
         fields = '__all__'
```

PS D:\Django\Practice\movieproject> tree
Folder PATH listing for volume New Volume
Volume serial number is F491-4E3E
D:.
├───movieproject
│   └───__pycache__
├───static
│   └───css
├───templates
│   └───testapp
└───testapp
    ├───migrations
    │   └───__pycache__
    └───__pycache__
write explanation how here model is working with adding and showing movie with forms explain flow step by step with example.

To explain how the model is working with adding and showing movies using forms in your Django project, I'll provide a step-by-step flow using the files you've shared. I'll assume you have a basic understanding of Django, such as models, views, templates, and forms.

1. Project Structure Overview

Here's the directory structure you provided:

```
D:.
├───movieproject
│   └───__pycache__
├───static
│   └───css
├───templates
│   └───testapp
└───testapp
    ├───migrations
    │   └───__pycache__
    └───__pycache__
```

- **movieproject**: This is the main project directory.

Khushal Savalakha

- **static/css**: Contains static CSS files.
- **templates/testapp**: Contains HTML templates for the app `testapp`.
- **testapp**: This is your Django app, which includes the `models.py`, `views.py`, `forms.py`, and other files related to the app.

2. Flow of Adding and Showing Movies

Step 1: Defining the Movie Model

The `Movie` model is defined in `models.py` (not provided but inferred). This model will have fields like `title`, `description`, `release_date`, etc. This model represents a movie entity in your application.

```
# Example model definition (inferred)
from django.db import models

class Movie(models.Model):
    title = models.CharField(max_length=100)
    description = models.TextField()
    release_date = models.DateField()
```

Step 2: Creating the Movie Form

In the `forms.py` file, you create a form class `MovieForm` that maps to the `Movie` model. This form will be used to add new movies through the frontend.

```
# forms.py
from django import forms
from .models import Movie

class MovieForm(forms.ModelForm):
    class Meta:
        model = Movie  # Links the form to the Movie mode
l
        fields = ['title', 'description', 'release_date']
# Fields to include in the form
```

Step 3: Handling Form Submission in Views

Khushal Savalakha

In the `views.py` file, you handle the logic for displaying the form and processing the form submission.

```
# views.py
from django.shortcuts import render, redirect
from .forms import MovieForm
from .models import Movie

def add_movie(request):
    if request.method == "POST":
        form = MovieForm(request.POST)
        if form.is_valid():
            form.save()  # Save the form data to the data
base
            return redirect('movie_list')  # Redirect to
the list of movies after saving
    else:
        form = MovieForm()  # Display an empty form

    return render(request, 'testapp/add_movie.html', {'fo
rm': form})
```

- **GET request**: When the user navigates to the form page, a blank `MovieForm` is rendered.
- **POST request**: When the user submits the form, the data is validated and saved to the `Movie` model. The user is then redirected to the movie list page.

Step 4: Displaying the Form in a Template

In the `templates/testapp/add_movie.html` file, the form is rendered using Django template language.

```
<!-- templates/testapp/add_movie.html -->
<form method="POST">
    {% csrf_token %}
    {{ form.as_p }}  <!-- Renders the form fields -->
    <button type="submit">Add Movie</button>
</form>
```


Khushal Savalakha

- **CSRF Token**: Protects against cross-site request forgery attacks.
- `form.as_p`: Renders the form fields as HTML paragraphs.

Step 5: Displaying the List of Movies

To display the list of movies, another view and template would be created.

```
# views.py
def movie_list(request):
    movies = Movie.objects.all()  # Retrieves all movies
from the database
    return render(request, 'testapp/movie_list.html', {'m
ovies': movies})
```

```
<!-- templates/testapp/movie_list.html -->
<h1>Movie List</h1>
<ul>
    {% for movie in movies %}
    <li>{{ movie.title }} - {{ movie.release_date }}</li>
    {% endfor %}
</ul>
```

Summary

- **Model**: The `Movie` model represents the data structure.
- **Form**: The `MovieForm` handles user input and validation.
- **Views**: The `add_movie` view processes form submissions, while the `movie_list` view displays stored movies.
- **Templates**: `add_movie.html` and `movie_list.html` handle the front-end display of the form and movie list.

This flow allows users to add new movies and view the list of movies using Django's powerful form handling and template rendering system.

Khushal Savalakha

💡 it means when we create models then we do makemigrations and migrate to make database then we are making writing code in admin.py which data we have to see then we are creating forms in which we are importing model and make class with meta class in which we are fetching all columns data using fields='**all**' then this data we can using making instance of main class in form then adding and retrival process happens am i true ?

Yes, you're on the right track! Let's break down the process step by step to clarify how Django handles adding and retrieving data:

1. Creating Models

- **Define Models**: You first create models in `models.py` to define the structure of your data (e.g., `Movie` model with fields like `title`, `description`, `release_date`).
- **Migrations**: After defining your models, you run `makemigrations` to generate migration files and `migrate` to apply these migrations to the database, creating the necessary tables.

2. Admin Configuration

- **Register Models in `admin.py`**: You register your models in `admin.py` so that you can view and manage them via Django's admin interface. This is useful for quick data entry and management.

3. Creating Forms

- **Forms**: In `forms.py`, you create forms that link to your models using the `ModelForm` class. The `Meta` class inside your form specifies the model to link to and the fields to include.
    - **Example**: If you use `fields = '__all__'`, all fields of the model are included in the form.

4. Views and Templates

- **Views**: In `views.py`, you handle the logic for displaying the form, processing submissions, and rendering data.
    - **Adding Data**: You create an instance of your form class, and when the form is submitted (POST request), the data is saved to the database if valid.

Khushal Savalakha

- **Retrieving Data**: You retrieve data from the database using queries (e.g., `Movie.objects.all()`).

5. Templates

- **Rendering Forms and Data**: In your HTML templates, you render the forms and display data using Django's template language. The forms handle user input, and the data is displayed back to the user.

Summary

1. **Define Models** -> `makemigrations` and `migrate` to create tables.
2. **Register Models** in `admin.py` to view/manage them in the Django admin.
3. **Create Forms** linked to models using `ModelForm`.
4. **Handle Data** in `views.py` for adding and retrieving using the forms and models.
5. **Render Data** and forms in templates to interact with users.

This process allows you to handle data efficiently in Django, leveraging the framework's ORM and form-handling capabilities.

forms.py

```
from django import forms
from testapp.models import Movie

class MovieForm(forms.ModelForm):
    def rating_validator(value):
        if value<5:
            raise forms.ValidationError()
    rating=forms.IntegerField(validators=[rating_validator])
    class Meta:
        model=Movie
        fields='__all__'
```

models.py

```
from django.db import models

# Create your models here.
```


Khushal Savalakha

```
class Movie(models.Model):
            rdate=models.DateField()
            moviename=models.CharField(max_length=30)
            hero=models.CharField(max_length=30)
            heroine=models.CharField(max_length=30)
            rating=models.IntegerField()
```

views.py

```
from django.shortcuts import render
from testapp.models import Movie
from testapp.forms import MovieForm
# Create your views here.
def index_view(request):
    return render(request,'testapp/index.html')

def list_movies_view(request):
    movies_list=Movie.objects.all()
    return render(request,'testapp/listMovies.html',{'movies

def add_movies_view(request):
    form=MovieForm()
    if request.method=='POST':
        form=MovieForm(request.POST)
        if form.is_valid():
            form.save()
        return index_view(request)
    return render(request,'testapp/addMovies.html',{'form':f
```

add_movies.html

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initi
    <title>Document</title>
```


Khushal Savalakha

```
    <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dis
      rel="stylesheet"
      integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr
      crossorigin="anonymous"
    />
    <link rel="stylesheet" href="{% static 'css/movies.css'
  </head>
  <body>
   <h1>Movies information</h1><!DOCTYPE html>
{% load static %}
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initi
    <title>Document</title>
    <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dis
      rel="stylesheet"
      integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr
      crossorigin="anonymous"
    />
    <link rel="stylesheet" href="{% static 'css/movies.css'
  </head>
  <body>
   <div class="container">
    <h1>Add Movie Information</h1>
    <form method="post">
        {{form.as_p}}
        {% csrf_token %}
        <input type="submit" class="btn btn-primary" name=""
    </form>
   </div>
  </body>
</html>
```

list_movies.html

Khushal Savalakha

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initi
    <title>Document</title>
    <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dis
      rel="stylesheet"
      integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr
      crossorigin="anonymous"
    />
    <link rel="stylesheet" href="{% static 'css/movies.css'
  </head>
  <body>
   <h1>Movies information</h1>
   {% if movies_list %}
   <table border="1">
    <thead>
        <th>Released Date</th>
        <th>Movie Name</th>
        <th>Hero</th>
        <th>Heroine</th>
        <th>Rating</th>
    </thead>
    {% for movie in movies_list %}
    <tr>
        <td>{{movie.rdate}}</td>
        <td>{{movie.moviename}}</td>
        <td>{{movie.hero}}</td>
        <td>{{movie.heroine}}</td>
        <td>{{movie.rating}}</td>
    </tr>
    {% endfor %}
   </table>
   {% else %}
   <h1>No movies Found!!</h1>
```


Khushal Savalakha

```
    {% endif %}
  </body>
</html>
```

index.html

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initi
    <title>Document</title>
    <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dis
      rel="stylesheet"
      integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr
      crossorigin="anonymous"
    />
    <link rel="stylesheet" href="{% static 'css/movies.css'
  </head>
  <body>
    <!-- <h1>Index Page</h1> -->
    <div class="conatainer" align="center">
        <div class="jumbotron">
            <h1>DURGA Movies</h1>
            <p>upto date movies information</p><hr>
            <a class="btn btn-primary btn-lg" href="../addmo
            <a class="btn btn-primary btn-lg" href="../listm
        </div>
    </div>
  </body>
</html>
```

admin.py

```
from django.contrib import admin
from testapp.models import Movie
# Register your models here.
```


Khushal Savalakha

```
class MovieAdmin(admin.ModelAdmin):
    list_display=['rdate','moviename','hero','heroine','rati

admin.site.register(Movie,MovieAdmin)
```

💡 df

- --->models.py
  from django.db import models

Create your models here.

class Product(models.Model):
name=models.CharField(max_length=100)
description=models.TextField()

```
def __str__(self):
    return self.name
```

here what is the need of writing function **str** explain simply and depthly with example.

The `__str__()` function in Django models (and Python in general) is used to define a **human-readable string representation** of an object. When you print an instance of the model, or when you see it in the Django admin or in a query result, the `__str__()` method determines what gets displayed.

Why is `__str__()` important?

1. **Readable Output**: Without `__str__()`, when you try to print a model instance, you would just get a generic object description like `<Product object (1)>`, which is not very informative.
2. **Admin and Shell Usage**: In the Django admin interface or in the Django shell, the `__str__()` function provides a clear and readable description of each object. For example, instead of seeing just a memory address, you'll see the name of the product.

Khushal Savalakha

3. **Custom Display**: It allows you to control what meaningful data should be shown when the object is represented as a string.

Example Without `__str__()`

```
class Product(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()
```

If you create a `Product` object and try to print it:

```
product = Product(name='Laptop', description='A powerful
gaming laptop')
print(product)
```

The output would look like:

```
<Product object (1)>
```

This output isn't very helpful, as it doesn't tell you anything about the product.

Example With `__str__()`

Now, with the `__str__()` method:

```
class Product(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()

    def __str__(self):
        return self.name
```

If you create the same `Product` object:

```
product = Product(name='Laptop', description='A powerful
gaming laptop')
print(product)
```

Now the output would be:

Khushal Savalakha

```
Laptop
```

This is much clearer and more meaningful, as the product's name is shown.

Deeper Example: Customizing `__str__()`

You can customize the `__str__()` method to display multiple attributes if necessary.

```
class Product(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()

    def __str__(self):
        return f"{self.name} - {self.description[:3
0]}..."  # Show only part of the description
```

Here, the `__str__()` method returns both the name and the first 30 characters of the description. When you print the object, you'll get:

```
Laptop - A powerful gaming laptop...
```

This is helpful for longer descriptions, providing a concise summary of the product.

In Summary:

- **Purpose**: The `__str__()` method defines what should be shown when an object is converted to a string, making it human-readable and useful in Django admin, the shell, or when printing the object.
- **Usage**: Helps in debugging, makes admin representation clearer, and enhances the usability of Django's interactive shell.

By using `__str__()`, you give the model instance a meaningful representation that can be customized to display key information.

💡 it means data is in whatever form user sees in described way in which **str** defined am i true?

Khushal Savalakha

Yes, you're absolutely correct! The `__str__()` method in Django (or Python in general) defines **how the object will appear when it is displayed as a string**—essentially, what users (or developers) will see when they view the object in contexts like:

1. **Django Admin Panel**: The object will be shown using the format defined in the `__str__()` method.
2. **Django Shell**: When you print an object, or it appears in a query result.
3. **Templates**: If you use `{{ object }}` in a Django template, it will show the string as defined in the `__str__()` method.
4. **Logging and Debugging**: If you're debugging or logging the object, the string representation will be what you've defined in `__str__()`.

So yes, the data the user sees when interacting with your model (or when developers print or log model instances) will be in the format you describe within the `__str__()` method.

For example:

```
class Product(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()

    def __str__(self):
        return f"Product Name: {self.name} | Description:
{self.description[:20]}"
```

If you then print the product object or view it in the Django admin:

```
product = Product(name='Laptop', description='A powerful
gaming laptop with high performance.')
print(product)
```

The output would be:

```
Product Name: Laptop | Description: A powerful gaming...
```

This is exactly what users (or developers) will see whenever the object is displayed as a string.

▼ Working with Advanced Template Features:

Khushal Savalakha

▼ Template Inheritance

```
Working with Advanced Template Features:
----------------------------------------
1. Template Inheritance
2. Template Filters
```

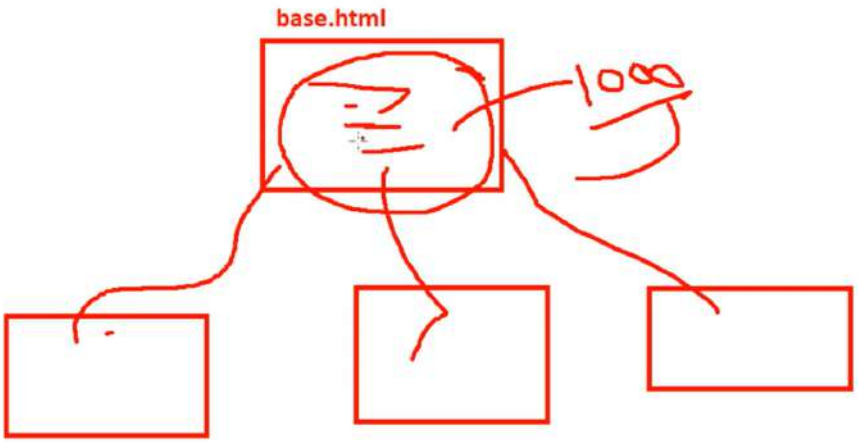
```
1. Template Inheritance:
------------------------
Inheritance--->Code Reusability

If multiple template files have some common code, it is not recommended to write
that common code in every template file. It increases length of the code and
reduces readability and it increases development time.
```

```

We have to separate that common code into a new template file, which is also known
as base template. The remaining template files should required to extend base
template so that common code will be inherited automatically.

```

```
class Parent:
    def property(self):
        print('Gold+Money+Lands+Shares')

class Child(Parent):
    pass

c = Child()
c.property()
```

```
Select Command Prompt
) / '(1,)type'.
  rain_drops = np.zeros(n_drops, dtype=[('position', float, 2),

D:\durgaclasses>py test.py
Gold+Money+Lands+Shares

D:\durgaclasses>
```


Khushal Savalakha




```
How to implement Template Inheritance:
--------------------------------------
base.html:
---------
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    common code required for every child template
    {% block body_block %}
    {% endblock %}

  </body>
</html>
```

```
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    common code required for every child template
    {% block body_block %}
       Anything outside of this block is available to the child template.
       This block should be filled by child template to write its own specific code
    {% endblock %}

  </body>
</html>
```

```
4078 child.html:
4079 -------------
4080 <!DOCTYPE html>
4081 {% extends 'testapp/base.html' %}
4082   {% block body_block %}
4083        child specific extra code
4084   {% endblock %}
4085
```

urls.py

```
from django.contrib import admin
from django.urls import path
from testapp import views


urlpatterns = [
    path('admin/', admin.site.urls),
    path('',views.home_page_view,name='home'),
    path('movies/',views.movies_page_view,name='movies'),
    path('sports/',views.sports_page_view,name='sports'),
    path('politics/',views.politics_page_view,name='polit
]
```

base.html

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
  <head>
```

```
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, in
    {% block title_block %}
    {% endblock %}
    <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/
      rel="stylesheet"
      integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnY
      crossorigin="anonymous"
    />
    <link rel="stylesheet" href="{% static 'css/at2.css' 
  </head>
  <body>
    <nav class="navbar navbar-expand-lg bg-body-tertiary"
        <div class="container-fluid">
          <a class="navbar-brand" href="#">DURGA NEWS</a>
          <button
            class="navbar-toggler"
            type="button"
            data-bs-toggle="collapse"
            data-bs-target="#navbarSupportedContent"
            aria-controls="navbarSupportedContent"
            aria-expanded="false"
            aria-label="Toggle navigation"
          >
            <span class="navbar-toggler-icon"></span>
          </button>
          <div class="collapse navbar-collapse" id="navba
            <ul class="navbar-nav me-auto mb-2 mb-lg-0">
                <li class="nav-item">
                  <a class="nav-link {% if request.path =
                </li>
                <li class="nav-item">
                  <a class="nav-link {% if request.path =
                </li>
                <li class="nav-item">
                  <a class="nav-link {% if request.path =
                </li>
                <li class="nav-item">
```


Khushal Savalakha

```
                    <a class="nav-link {% if request.path =
                  </li>
                </ul>

              <form class="d-flex" role="search">
                <input
                  class="form-control me-2"
                  type="search"
                  placeholder="Search"
                  aria-label="Search"
                />
                <button class="btn btn-outline-success" typ
                  Search
                </button>
              </form>
            </div>
          </div>
        </nav>
      {% block body_block%} {% endblock %}
    </body>
  </html>
```

at2.css

```
h1 {
    color: red !important;
    text-align: center !important;
    font-style: italic !important;
}
/* General styling for all navbar links */
.nav-link {
    color: black !important;
    text-decoration: none !important;
}

/* When a link is active, underline it */
.nav-link.active {
    text-decoration: underline !important;
    font-weight: bold !important; /* Optional: Makes the
```


Khushal Savalakha

```
}

/* Optional: Hover effect to indicate clickability */
.nav-link:hover {
    text-decoration: underline !important;
    color: #007bff !important; /* Change color on hover *
}
```

politics.html

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
    {% block title_block %}
    <title>DURFA NEWS POLITICS PAGE</title>
    {% endblock %}
    {% block body_block %}
        <h1>This is Politics Page</h1>
    {% endblock %}
```

```
1. In the parent template, we can define any number of blocks. But child template
   is responsible to implement these blocks.

2. It is not mandatory to implement every parent block in child. Based on
   requirement,          WE can decide which blocks are required to implement.

3. While implementing blocks in child template, it is not required to follow order.

4. child template can extend any number of parent templates. ie multiple
   inheritance is allowed.
```

```
Advantages of Template Inheritance:
-----------------------------------
1. What ever code available in base template is bydefault available to child
   templates and we are not required to write again. Hence it promotes code
   reusability.

2. It reduces length of the code and improves readability
3. It reduces development time
4. It provides unique and same look and feel for total web application.
```

▼ Template Filters

Khushal Savalakha

```
Template Filters:
----------------
In the template file, the injected data can be displayed by using template tags.
{{name}}      {{emp.eno}}

Before displaying to the end user, if we want to perform some modifications to the
injected text, like add something or cut something, case conversions etc then we
should go for filter.
```

```
Syntax of Template Filter:
--------------------------
{{ value | filtername }}
{{ value | filtername: "argument" }}
Arguments to the filter are optional.
```

```

The Complete Character Code List
Other than the character strings given above, the strftime method takes several
other directives for formatting date values:

%a: Returns the first three characters of the weekday, e.g. Wed.
%A: Returns the full name of the weekday, e.g. Wednesday.
%B: Returns the full name of the month, e.g. September.
%w: Returns the weekday as a number, from 0 to 6, with Sunday being 0.
%m: Returns the month as a number, from 01 to 12.
%p: Returns AM/PM for time.
%y: Returns the year in two-digit format, that is, without the century. For
example, "18" instead of "2018".
%f: Returns microsecond from 000000 to 999999.
%Z: Returns the timezone.
%z: Returns UTC offset.
%j: Returns the number of the day in the year, from 001 to 366.
%W: Returns the week number of the year, from 00 to 53, with Monday being counted
as the first day of the week.
%U: Returns the week number of the year, from 00 to 53, with Sunday counted as the
first day of each week.
```

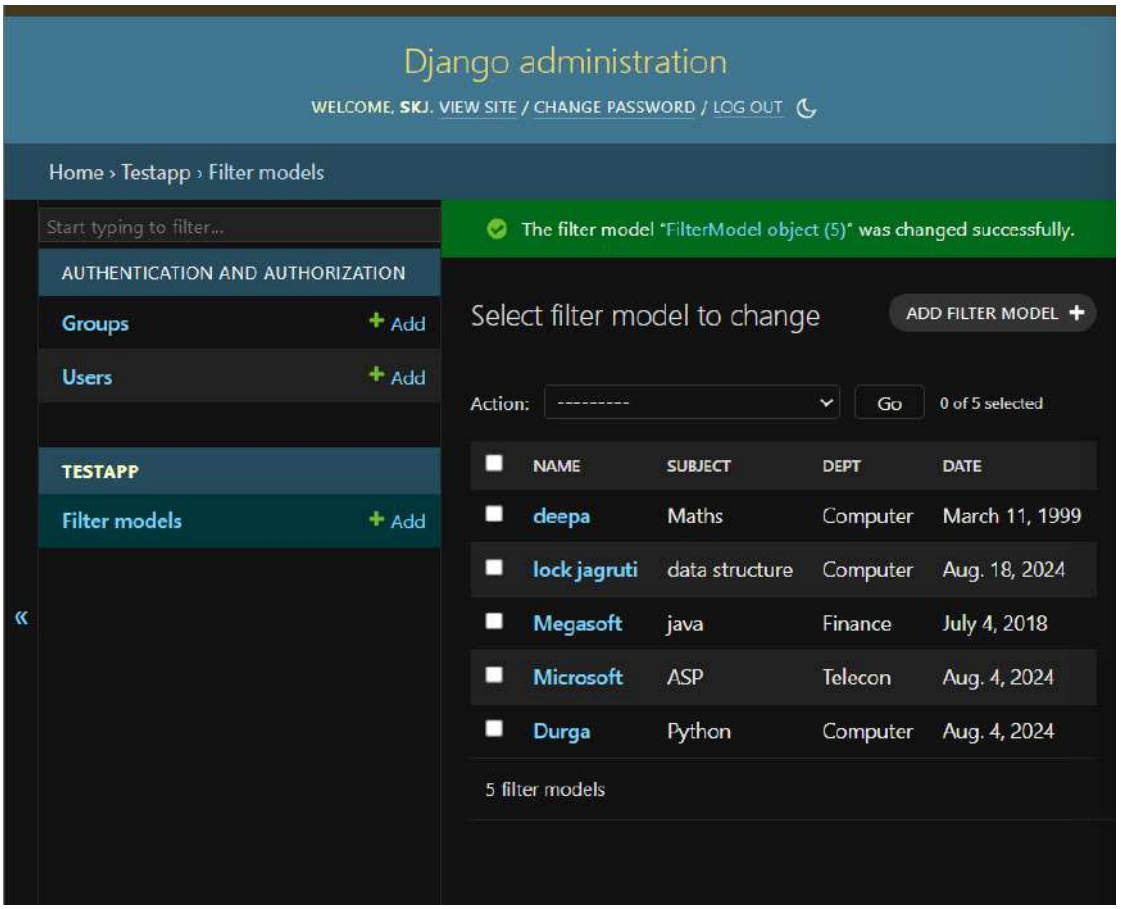

Khushal Savalakha

```
%A: Returns the full name of the weekday, e.g. Wednesday.
%B: Returns the full name of the month, e.g. September.
%w: Returns the weekday as a number, from 0 to 6, with Sunday being 0.
%m: Returns the month as a number, from 01 to 12.
%p: Returns AM/PM for time.
%y: Returns the year in two-digit format, that is, without the century. For example, "18" instead of "2018".
%f: Returns microsecond from 000000 to 999999.
%Z: Returns the timezone.
%z: Returns UTC offset.
%j: Returns the number of the day in the year, from 001 to 366.
%W: Returns the week number of the year, from 00 to 53, with Monday being counted as the first day of the week.
%U: Returns the week number of the year, from 00 to 53, with Sunday counted as the first day of each week.
%c: Returns the local date and time version.
%x: Returns the local version of date.
%X: Returns the local version of time.
```

```
%b: Returns the first three characters of the month name. In our example, it returned "Sep"
%d: Returns day of the month, from 1 to 31. In our example, it returned "15".
%Y: Returns the year in four-digit format. In our example, it returned "2018".
%H: Returns the hour. In our example, it returned "00".
%M: Returns the minute, from 00 to 59. In our example, it returned "00".
%S: Returns the second, from 00 to 59. In our example, it returned "00".
```

```
from django.db import models
# Create your models here.
class FilterModel(models.Model):
    name=models.CharField(max_length=30)
    subject=models.CharField(max_length=30)
    dept=models.CharField(max_length=30)
    date=models.DateField()
```

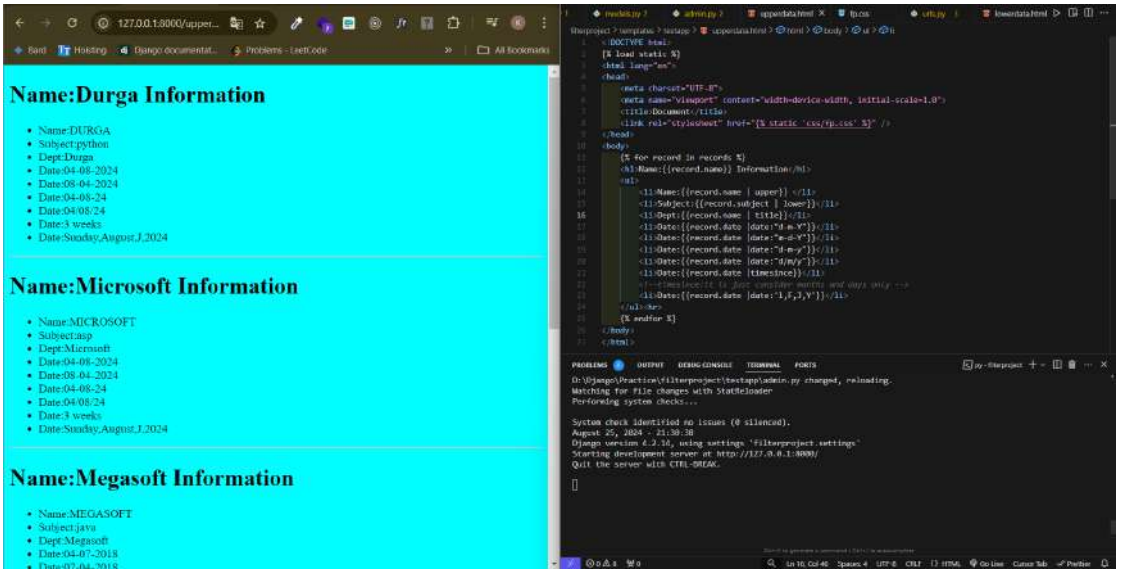

Khushal Savalakha




```
<!DOCTYPE html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, in
    <title>Document</title>
    <link rel="stylesheet" href="{% static 'css/fp.css' %
</head>
<body>
    {% for record in records %}
    <h1>Name:{{record.name}} Information</h1>
    <ul>
        <li>Name:{{record.name | upper}} </li>
        <li>Subject:{{record.subject | lower}}</li>
        <li>Dept:{{record.name | title}}</li>
        <li>Date:{{record.date |date:"d-m-Y"}}</li>
        <li>Date:{{record.date |date:"m-d-Y"}}</li>
```


Khushal Savalakha

```
        <li>Date:{{record.date |date:"d-m-y"}}</li>
        <li>Date:{{record.date |date:"d/m/y"}}</li>
        <li>Date:{{record.date |timesince}}</li>
        <!--timesince:it is just consider months and days
        <li>Date:{{record.date |date:'l,F,J,Y'}}</li>
    </ul><hr>
    {% endfor %}
</body>
</html>
```




```
How to create our own filters:
------------------------------
Based on our requirement, we can create our own filters, if pre defined filters are
not fullfill our requirement.

steps:
------
1. create a folder 'templatetags' inside our application folder
2. Create a special file named with '__init__.py' inside this folder(templatetags),
so that django will consider this folder as a python package.
3. Create a python file inside this folder to define our own filters.
  cust_filters.py ----->It can be anyname
```

here “templatetags” name only manadatory(compulsory) .we can change file name of __init__.py and cust_filters.py this is not mandetory.

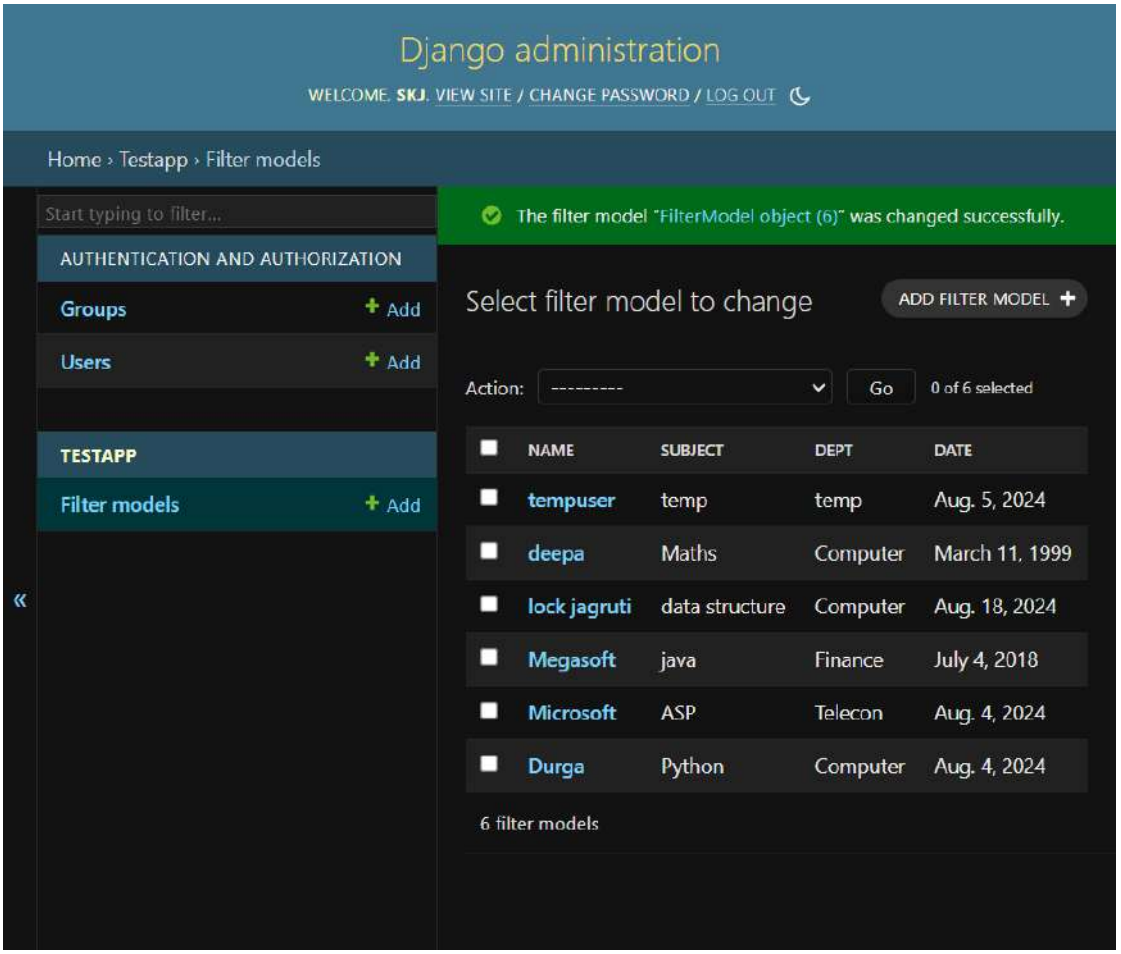
cust_filters.py

```
from django import template
register=template.Library()

def first_five_upper(value):
```

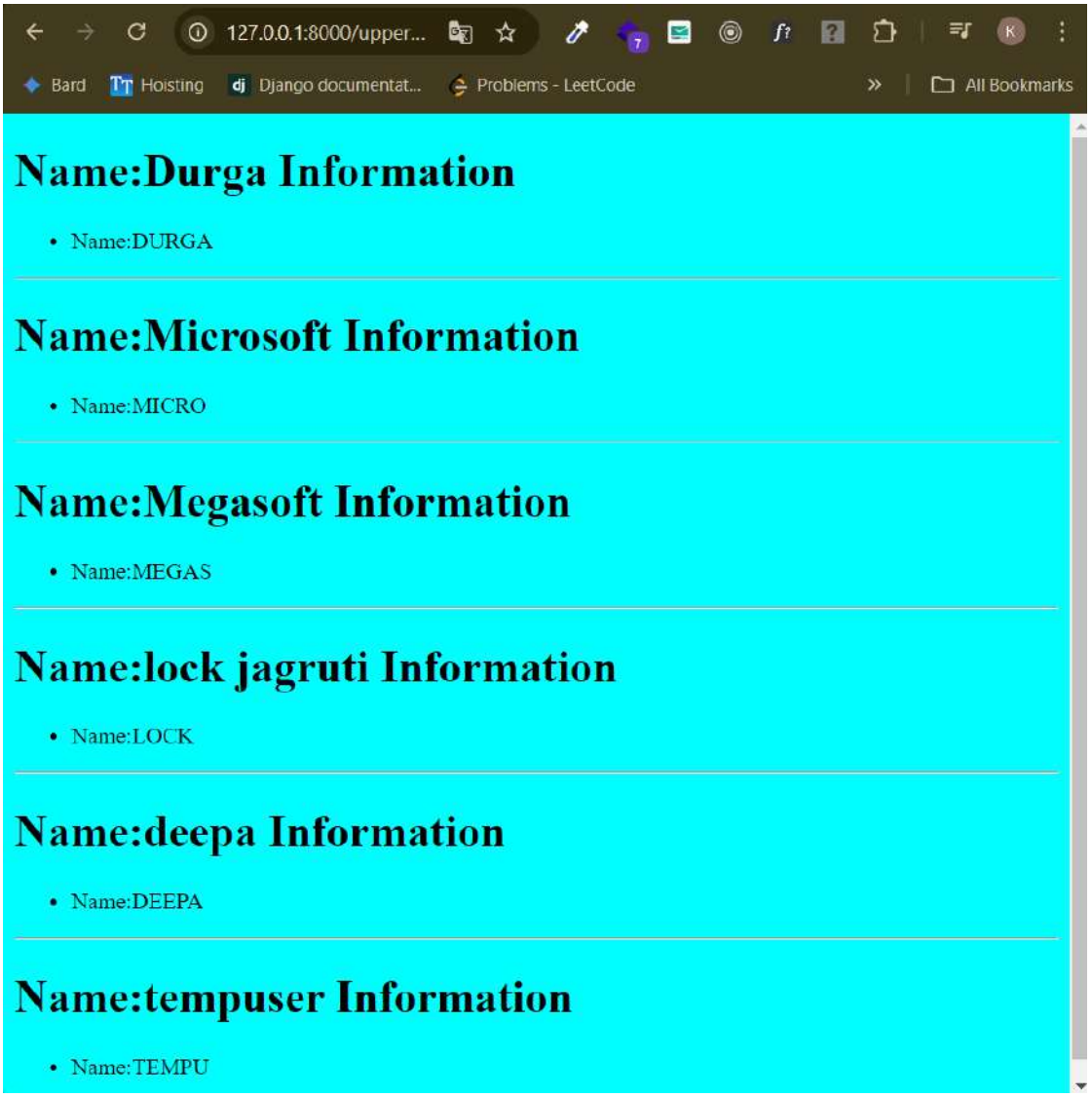
```
    result=value[:5].upper()
    return result
#only 5 charcters with uppercase it will return
# register.filter('first_five_upper',first_five_upper)
register.filter('ffu',first_five_upper)
```

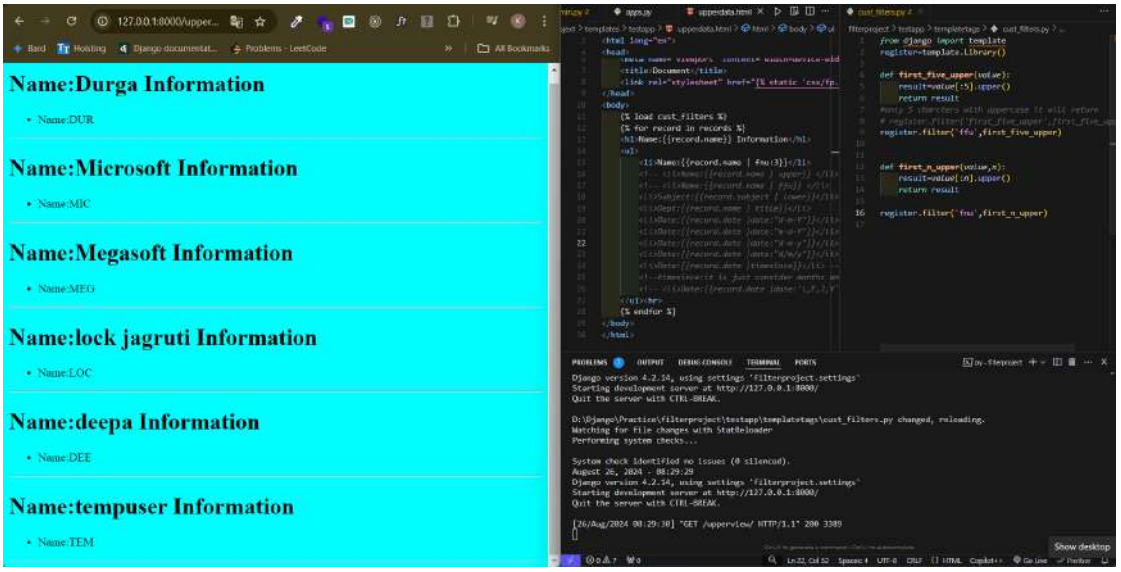
```
<!DOCTYPE html>
{%  load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, in
    <title>Document</title>
    <link rel="stylesheet" href="{% static 'css/fp.css' %
</head>
<body>
    {% load cust_filters %}
    {% for record in records %}
    <h1>Name:{{record.name}} Information</h1>
    <ul>
        <li>Name:{{record.name | ffu}}</li>
    </ul><hr>
    {% endfor %}
</body>
</html>
```

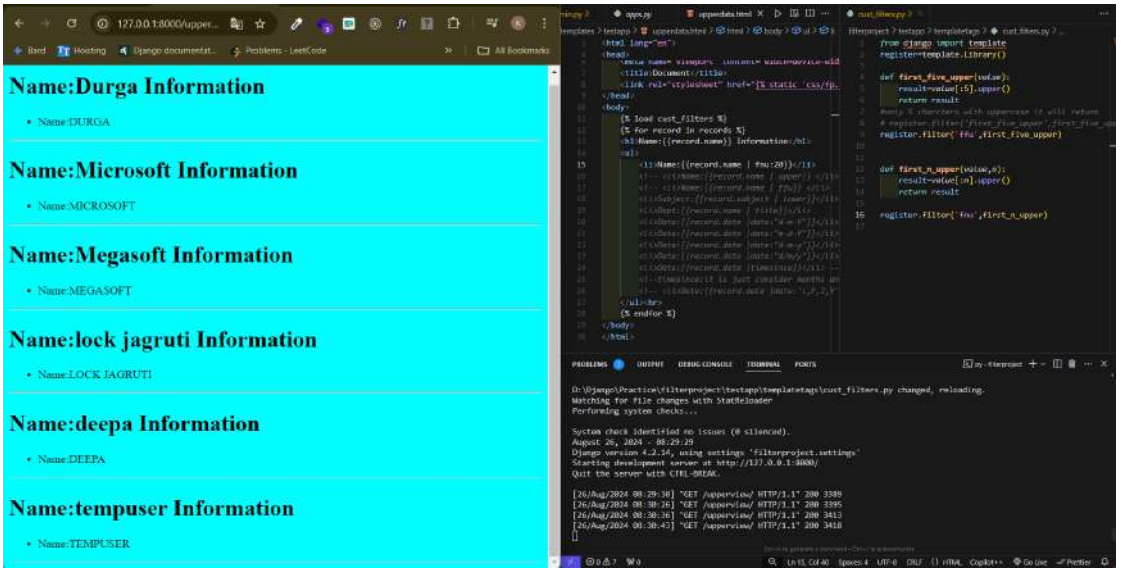

Khushal Savalakha

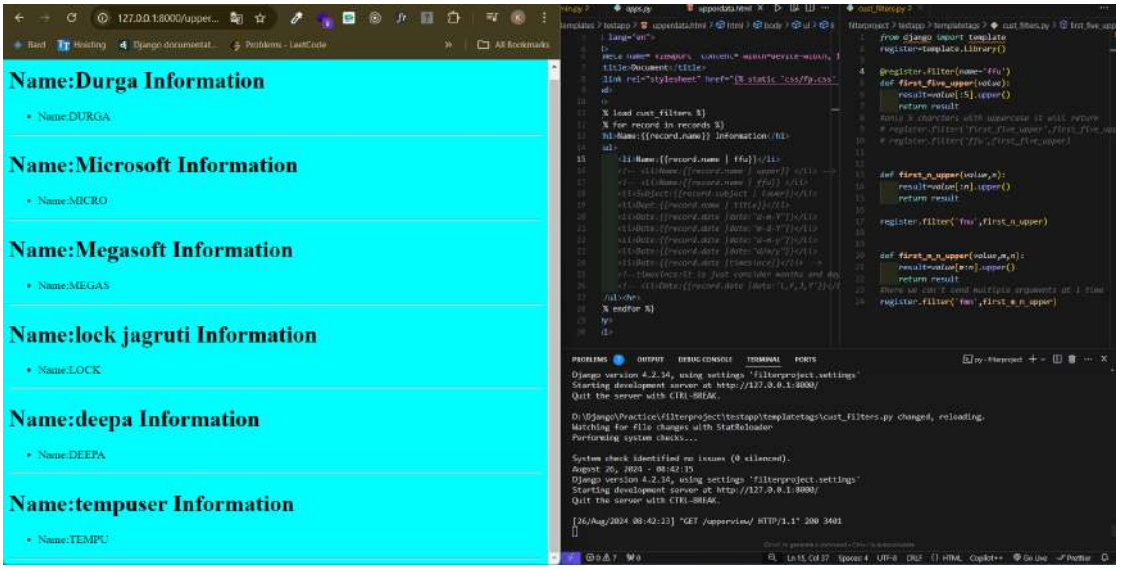


Khushal Savalakha


127.0.0.1:8000/upper...
Bard
Hoisting
Django documentat...
Problems - LeetCode
All Bookmarks
Name:Durga Information
Name:DURGA
Name:Microsoft Information
Name:MICRO
Name:Megasoft Information
Name:MEGAS
Name:lock jagruti Information
Name:LOCK
Name:deepa Information
Name:DEEPA
Name:tempuser Information
Name:TEMPU


Name:Durga Information
Name:DUR
Name:Microsoft Information
Name:MIC
Name:Megasoft Information
Name:MEG
Name:lock jagruti Information
Name:LOC
Name:deepa Information
Name:DEE
Name:tempuser Information
Name:TEM





filterproject\templates\testapp\upperdata.html

```
<!DOCTYPE html>
{%  load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, in
    <title>Document</title>
    <link rel="stylesheet" href="{% static 'css/fp.css' %
</head>
<body>
    {% load cust_filters %}
```


Khushal Savalakha

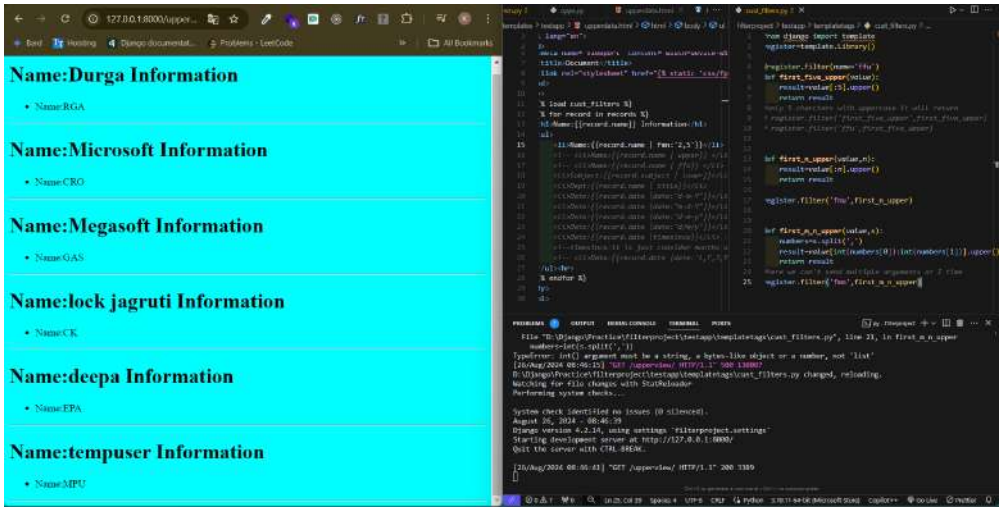
```
    {% for record in records %}
    <h1>Name:{{record.name}} Information</h1>
    <ul>
        <li>Name:{{record.name | fmn:'2,5'}}</li>
        <!-- <li>Name:{{record.name | upper}} </li> -->
        <!-- <li>Name:{{record.name | ffu}} </li>
        <li>Subject:{{record.subject | lower}}</li>
        <li>Dept:{{record.name | title}}</li>
        <li>Date:{{record.date |date:"d-m-Y"}}</li>
        <li>Date:{{record.date |date:"m-d-Y"}}</li>
        <li>Date:{{record.date |date:"d-m-y"}}</li>
        <li>Date:{{record.date |date:"d/m/y"}}</li>
        <li>Date:{{record.date |timesince}}</li> -->
        <!--timesince:it is just consider months and days
        <!-- <li>Date:{{record.date |date:'l,F,J,Y'}}</li
    </ul><hr>
    {% endfor %}
</body>
</html>
```

filterproject\testapp\templatetags\cust_filters.py

```
from django import templat
register=template.Library()

@register.filter(name='ffu')
def first_five_upper(value):
    result=value[:5].upper()
    return result
#only 5 charcters with uppercase it will return
# register.filter('first_five_upper',first_five_upper)
# register.filter('ffu',first_five_upper)


def first_n_upper(value,n):
    result=value[:n].upper()
    return result

register.filter('fnu',first_n_upper)
```


Khushal Savalakha

```
def first_m_n_upper(value,s):
    numbers=s.split(',')
    result=value[int(numbers[0]):int(numbers[1])].upper()
    return result
#here we can't send multiple arguments at 1 time
register.filter('fmn',first_m_n_upper)
```



▼ Session Management [Pending Lec:35-41 page:83-102]

▼ User Authentication And Authorization

```
User Authentication and Authorization:
--------------------------------------
Authentication: The process of validating user
Authorization: The process of validating access permissions of a user



login page
user name and password --->should be stored for future purpose
validation must be required

auth application

1. django.contrib.auth
2. django.contrib.contenttypes
```


Khushal Savalakha

▼ Implementing Log in Functionality

login.html

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
    {% block body_block %}
    <h1>Home Page</h1>
        <div class="container">
         <h1>Please Login to write exams....</h1>
         <form method="post">
            {{form.as_p}}
            {% csrf_token %}
            <input type="submit" value="Login">
         </form>
        </div>
    {% endblock %}
```

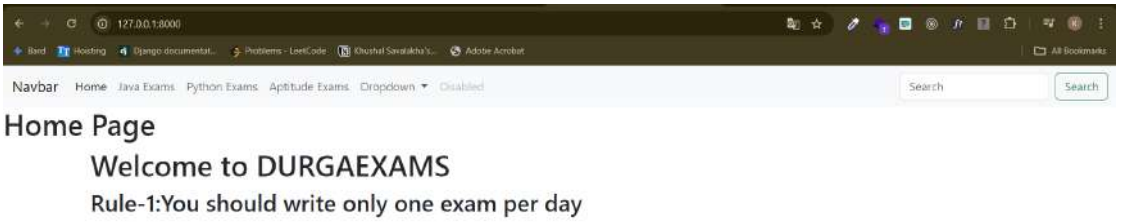
views.py

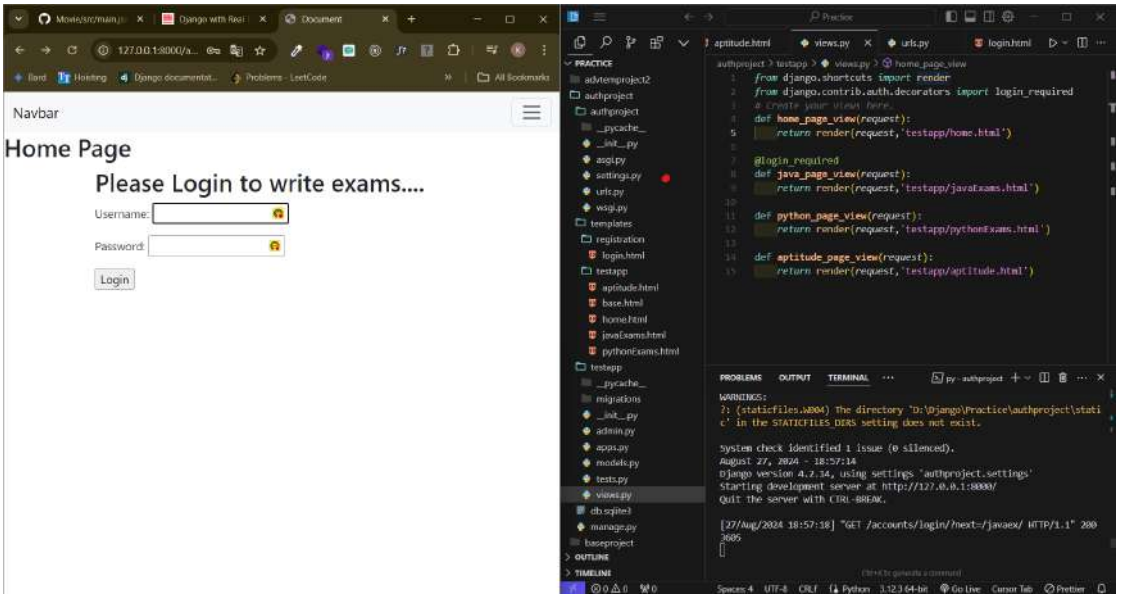
```
from django.shortcuts import render
from django.contrib.auth.decorators import login_required
# Create your views here.
def home_page_view(request):
    return render(request,'testapp/home.html')

@login_required
def java_page_view(request):
    return render(request,'testapp/javaExams.html')

def python_page_view(request):
    return render(request,'testapp/pythonExams.html')

def aptitude_page_view(request):
    return render(request,'testapp/aptitude.html')
```

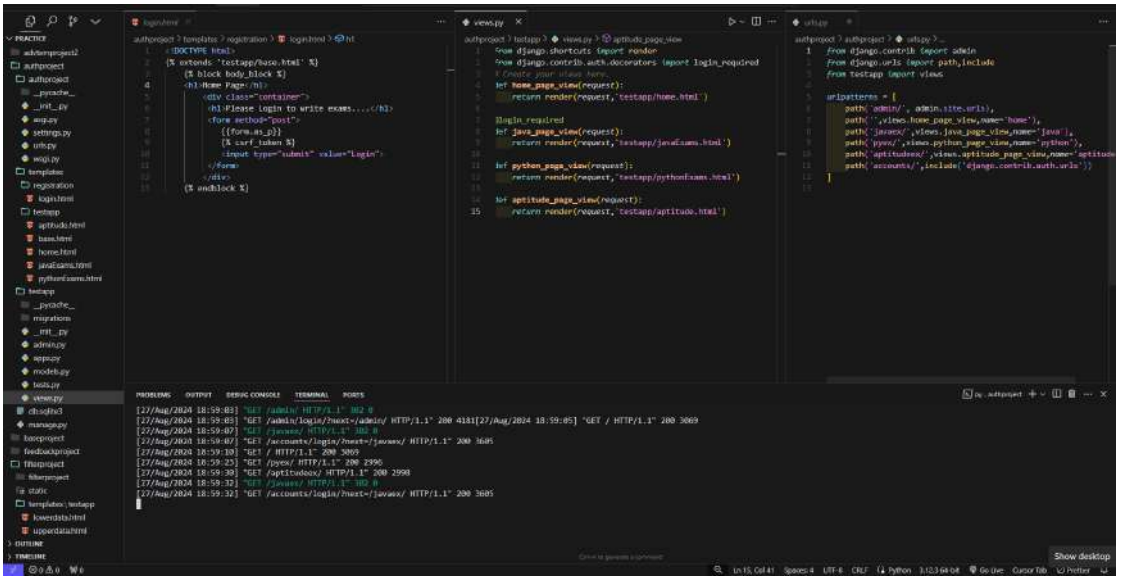
urls.py

Khushal Savalakha

```
from django.contrib import admin
from django.urls import path,include
from testapp import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('',views.home_page_view,name='home'),
    path('javaex/',views.java_page_view,name='java'),
    path('pyex/',views.python_page_view,name='python'),
    path('aptitudeex/',views.aptitude_page_view,name='apt
    path('accounts/',include('django.contrib.auth.urls'))
]
```







explain here how authentication works in django with simple way with example .

How Authentication Works in Django

Django's authentication system is designed to handle user authentication (logging in and out), as well as permissions (who can do what). Here’s a simplified explanation based on the code and setup in the image you provided.

1. Setting Up Authentication

- **URL Configuration (`urls.py`):**
  In your `urls.py` file, you’ve included Django’s built-in authentication URLs with this line:

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

  This includes routes for login, logout, password reset, etc. Django automatically handles these routes once included.

- **Login Page (`login.html`):**
  Your `login.html` file is set up to display a login form. The `{% csrf_token %}` tag is included to protect against Cross-Site Request Forgery (CSRF) attacks. The form’s action is not specified, so it will submit to the same URL where the login form is served, typically `/accounts/login/`.

Khushal Savalakha

2. Using the Login Required Decorator

- **Protecting Views (`views.py`):**
  In `views.py`, you've imported `login_required` from `django.contrib.auth.decorators`:
  Then, you've used it to protect views that should only be accessible to authenticated users:
  This means that only logged-in users can access the `java_page_view`. If a user is not logged in and tries to access this view, they will be redirected to the login page.

```
from django.contrib.auth.decorators import login_req
uired
```

```
@login_required
def java_page_view(request):
    return render(request, 'testapp/javaExams.html')
```

3. Login Process:

- When a user tries to access a protected view (e.g., `/javaex/`), Django checks if the user is authenticated.
- If the user is **not authenticated** (not logged in), they are redirected to the login page (`/accounts/login/`).
- If the user is **authenticated** (logged in), they can access the view, and the corresponding template (e.g., `javaExams.html`) is rendered.

4. Logout Process:

- Django also provides a built-in view for logging out, typically available at `/accounts/logout/`. When a user logs out, they are usually redirected to a page like the homepage or the login page.

Example Workflow:

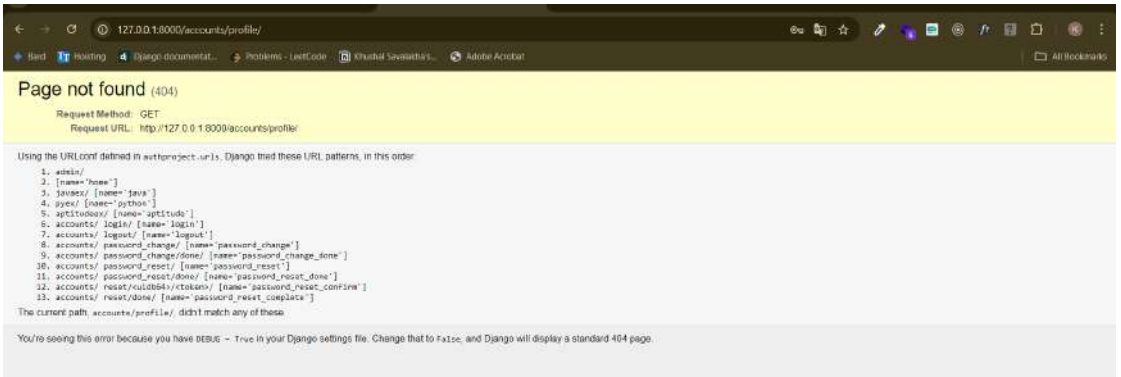
1. **Accessing a Protected Page:**
   - A user attempts to access `/javaex/`.
   - Since `java_page_view` is decorated with `@login_required`, Django checks if the user is logged in.

Khushal Savalakhia

2. **Redirect to Login Page:**
    - If the user is not logged in, they are redirected to `/accounts/login/`.
    - The `login.html` form is displayed. The user enters their credentials.
3. **Successful Login:**
    - If the credentials are correct, the user is authenticated and redirected back to the originally requested page (`/javaex/`).
4. **Logging Out:**
    - The user clicks a "Logout" link, which leads to `/accounts/logout/`.
    - Django logs the user out and redirects them, typically to the homepage.

Summary

Django's authentication system simplifies managing user authentication by providing built-in views and decorators. You can protect specific views using `@login_required`, ensuring only authenticated users can access certain content. The process involves redirecting users to the login page when needed and managing user sessions efficiently.



The error occurs because Django's built-in authentication system tries to redirect users to a profile page at the URL `/accounts/profile/` after a successful login. Since you haven't created a view or URL pattern to handle this URL, Django raises a "Page not found (404)" error.

Solutions

1. Redirect to a Custom Page After Login:

You can customize where Django redirects users after they log in by setting the `LOGIN_REDIRECT_URL` in your `settings.py`. For example, if you want to redirect users to the homepage:

```
# settings.py
LOGIN_REDIRECT_URL = '/'
```

This will redirect users to the home page after they log in.

2. Create a Profile View:

If you want to keep the `/accounts/profile/` URL, you can create a view and template for the profile page.

1. **Create a View:**

   ```
   # views.py
   from django.shortcuts import render
   from django.contrib.auth.decorators import login_req
   uired

   @login_required
   def profile_view(request):
       return render(request, 'profile.html')
   ```

2. **Add the URL Pattern:**

   ```
   # urls.py
   from django.urls import path
   from . import views

   urlpatterns = [
       # other URL patterns
       path('accounts/profile/', views.profile_view, na
   me='profile'),
   ]
   ```

3. **Create the Template:**

   Create a `profile.html` template in your `templates` directory.

Khushal Savalakha

```
<!-- profile.html -->
<h1>Welcome to your profile, {{ user.username }}!</h
1>
```

This will create a profile page that users will be redirected to after they log in.

Conclusion

If you want to avoid creating a profile page, just set `LOGIN_REDIRECT_URL` in your `settings.py`. If you want to implement a profile page, create the view, URL, and template as described above.

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block %}
<div class="container">
<h1>Please Login to write exams....</h1>
<form method="post">
{{form.as_p}}
{% csrf_token %}
<input type="submit" value="Login">
</form>
</div>
{% endblock %}
```

💡 here when i loggein and click on login button it still shows login page tell me which condition i have to make by which if user logged in then it shows profile.html directly?

```
from django.shortcuts import render
from django.contrib.auth.decorators import login_required
# Create your views here.
def home_page_view(request):
    return render(request,'testapp/home.html')

@login_required
```


Khushal Savalakha

```
def java_page_view(request):
    return render(request,'testapp/javaExams.html')

def python_page_view(request):
    return render(request,'testapp/pythonExams.html')

def aptitude_page_view(request):
    return render(request,'testapp/aptitude.html')

def profile_view(request):
    return render(request,'registration/profile.html')
```

💡 explain here how it checks user.is_authenticated ?

In your Django template, the `{% if user.is_authenticated %}` condition is used to check whether the user is currently authenticated (i.e., logged in). Here's how it works:

Understanding `user.is_authenticated`:

1. **Context Processor**:
   - Django automatically adds a `user` object to the context of every template if you have the `django.contrib.auth.context_processors.auth` context processor enabled, which it usually is by default.
   - This `user` object represents the currently logged-in user.
2. `is_authenticated` **Attribute**:
   - The `user` object has an attribute called `is_authenticated`. This attribute is a boolean that returns `True` if the user is logged in and `False` if the user is not logged in (i.e., they are an anonymous user).
   - `is_authenticated` is always `True` for logged-in users and `False` for anonymous users.
3. **Template Logic**:
   - When your template runs, it checks the value of `user.is_authenticated`.
   - If the user is authenticated (`True`), the code inside the `{% if %}` block executes. In this case, the `<script>` tag with the redirect code is executed, which will redirect the authenticated user to the `profile` page.

Khushal Savalakha

- If the user is not authenticated (`False`), the script won't execute, and the login form will be displayed.

Example Breakdown:

```
{% if user.is_authenticated %}
    <script>
        window.location.href = "{% url 'profile' %}";
    </script>
{% endif %}
```

- `{% if user.is_authenticated %}`: This checks whether the user is authenticated.
- `<script> window.location.href = "{% url 'profile' %}"; </script>`: If the user is authenticated, this JavaScript code is executed, which redirects the user to the profile page.
- `{% endif %}`: This ends the `if` block.

How It Fits in Your Views:

In your Django views, you have the `profile_view` function that renders the `profile.html` template:

```
def profile_view(request):
    return render(request,'registration/profile.html')
```

When a user logs in, if they visit a page where the `user.is_authenticated` check is present, it will automatically redirect them to the `profile_view`.

Summary:

- The `user.is_authenticated` check is a built-in feature of Django's authentication system. It determines whether a user is logged in or not.
- The template uses this check to conditionally execute a redirect script if the user is already authenticated.
- This prevents logged-in users from seeing the login form and instead directs them to their profile page immediately.

Important Notes:

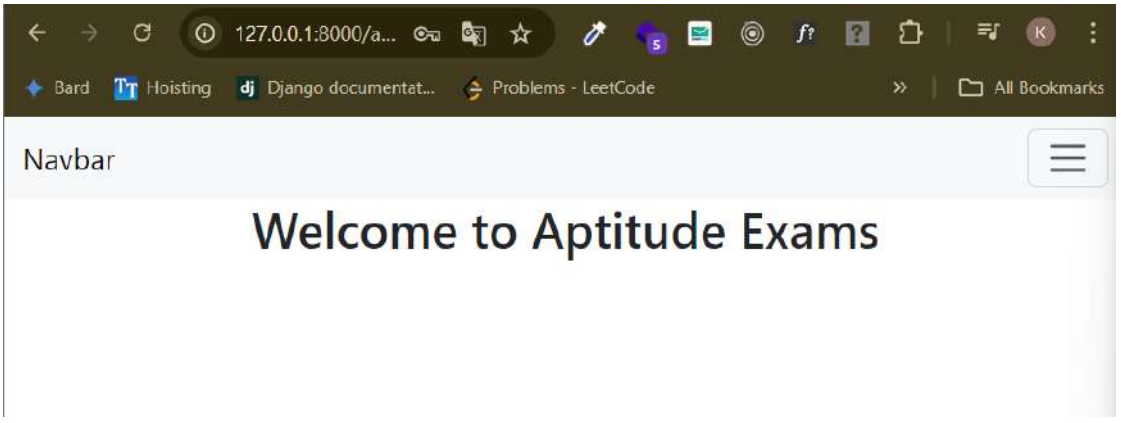
Khushal Savalakha

- Ensure that your Django project is using the `django.contrib.auth` application, as it provides the necessary middleware and context processors for authentication-related features like `user.is_authenticated`.
- The `login_required` decorator in your views ensures that only authenticated users can access specific views, adding another layer of access control.

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/5.0/howto/static-file

STATIC_URL = 'static/'
STATICFILES_DIRS=[
    BASE_DIR/'static'
]

LOGIN_REDIRECT_URL='/'
```

if we write this then it will directly redirect to home page.

```
STATIC_URL = 'static/'
STATICFILES_DIRS=[
    BASE_DIR/'static'
]

LOGIN_REDIRECT_URL='/aptitudeex'
```

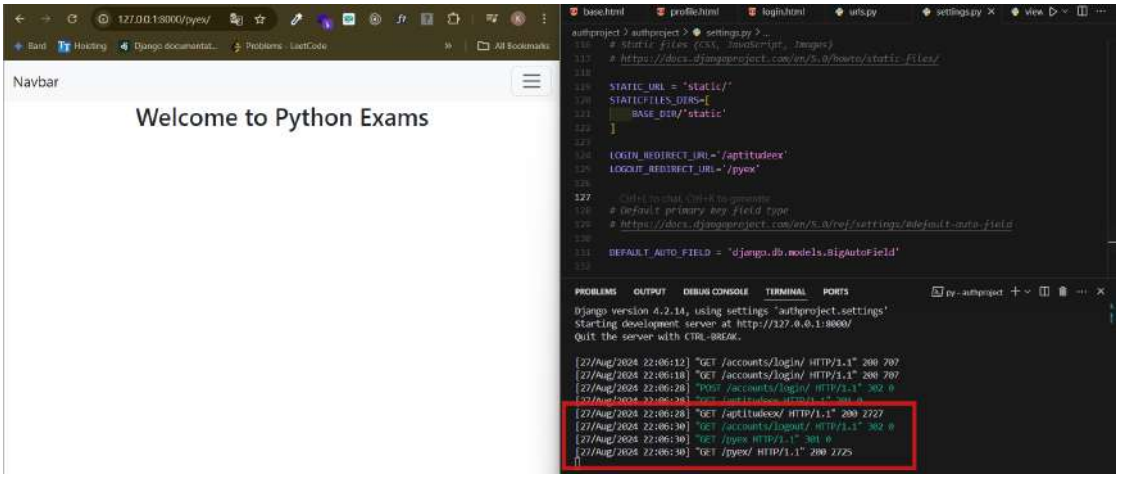


```
If we click login button, After login the next page is profile page bydefault.
If we want to configure our own destination page, we have to write the following line in settings.py

LOGIN_REDIRECT_URL='/'

step-1: Update login link with the url pattern: /accounts/login
step-2: configure destination page inside settings.py
        LOGIN_REDIRECT_URL='/'
```

▼ Implementing Log Out Functionality

```
LOGIN_REDIRECT_URL='/'

step-1: Update login link with the url pattern: /accounts/login
         <a class="nav-link" href="/accounts/login">Login</a>
step-2: configure destination page inside settings.py
        LOGIN_REDIRECT_URL='/'

after getting homepage then we acces all pages without log in

Implementing logout functionality:
----------------------------------
step-1: Update logout link with the url pattern: /accounts/logout
         <a class="nav-link" href="/accounts/logout">Logout</a>

Step-2:
  The default destination page for logout is admin logout page
  We can configure our own logout destination page
```

```
STATIC_URL = 'static/'
STATICFILES_DIRS=[
    BASE_DIR/'static'
]

LOGIN_REDIRECT_URL='/aptitudeex'
LOGOUT_REDIRECT_URL='/pyex'
```



Khushal Savalakha

```
 <a class="nav-link" href="{% url 'login' %}">Login</a>
<a class="nav-link" href="{% url 'logout' %}">Logout</a>
<a class="nav-link" href="../accounts/login">Login</a>
```

```
5545 views.py:
5546 ---------
5547 def logout_view(request):
5548     return render(request,'testapp/logout.html')
5549 urls.py:
5550 ---------
5551 path('logout/', views.logout_view),
5552
5553 settings.py:
5554 ----------
5555 LOGOUT_REDIRECT_URL='/logout'
5556
5557
5558
```

```
STATIC_URL = 'static/'
STATICFILES_DIRS=[
    BASE_DIR/'static'
]

LOGIN_REDIRECT_URL='/aptitudeex'
LOGOUT_REDIRECT_URL='/logout'
```

urls.py

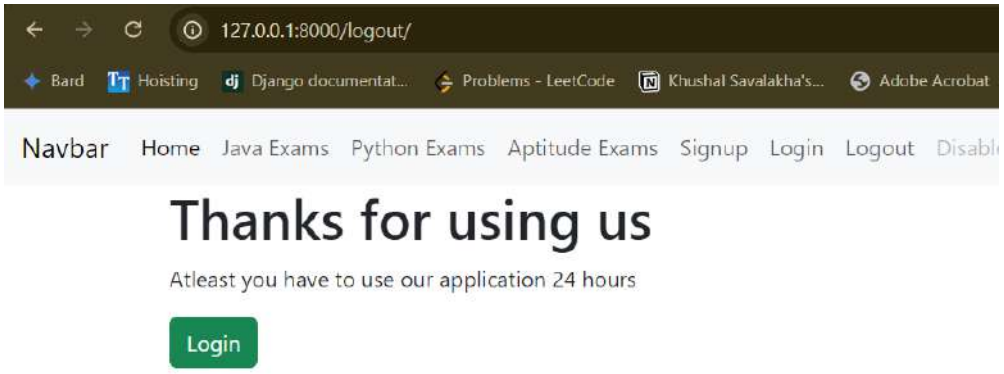
```
from django.contrib import admin
from django.urls import path,include
from testapp import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('',views.home_page_view,name='home'),
    path('javaex/',views.java_page_view,name='java'),
    path('pyex/',views.python_page_view,name='python'),
    path('aptitudeex/',views.aptitude_page_view,name='apt
    path('accounts/',include('django.contrib.auth.urls'),
```


Khushal Savalakha

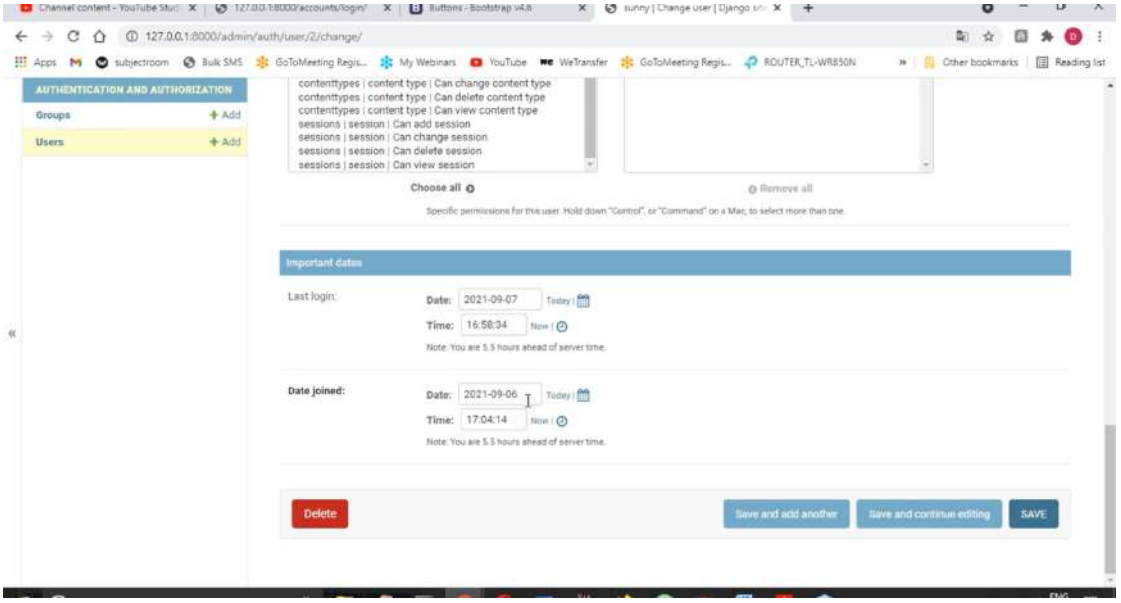
```
    # path('accounts/',include('django.contrib.auth.urls'
    path('accounts/profile/', views.profile_view, name='p
    path('logout/',views.logout_view)
]
```

views.py

```
from django.shortcuts import render
from django.contrib.auth.decorators import login_required
# Create your views here.
def home_page_view(request):
    return render(request,'testapp/home.html')

@login_required
def java_page_view(request):
    return render(request,'testapp/javaExams.html')

def python_page_view(request):
    return render(request,'testapp/pythonExams.html')

def aptitude_page_view(request):
    return render(request,'testapp/aptitude.html')

def profile_view(request):
    return render(request,'registration/profile.html')

def logout_view(request):
    return render(request,'testapp/logout.html')
```



▼ Implementing Signup Functionality & Changing password Algorithms



views.py

```
def signup_view(request):
    form=SignUpForm()
    return render(request,'testapp/signup.html',{'form':f
```

signup.html

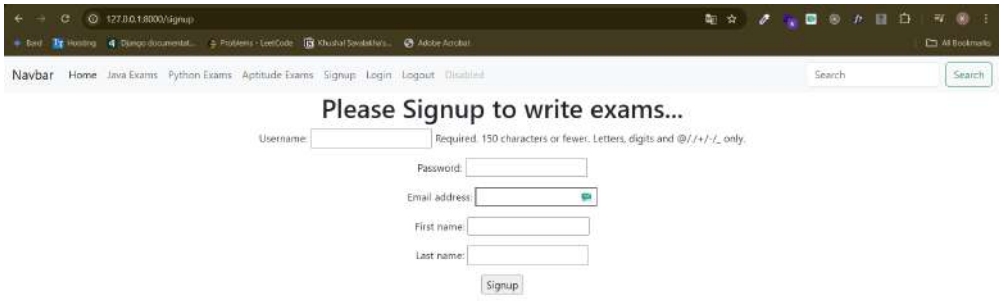
```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
    {% block body_block %}
        <div class="container" align="center">
            <h1>Please Signup to write exams...</h1>
            <form method="post">
                {{form.as_p}}
                {% csrf_token %}
                <input type="submit" name="" value="Signu
            </form>
```


Khushal Savalakha

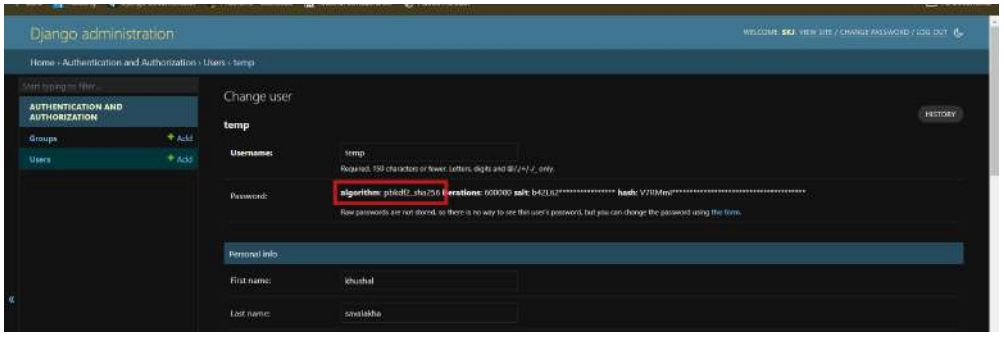
```
        </div>
    {% endblock %}
```

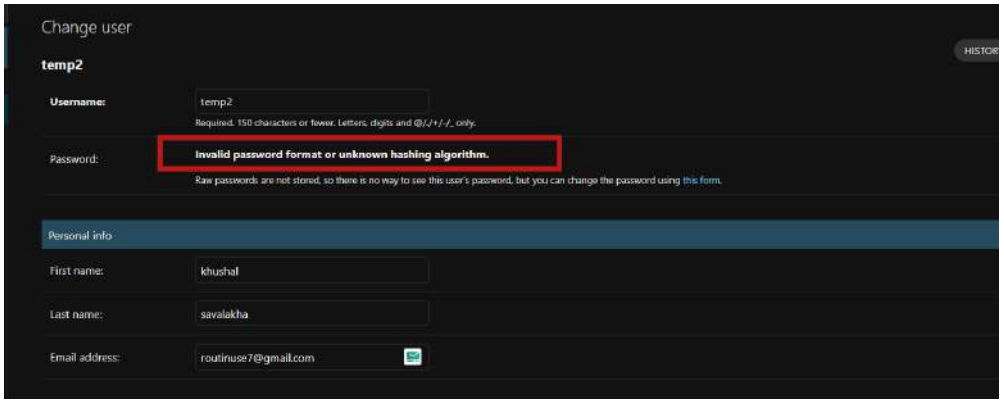
forms.py

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

class SignUpForm(forms.ModelForm):
    class Meta:
        model=User
        #here there is too many fields in User
        #  because it is default database like permission
        # fields='__all__'
        fields=['username','password','email','first_name
```




```
5606 in django auth application, user model, the passwords  should not be saved
     directly. It should be hashed by using some security algorithm. The default
     password hashing algorithm: pbkdf2_sha256
5607
5608 form.save()--->We are trying to save password in the plain text form.
5609 Invalid password format or unknown hashing algorithm.
5610
5611
5612 views.
5613 from django.http import HttpResponseRedirect
5614 def signup_view(request):
5615     form = SignUpForm()
5616     if request.method == 'POST':
5617         form = SignUpForm(request.POST)
5618         user = form.save()
5619         user.set_password(user.password)
5620         user.save()
5621         return HttpResponseRedirect('/accounts/login')
5622     return render(request,'testapp/signup.html',{'form':form})
```





views.py

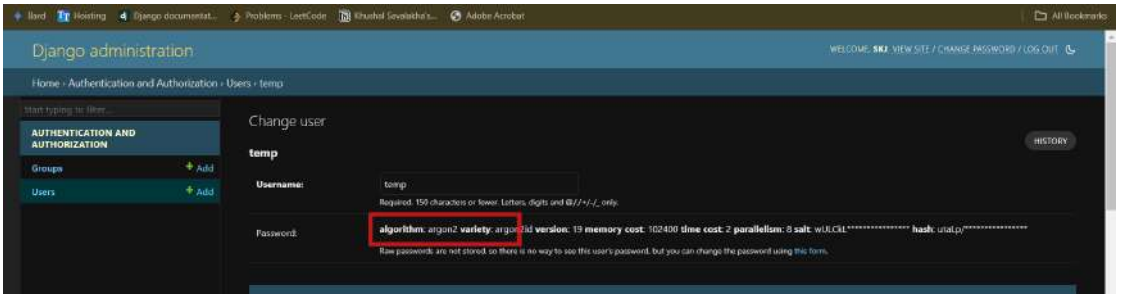
```
def signup_view(request):
    form=SignUpForm()
    if request.method=='POST':
        form=SignUpForm(request.POST)
        user=form.save()
        user.set_password(user.password)#for incription o
        user.save()
        return HttpResponseRedirect('/accounts/login')
    return render(request,'testapp/signup.html',{'form':f
```

```
password hashers:
----------------
The default password hasher: pbkdf2_sha256
We can use other more secured password hashers also
like argon2,bcrypt etc

pip install bcrypt
pip install django[argon2]


More secured algorithm is argon2 followed by bcrypt and then pbkdf2_sha256.
```

In settings.py,we have to configure password hashers as follows.

```
PASSWORD_HASHERS=[
'django.contrib.auth.hashers.Argon2PasswordHasher',
'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
'django.contrib.auth.hashers.BCryptPasswordHasher',
'django.contrib.auth.hashers.PBKDF2PasswordHasher',
'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher'
]

The order is important.
First available hasher will be considered from top to bottom.
```

```
# Password Hashers
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.Argon2PasswordHasher',
    'django.contrib.auth.hashers.BCryptSHA256PasswordHash
    'django.contrib.auth.hashers.BCryptPasswordHasher',
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher
]
```



▼ CRUD operations By Using Both CBVs and FBVs

▼ Function Based Views(FBVs)

▼ ***READ AND INSERT***

```
3. Social login

CRUD Operations:
----------------
FBVS--->Function Based Views
CBVS--->Class Based Views
Django ORM


C --->Create (Insert Operation)
R --->Retrieve/Read (Select query)
U ---->Update  (update)
D --->Delete  (delete)
```

forms.py

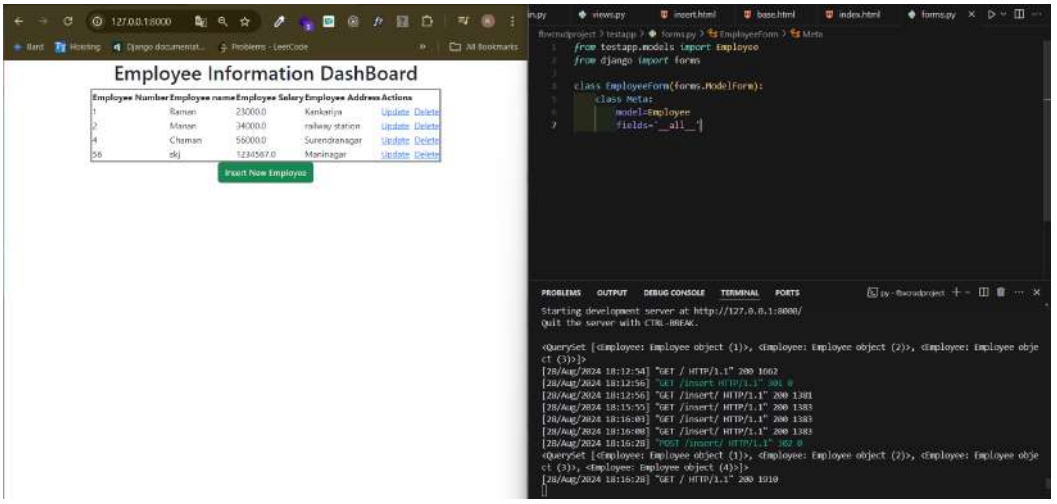
```
from testapp.models import Employee
from django import forms

class EmployeeForm(forms.ModelForm):
    class Meta:
        model=Employee
        fields='__all__'
```

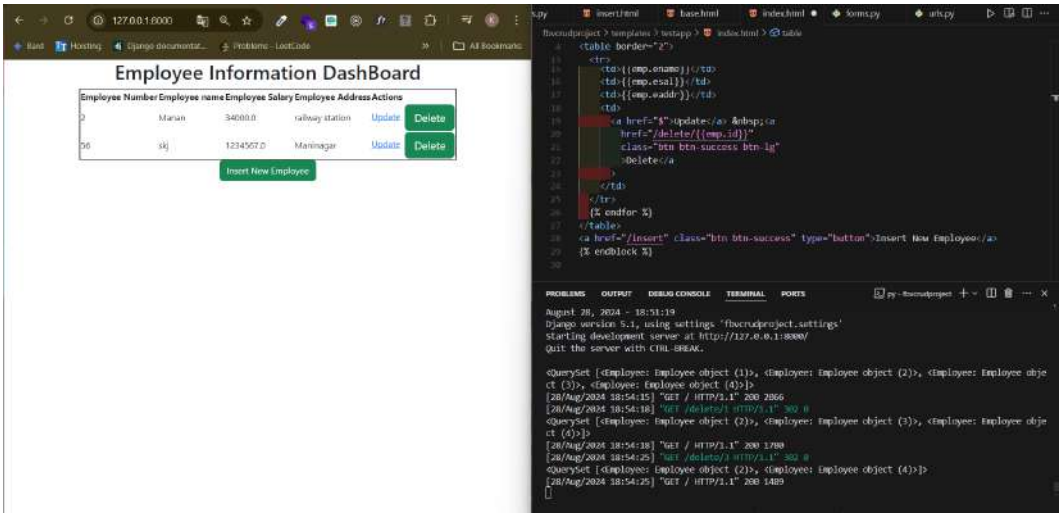
views.py

```
from django.shortcuts import render,redirect
from testapp.models import Employee
from testapp.forms import EmployeeForm
# Create your views here.
def retrive_view(request):
    emp_list=Employee.objects.all()
    print(emp_list)
    return render(request,'testapp/index.html',{'emp_l

def insert_view(request):
    form=EmployeeForm()
    if request.method=='POST':
```


Khushal Savalakha

```
        form=EmployeeForm(request.POST)
        if form.is_valid():
            form.save()
        return redirect('/')
    return render(request,'testapp/insert.html',{'form
```





Khushal Savalakhia

▼ ***DELETE AND UPDATE***

```
7. Delete record:
------------------

1. <a href='/delete/{{emp.id}}'>Delete</a>
  /delete/12

2. path('delete/<int:id>', views.delete_view),

3.
def delete_view(request,id):
    employee = Employee.objects.get(id=id)
    employee.delete()
    return redirect('/')

```

```
8. Update operation:
--------------------
step-1: <a href="/update/{{emp.id}}" class="btn btn-primary">Update</a>

step-2: urls.py
path('update/<int:id>', views.update_view),


step-3:
def update_view(request,id):
    employee = Employee.objects.get(id=id)
    form = EmployeeForm(instance = employee)
    return render(request,'testapp/update.html',{'form':form})


```

urls.py

```
from django.contrib import admin
from django.urls import path
from testapp import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('',views.retrive_view),
    path('insert/',views.insert_view),
    path('delete/<int:id>',views.delete_view),
    path('update/<int:id>',views.update_view)
]
```

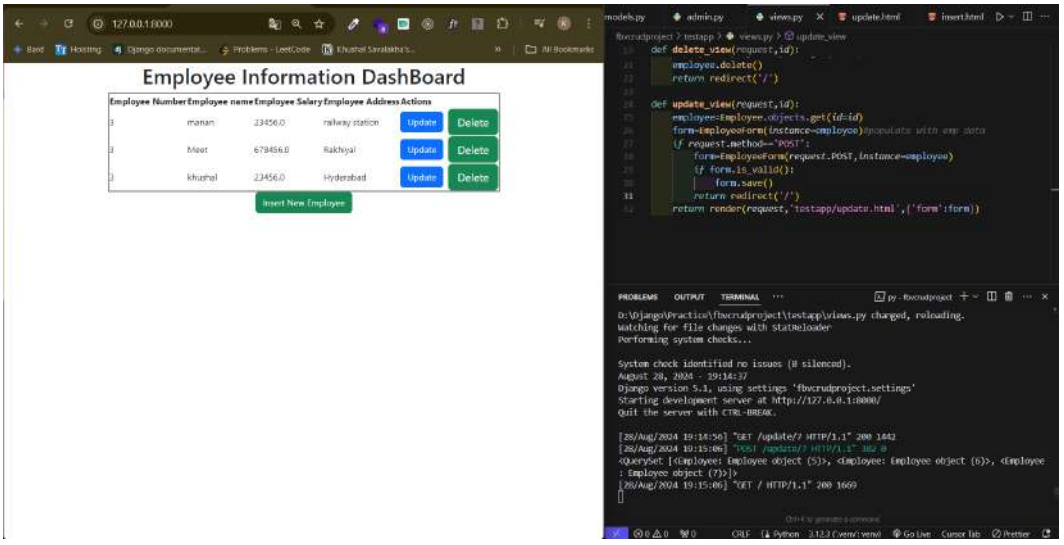
views.py

```
from django.shortcuts import render,redirect
from testapp.models import Employee
```


Khushal Savalakha

```
from testapp.forms import EmployeeForm
# Create your views here.
def retrive_view(request):
    emp_list=Employee.objects.all()
    print(emp_list)
    return render(request,'testapp/index.html',{'emp_l

def insert_view(request):
    form=EmployeeForm()
    if request.method=='POST':
        form=EmployeeForm(request.POST)
        if form.is_valid():
            form.save()
        return redirect('/')
    return render(request,'testapp/insert.html',{'form

def delete_view(request,id):
    employee=Employee.objects.get(id=id)
    employee.delete()
    return redirect('/')

def update_view(request,id):
    employee=Employee.objects.get(id=id)
    form=EmployeeForm(instance=employee)#populate with
    if request.method=='POST':
        form=EmployeeForm(request.POST,instance=employ
        if form.is_valid():
            form.save()
        return redirect('/')
    return render(request,'testapp/update.html',{'form
```


Khushal Savalakha



▼ Class Based Views(CBVs)

```
2. CBVs

1. FBVs are old where as CBVs are new. CBVs are introduced in Django 1.3 Version to implement generic views.

2. CBVs are very very easy to use when compared with FBVs. The most commonly used type of views in real time is CBVs.

3. FBVs are more powerful when compared with CBVs. If you are unable to handle with CBVs then only we have to go for FBVs.

```

```
Note:
1. While defining class based views we have to extend View class.
2. To provide response to GET request, Django will always call get() method. Hence we have to override this method in our view class. similarly other http methods like post(),head(),put(),delete() etc
3. While defining url pattern we have to use as_view() method.
```

urls.py

```
from django.contrib import admin
from django.urls import path
from testapp import views
```

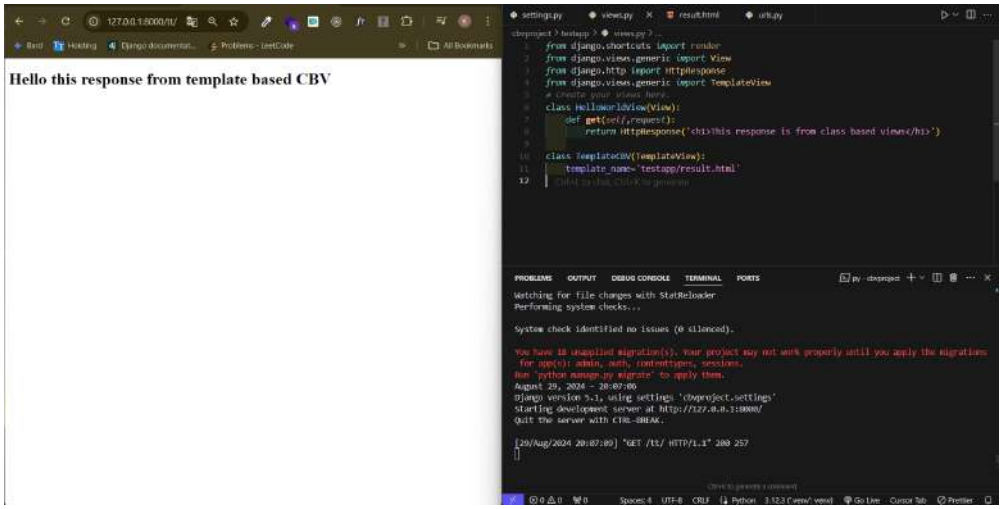

Khushal Savalakha

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('hello/',views.HelloWorldView.as_view()),
    path('tt/',views.TemplateCBV.as_view()),
     path('tt2/',views.TemplateCBV2.as_view()),
]
```

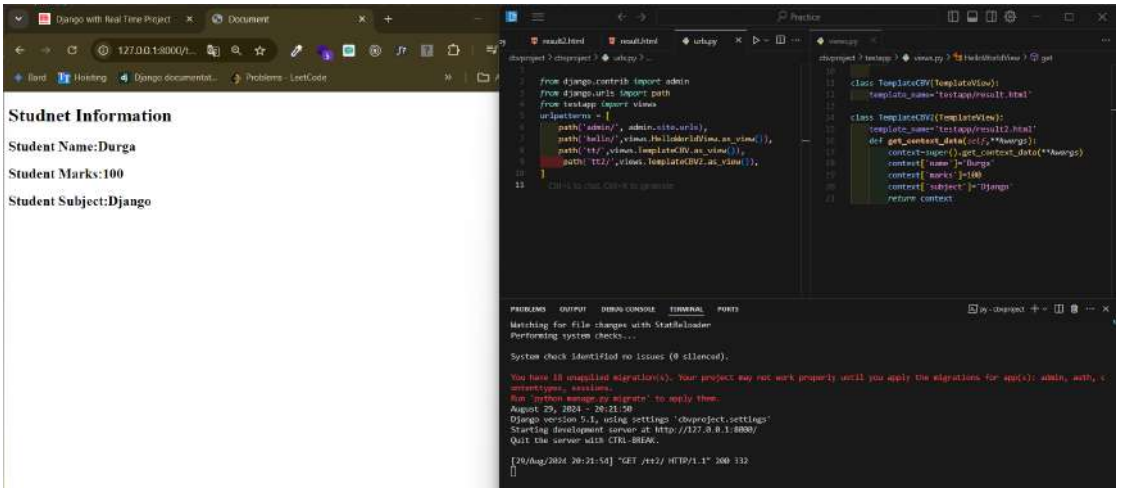
views.py

```
from typing import Any
from django.shortcuts import render
from django.views.generic import View
from django.http import HttpResponse
from django.views.generic import TemplateView
# Create your views here.
class HelloWorldView(View):
    def get(self,request):
        return HttpResponse('<h1>This response is from cl

class TemplateCBV(TemplateView):
    template_name='testapp/result.html'

class TemplateCBV2(TemplateView):
    template_name='testapp/result2.html'
    def get_context_data(self,**kwargs):
        context=super().get_context_data(**kwargs)
        context['name']='Durga'
        context['marks']=100
        context['subject']='Django'
        return context
```


Khushal Savalakha




```
Note: template_name is predefined and TemplateView is also predefined class.
results.html:
------------
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h1>Hello This response from template based CBV</h1>
  </body>
</html>
```



Let's break down how the class-based views (CBVs) work in the provided Django code and how context is sent using the `super()` method.

1. `HelloWorldView` (Basic Class-Based View)

```
class HelloWorldView(View):
    def get(self, request):
```


Khushal Savalakha

```
        return HttpResponse('<h1>This response is from
class based views</h1>')
```

- **What It Does**:
  - This is a simple class-based view that handles GET requests.
  - When a GET request is made to a URL mapped to this view, it responds with an HTML heading `<h1>This response is from class based views</h1>`.
- **How It Works**:
  - The `HelloWorldView` class inherits from Django's `View` class.
  - The `get()` method handles GET requests. The `request` parameter contains all the HTTP request data.
  - The `HttpResponse` object is returned, which sends back the HTML content to the client (browser).

2. `TemplateCBV` (Template-Based Class View)

```
class TemplateCBV(TemplateView):
    template_name='testapp/result.html'
```

- **What It Does**:
  - This class-based view renders a template (`result.html`) when a GET request is made.
- **How It Works**:
  - `TemplateCBV` inherits from Django's `TemplateView`, which is a CBV designed to render templates.
  - The `template_name` attribute specifies the path to the template (`result.html`) that should be rendered when the view is accessed.

3. `TemplateCBV2` (Template-Based Class View with Context)

```
class TemplateCBV2(TemplateView):
    template_name='testapp/result2.html'

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
```


Khushal Savalakha

```
        context['name'] = 'Durga'
        context['marks'] = 100
        context['subject'] = 'Django'
        return context
```

- **What It Does**:
  - This view renders a template (`result2.html`) and also sends additional context data (name, marks, subject) to the template.
- **How It Works**:
  - `TemplateCBV2` also inherits from `TemplateView`, but it overrides the `get_context_data()` method to add more data to the context.
  - **`get_context_data(**kwargs)` Method**:
    - **Purpose**: This method allows you to pass additional data to the template.
    - **How It Works**:
      1. The `super().get_context_data(**kwargs)` call fetches the default context data provided by `TemplateView`.
      2. `super()` refers to the parent class (`TemplateView` in this case). Calling `super().get_context_data(**kwargs)` ensures that any context data prepared by the parent class is not lost.
      3. The `context` dictionary is then updated with additional key-value pairs (`'name'`, `'marks'`, and `'subject'`).
      4. Finally, the updated `context` dictionary is returned, which will be passed to the template.

Summary

- **Class-Based Views (CBVs)** allow you to structure your views using classes rather than functions. This provides a more organized and reusable way of handling different types of requests.
- **`super().get_context_data(**kwargs)`** is used to inherit the default context provided by the parent class (`TemplateView`). By extending this context with custom data, you can send more information to your template without losing any of the context that `TemplateView` might already provide. This approach helps maintain the functionality of the base class while customizing it to your needs.

Khushal Savalakha

💡 here TemplateView and views are default parameters and when we import them from django it will override the default data like get method in TemplateCBV2 am i true?

Not quite. Let's clarify the concepts you're referring to:

1. TemplateView and View Classes:

- **TemplateView** and **View** are not default parameters but rather classes provided by Django's `django.views.generic` module. These classes are designed to handle common tasks in views, like rendering templates or processing different HTTP methods (GET, POST, etc.).
- When you import and use `TemplateView` or `View`, you're using these classes to create your own custom views.

2. Overriding Methods:

- When you create a class that inherits from `TemplateView` or `View`, you have the option to override the methods of these parent classes (like `get()`, `post()`, or `get_context_data()`) to customize their behavior.
- **Example**:
  - In the `TemplateCBV2` example, you override the `get_context_data()` method to add custom data to the context before it gets passed to the template.
  - By calling `super().get_context_data(**kwargs)`, you're not overriding or replacing the default behavior. Instead, you're extending it—you're fetching the context that the parent `TemplateView` class prepares, and then adding your custom data to it.

3. Does Importing Override Default Data?

- Importing `TemplateView` or `View` doesn't override any "default data." Instead, these classes come with predefined methods and behavior that you can use as-is or override to customize.
- **In Context of `TemplateCBV2`**:
  - The `get_context_data()` method in `TemplateView` has a default behavior: it gathers data and returns it as a context dictionary.

Khushal Savalakha

- When you override `get_context_data()` in your subclass (`TemplateCBV2`), you're modifying or extending this behavior, not replacing it completely unless you choose not to call `super()`.

Example Breakdown

Here's how it works in `TemplateCBV2`:

```
class TemplateCBV2(TemplateView):
    template_name = 'testapp/result2.html'

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)  #
Fetches the default context
        context['name'] = 'Durga'  # Adds custom data
        context['marks'] = 100
        context['subject'] = 'Django'
        return context  # Returns the modified context
```

- `super().get_context_data(**kwargs)` calls the `get_context_data()` method from `TemplateView`, ensuring that any context data that `TemplateView` would normally provide is included.
- You then add additional key-value pairs (`'name'`, `'marks'`, `'subject'`) to the `context` dictionary.
- Finally, you return the modified `context` dictionary to be used in the template.

Summary

- **Importing** `TemplateView` or `View` gives you access to these classes and their built-in methods.
- **Overriding methods** like `get()` or `get_context_data()` allows you to customize their behavior.
- Using `super()` within overridden methods ensures that the original method's functionality is retained while adding or modifying it according to your needs.

This way, you extend the functionality rather than override or replace it entirely.

Khushal Savalakha

💡 it means TemplateView or View are like instance of partiulcar class which gives access to class data like get and post methods which we can customize am i true ?

You're on the right track, but let's refine the explanation a bit.

TemplateView and View Classes in Django

- `TemplateView` **and** `View` are **not instances** but **classes** themselves, which you can use to create your own views in Django.
- **When you create a subclass** of `TemplateView` or `View` (like `HelloWorldView` or `TemplateCBV2`), you're creating a new class that inherits all the behavior (methods and attributes) of the parent class.

Access to Methods and Customization

- **Access to Methods**:
  - When you inherit from `TemplateView` or `View`, you gain access to the methods defined in those classes, like `get()` or `post()`.
  - These methods handle specific HTTP requests. For example, the `get()` method handles GET requests, and the `post()` method handles POST requests.
- **Customization**:
  - You can **override** these methods in your subclass to customize their behavior. For example, you can override the `get()` method to define what should happen when a GET request is made to your view.
  - If you don't override a method, the default behavior defined in `TemplateView` or `View` will be used.

Example Breakdown

```
class HelloWorldView(View):
    def get(self, request):
        return HttpResponse('<h1>This response is from
class based views</h1>')
```

- `View` **is a class**: This is the parent class.

Khushal Savalakha

- `HelloWorldView` **is a subclass** of `View`. By inheriting from `View`, `HelloWorldView` automatically gets access to the `get()` method, which you can override.
- **When you override the `get()` method** in `HelloWorldView`, you're customizing what happens when a GET request is made to this view. In this case, it returns a simple HTML response.

In Summary

- `TemplateView` **and** `View` **are classes** provided by Django.
- When you **inherit** from these classes, you can **access** and **customize** their methods like `get()`, `post()`, etc.
- By **overriding** these methods in your subclass, you can define specific behaviors for your views.

So yes, you're correct in saying that `TemplateView` or `View` provide access to class data (methods like `get()` and `post()`), and you can customize them to suit your needs. The key is that they are classes, not instances, and you create subclasses to modify or extend their functionality.

```
To perform CRUD Operations, the following predefined view classes
    ListView --->To select all records(R)
    DetailView-->To get details of a particular record (R)
    CreateView --->To insert a record(C)
    DeleteView -->To delete records(D)
    UpdateView--->To update records(U)

```

```
 1. ListView:
 ------------
 We can use ListView class to list out all records from database table(model).
 It is alternative to :  ModelClassName.objects.all()
```

```
 1. ListView:
 ------------
 We can use ListView class to list out all records from database table(model).
 It is alternative to :  ModelClassName.objects.all()

default template file name: modelname_list.html  eg: book_list.html
default context object name: modelname_list  eg: book_list

models.py:
---------
from django.db import models

# Create your models here.
class Book(models.Model):
    title = models.CharField(max_length=300)
    author = models.CharField(max_length=30)
    pages = models.IntegerField()
    price = models.FloatField()
```

book_list.html

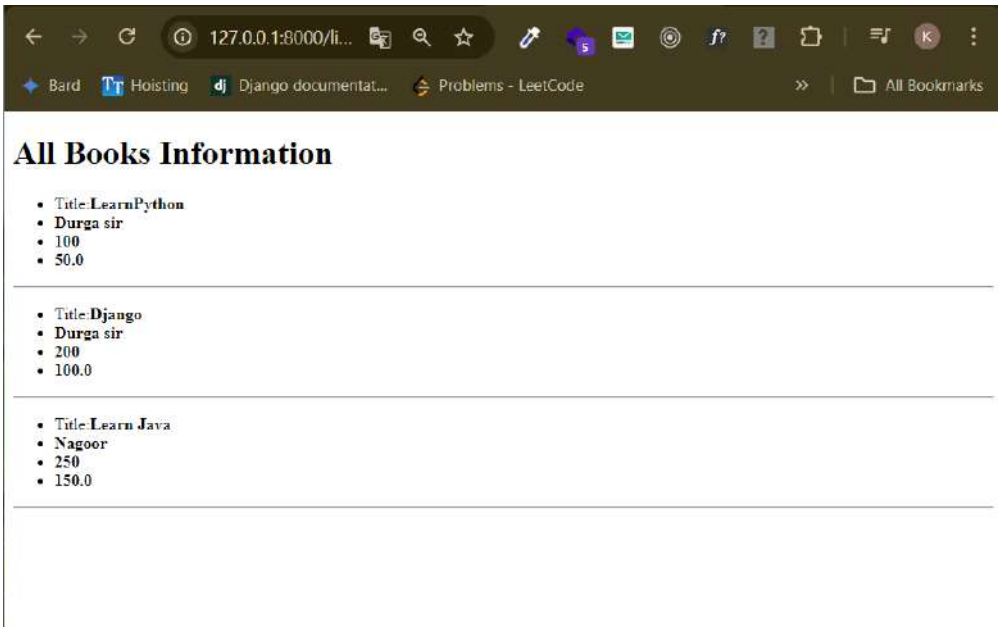
Khushal Savalakha

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, in
    <title>Document</title>
</head>
<body>
    <h1>All Books Information</h1>
    {% for book in book_list %}
    <ul>
        <li>Title:<strong>{{book.title}}</strong></li>
        <li><strong>{{book.author}}</strong></li>
        <li><strong>{{book.pages}}</strong></li>
        <li><strong>{{book.price}}</strong></li>
    </ul>
    <hr>
    {% endfor %}
</body>
</html>
```

views.py

```
from django.shortcuts import render
from testapp.models import Book
from django.views.generic import ListView
# Create your views here.
# def list_view(request):
#     books_list=Book.objects.all()
#     return render(request,'testapp/books.html',{'books_


class BookListView(ListView):
    model=Book
```


Khushal Savalakha



> 💡 explain here how listView works here and how it will show particular html without specifyinh and fetched data explain simply and depthly.

Understanding How `ListView` Works in Django

The code you've provided involves a `ListView` class-based view in Django, which is designed to display a list of objects from a database. Let's break it down step by step to understand how it works and how it renders the HTML template without explicitly specifying it.

1. What is `ListView`?

- `ListView` is a generic class-based view provided by Django, specifically used for displaying a list of objects (usually model instances).
- It automatically handles retrieving the list of objects from the database and passing them to a template.

2. How `ListView` Works

Here's the relevant code:

```
# views.py
from django.views.generic import ListView
from testapp.models import Book
```


Khushal Savalakha

```
class BookListView(ListView):
    model = Book
    # default template file: book_list.html
    # default context object name: book_list
```

- `BookListView` **Class**: This is your custom view that inherits from Django's `ListView`.
  - `model = Book`: This tells Django that the view should retrieve data from the `Book` model.

3. Automatic Template Selection

- **Naming Convention for Templates**:
  - By default, `ListView` will look for a template with a specific name based on the model's name.
  - Since you're using the `Book` model, `ListView` will automatically look for a template named `book_list.html`. This is a convention Django follows:
    - The template name is the lowercase name of the model (`book`) followed by `_list.html`.
  - You don't need to specify the template name unless you want to use a different one. If you did want to specify it, you could use the `template_name` attribute like this:

    ```
    template_name = 'your_custom_template.html'
    ```

4. Fetching Data Automatically

- **Retrieving Data**:
  - `ListView` automatically queries the database for all instances of the model you specified (`Book` in this case).
  - This is equivalent to running `Book.objects.all()` in a function-based view.

5. Passing Data to the Template

- **Default Context Object**:
  - The data retrieved from the database is automatically passed to the template using a context variable.

Khushal Savalakha

- By default, the context variable is named `book_list`, where `book` is derived from the model name.
- In the template (`book_list.html`), you can access this list of `Book` objects using `{{ book_list }}`.

6. Example Workflow

- **1. Request**: When a user visits the URL mapped to `BookListView`, Django triggers this view.
- **2. Query**: The `ListView` automatically runs a query like `Book.objects.all()` to fetch all `Book` objects from the database.
- **3. Template Rendering**:
  - Django looks for a template named `book_list.html` in your `templates` directory.
  - The retrieved `Book` objects are passed to this template as the context variable `book_list`.
- **4. HTML Response**: The template is rendered with the list of books, and the HTML is sent back to the user's browser.

7. Equivalent Function-Based View

For comparison, here's how the same functionality would look in a function-based view:

```
# views.py (Function-Based View)
from django.shortcuts import render
from testapp.models import Book

def list_view(request):
    books_list = Book.objects.all()  # Manually fetch a
ll books
    return render(request, 'testapp/books.html', {'book
s_list': books_list})  # Manually specify the template
and context
```

In this function-based view, you manually:

- Query the database.

Khushal Savalakha

- Specify the template name (`'testapp/books.html'`).
- Pass the context to the template.

With `ListView`, all these steps are handled automatically.

Summary

- `ListView` is a powerful class-based view in Django that automates the process of displaying a list of objects.
- **Automatic Template Selection**: It looks for a template named `<model_name>_list.html` by default (e.g., `book_list.html` for the `Book` model).
- **Automatic Data Retrieval**: It queries the database for all instances of the specified model (`Book.objects.all()`) and passes them to the template using a default context variable (`book_list`).
- This allows you to quickly and easily create views that display lists of data with minimal code.

```
6206 ----------------------------------------
6207 By using template_name variable we have to specify our own template file
6208
6209 class BookListView(ListView):
6210     model=Book
6211     #default template file: book_list.html
6212     template_name = 'testapp/books.html'
6213     #default context object name: book_list
6214
6215 How to configure our own context object:
6216 ----------------------------------------
6217 We have to use : context_object_name variable.
6218
6219 class BookListView(ListView):
6220     model=Book
6221     #default template file: book_list.html
6222     template_name = 'testapp/books.html'
6223     #default context object name: book_list
6224     context_object_name = 'books'
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, in
    <title>Document</title>
</head>
<body>
    <h1>All Books Information</h1>
    {% for book in books %}
    <ul>
```


Khushal Savalakha

```
        <li>Title:<strong>{{book.title}}</strong></li>
        <li><strong>{{book.author}}</strong></li>
        <li><strong>{{book.pages}}</strong></li>
        <li><strong>{{book.price}}</strong></li>
    </ul>
    <hr>
    {% endfor %}
</body>
</html>
```

views.py

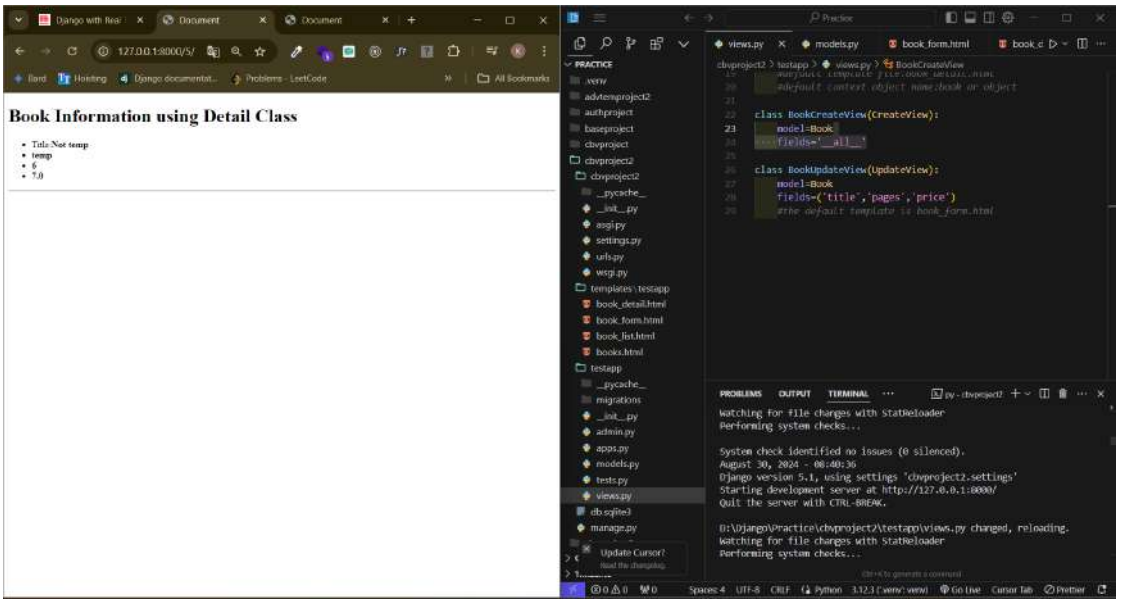
```
from django.shortcuts import render
from testapp.models import Book
from django.views.generic import ListView
# Create your views here.
# def list_view(request):
#     books_list=Book.objects.all()
#     return render(request,'testapp/books.html',{'books_


class BookListView(ListView):
    model=Book
    #default template file:book_list.html
    #default context object name:book_list.html
    # template_name='testapp/books.html'
    context_object_name='books'
```

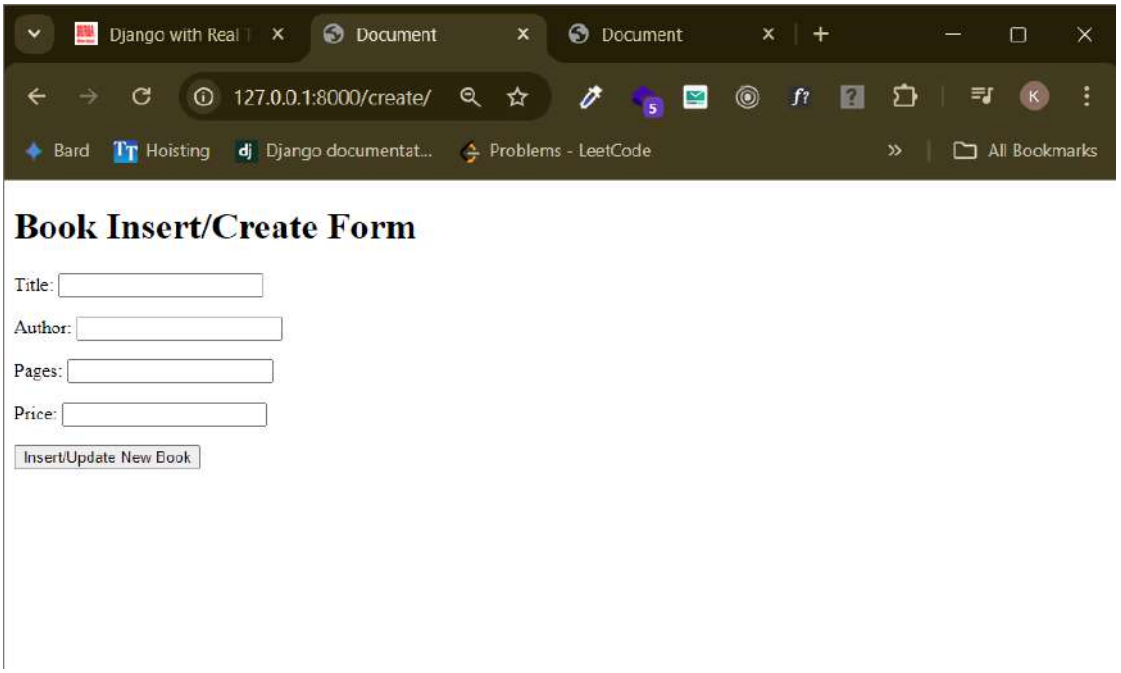
▼ Project with CRUD operation

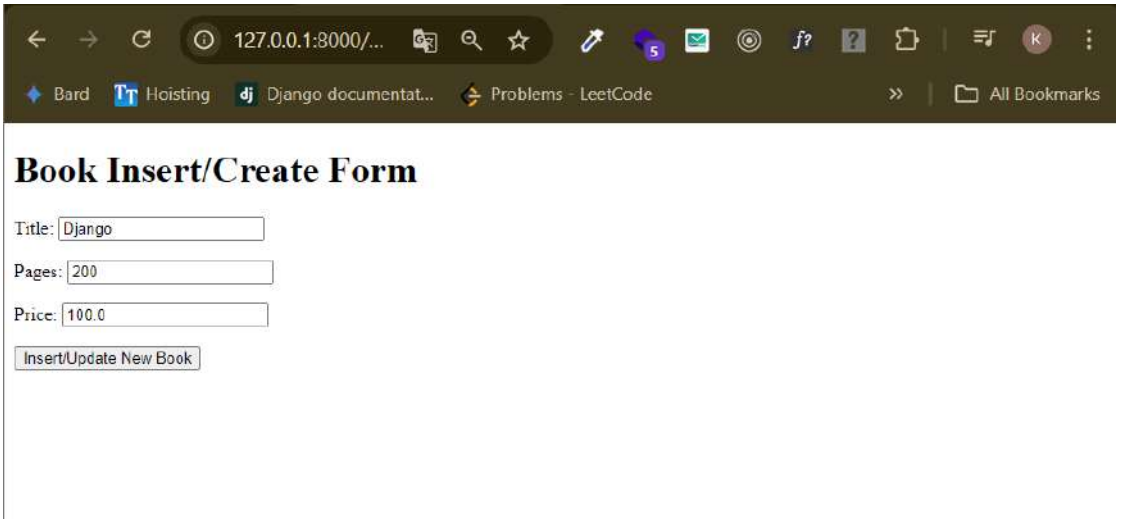
```
ListView -->To list out all records
            default template_name is book_list.html
            default context_object_name  is book_list
DetailView -->To get details of a particular record
             default template_name is book_detail.html
             default context_object_name  is book or object
CreateView -->To insert/create a new record
             default template_name is book_form.html
```


Khushal Savalakha

```
UpdateView:
------------
We can use this class to update existing record.

class BookUpdateView(UpdateView):
    model = Book
    fields=('pages','price')

urls.py:
--------
path('update/<int:pk>', views.BookUpdateView.as_view()),


The default template is: book_form.html
```







here update and insert uses same form to insert and update but data is chnaged for input internally.

```
----------------
We can use DeleteView class to delete records.

127.0.0.1:8000/delete/1


from django.urls import reverse_lazy
class BookDeleteView(DeleteView):
    model = Book
    success_url = reverse_lazy('/list2')

success_url represents the target page which should be displayed after delete operation.
reverse_lazy() function will wait until deleting the record.
```

```
If we are sending the request:
127.0.0.1:8000/delete/1

We are getting error:
TemplateDoesNotExist at /delete/1
testapp/book_confirm_delete.html

If we are trying to delete, DeleteView class will ask for confirmation by showing book_confirm_delete.html. We should provide this page.
```


Khushal Savalakha

```
6466 ----------
6467 from django.urls import reverse_lazy
6468 class BookDeleteView(DeleteView):
6469     model = Book
6470     success_url = reverse_lazy('listbooks')
6471
6472 success_url represents the target page which should be displayed after delete
     operation. Compulsory it should be reverse url.
6473 reverse_lazy() function will wait until deleting the record.
6474
6475
6476 urls.py:
6477 ------
6478 path('delete/<int:pk>', views.BookDeleteView.as_view()),
6479
6480
6481
6482 If we are sending the request:
```

models.py

```
from django.db import models
from django.urls import reverse
# Create your models here.
class Book(models.Model):
    title=models.CharField(max_length=300)
    author=models.CharField(max_length=300)
    pages=models.IntegerField()
    price=models.FloatField()

    def get_absolute_url(self):
        return reverse('detail',kwargs={'pk':self.pk})
```

urls.py

```
from django.contrib import admin
from django.urls import path
from testapp import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('list/',views.BookListView.as_view(),name='list'
    path('<int:pk>/',views.BookDetailView.as_view(),name=
    path('create/',views.BookCreateView.as_view()),
    path('update/<int:pk>//',views.BookUpdateView.as_view
    path('delete/<int:pk>//',views.BookDeleteView.as_view
```


Khushal Savalakha

```
]
```

views.py

```
from django.shortcuts import render
from django.urls import reverse_lazy
from testapp.models import Book
from django.views.generic import ListView,DetailView,Crea
# Create your views here.
# def list_view(request):
#     books_list=Book.objects.all()
#     return render(request,'testapp/books.html',{'books_


class BookListView(ListView):
    model=Book
    #default template file:book_list.html
    #default context object name:book_list.html
    # template_name='testapp/books.html'
    context_object_name='books'

class BookDetailView(DetailView):
    model=Book
    #default template file:book_detail.html
    #default context object name:book or object

class BookCreateView(CreateView):
    model=Book
    fields='__all__'

class BookUpdateView(UpdateView):
    model=Book
    fields=('title','pages','price')
    #the default template is book_form.html

class BookDeleteView(DeleteView):
```


Khushal Savalakha

```
    model=Book
    success_url=reverse_lazy('list')
```

book_form.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, in
    <title>Document</title>
  </head>
  <body>
    <h1>Book Insert/Create Form</h1>
    <form method="post">{{form.as_p}}
    {% csrf_token %}
    <input type="submit" value="Insert/Update New Book"/>
  </body>
</html>
```

book_list.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, in
    <title>Document</title>
</head>
<body>
    <h1>All Books Information</h1>
    {% for book in books %}
    <ul>
        <li><a href="{% url 'update' book.pk %}">UPDATE</
        <li><a href="{% url 'delete' book.pk %}">DELETE</
        <li>Title:<strong>{{book.title}}</strong></li>
        <li><strong>{{book.author}}</strong></li>
        <li><strong>{{book.pages}}</strong></li>
        <li><strong>{{book.price}}</strong></li>
```


Khushal Savalakha

```
    </ul>
    <hr>
    {% endfor %}
</body>
</html>
```

book_confirm_delete.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, in
    <title>Document</title>
  </head>
  <body>
    <h1>Do you want to delete this book</h1>
    <form method="post">
    {% csrf_token %}
    <input type="submit" value="DELETE BOOK"/></form>
  </body>
</html>
```

Let's break down the flow of CRUD operations in Django's class-based views and address your questions about default context object names, template file names, and how they relate to your `Book` model.

CRUD Operations in Class-Based Views

1. **ListView (BookListView)**
   - **Purpose:** Displays a list of all `Book` objects.
   - **Flow:**
     1. When a request is made to the URL mapped to `BookListView`, Django calls the `get_queryset` method (which defaults to `Book.objects.all()`).
     2. The view uses the `ListView`'s default template `book_list.html` (or a custom one if specified) to render the list of books.
     3. The context variable used in the template is `books` (set by `context_object_name='books'`), which is a list of `Book` objects.

Khushal Savalakha

- **Example:**

```
<!-- book_list.html -->
<h1>Book List</h1>
<ul>
    {% for book in books %}
        <li>{{ book.title }} by {{ book.author }}
</li>
    {% endfor %}
</ul>
```

2. **DetailView (BookDetailView)**
   - **Purpose:** Displays details of a single `Book` object.
   - **Flow:**
     1. When a request is made to a URL with a specific `pk` (e.g., `/1/`), Django retrieves the `Book` object with that `pk`.
     2. The view uses the `DetailView`'s default template `book_detail.html` (or a custom one if specified) to render the details.
     3. The context variable used in the template is `book` (or `object`), which is a single `Book` object.
   - **Example:**

```
<!-- book_detail.html -->
<h1>{{ book.title }}</h1>
<p>Author: {{ book.author }}</p>
<p>Pages: {{ book.pages }}</p>
<p>Price: {{ book.price }}</p>
```

3. **CreateView (BookCreateView)**
   - **Purpose:** Provides a form to create a new `Book` object.
   - **Flow:**
     1. When a request is made to the URL mapped to `BookCreateView`, Django displays a form to create a new `Book` object.
     2. Upon form submission, Django saves the new `Book` object and redirects to the URL specified by `success_url`.

Khushal Savalakha

3. The view uses the `CreateView`'s default template `book_form.html` (or a custom one if specified) to render the form.

- **Example:**

```
<!-- book_form.html -->
<h1>Create a new Book</h1>
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Save</button>
</form>
```

4. **UpdateView (BookUpdateView)**

- **Purpose:** Provides a form to update an existing `Book` object.
- **Flow:**

1. When a request is made to a URL with a specific `pk` (e.g., `/update/1/`), Django retrieves the `Book` object with that `pk` and displays a form with its existing data.
2. Upon form submission, Django updates the `Book` object and redirects to the URL specified by `success_url`.
3. The view uses the `UpdateView`'s default template `book_form.html` (or a custom one if specified) to render the form.

- **Example:**

```
<!-- book_form.html -->
<h1>Update Book</h1>
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Save</button>
</form>
```

5. **DeleteView (BookDeleteView)**

- **Purpose:** Provides a confirmation page to delete a `Book` object.
- **Flow:**

Khushal Savalakha

1. When a request is made to a URL with a specific `pk` (e.g., `/delete/1/`), Django retrieves the `Book` object and displays a confirmation page.
2. Upon confirmation, Django deletes the `Book` object and redirects to the URL specified by `success_url`.
3. The view uses the `DeleteView`'s default template `book_confirm_delete.html` (or a custom one if specified) to render the confirmation page.

- **Example:**

```
<!-- book_confirm_delete.html -->
<h1>Are you sure you want to delete "{{ book.titl
e }}"?</h1>
<form method="post">
    {% csrf_token %}
    <button type="submit">Confirm</button>
    <a href="{% url 'list' %}">Cancel</a>
</form>
```

Default Context Object Names

- **ListView:** The default context object name is `book_list`, but you set it to `books` using `context_object_name='books'`.
- **DetailView:** The default context object name is `object`, but it's often renamed to `book` for clarity in templates.
- **CreateView and UpdateView:** The default context object name for the form is `form`, but the template for forms is typically `book_form.html`.
- **DeleteView:** The default context object name is `object`, but it's often renamed to `book` in the template for clarity.

Template File Naming

- Django uses default template names based on the class name. For example:
  - `CreateView` and `UpdateView` use `book_form.html` because the view is related to a `Book` model.
  - `DeleteView` uses `book_confirm_delete.html`.

Khushal Savalakha

The names are derived from the model name in lowercase and are meant to follow a naming convention that helps in organizing templates.

If you want to use custom names, you can set `template_name` in your views. For example, in `BookCreateView`, if you want to use `create_book.html` instead of `book_form.html`, you can specify:

```
class BookCreateView(CreateView):
    model = Book
    fields = '__all__'
    template_name = 'create_book.html'
```

How Context Objects are Generated

- **ListView:** Generates a list of objects (`books`) and passes it to the template.
- **DetailView:** Retrieves a single object (`book`) and passes it to the template.
- **CreateView and UpdateView:** Provides a form (`form`) for creating or updating an object.
- **DeleteView:** Provides a confirmation form and the object (`book`) for deletion.

Understanding these conventions and default behaviors helps in customizing and extending Django's class-based views to suit your application's needs.

Khushal Savalakha

sdfg

from django.db import models
from django.urls import reverse

Create your models here.

class Book(models.Model):
title=models.CharField(max_length=300)
author=models.CharField(max_length=300)
pages=models.IntegerField()
price=models.FloatField()

```
def get_absolute_url(self):
    return reverse('detail',kwargs={'pk':self.p
k})
```

default name of template is created by convert name of class into small letters and attach _list am i true ?because my class name is Book and it is go for search template book_list.html.

You're almost correct in your understanding of how Django determines default template names for class-based views. Let me clarify the process:

Default Template Naming in Django Class-Based Views

1. **ListView**
   - **Default Template Name:** `book_list.html`
   - **Explanation:** For `ListView`, Django generates the default template name by converting the model name to lowercase and appending `_list`. Since your model is `Book`, the default template name becomes `book_list.html`.
2. **DetailView**
   - **Default Template Name:** `book_detail.html`
   - **Explanation:** For `DetailView`, Django generates the default template name by converting the model name to lowercase and appending `_detail`. So for your `Book` model, the default template name would be `book_detail.html`.

Khushal Savalakha

3. **CreateView**
   - **Default Template Name:** `book_form.html`
   - **Explanation:** For `CreateView` and `UpdateView`, Django uses the template name `book_form.html`, which is derived from the model name in lowercase and appends `_form`.
4. **UpdateView**
   - **Default Template Name:** `book_form.html`
   - **Explanation:** The default template for `UpdateView` is the same as for `CreateView`, which is `book_form.html`.
5. **DeleteView**
   - **Default Template Name:** `book_confirm_delete.html`
   - **Explanation:** For `DeleteView`, Django uses the default template name `book_confirm_delete.html`, which is derived by appending `_confirm_delete` to the model name in lowercase.

Example

Given your model `Book`, here's how Django determines the default template names for different views:

- **ListView:** `book_list.html`
- **DetailView:** `book_detail.html`
- **CreateView:** `book_form.html`
- **UpdateView:** `book_form.html`
- **DeleteView:** `book_confirm_delete.html`

Summary

Yes, you are correct that Django generates default template names by converting the class name to lowercase and appending specific suffixes based on the view type:

- `_list` for `ListView`
- `_detail` for `DetailView`
- `_form` for `CreateView` and `UpdateView`
- `_confirm_delete` for `DeleteView`

Khushal Savalakha

You can always customize these template names by setting the `template_name` attribute in your views if you prefer a different naming convention.

```
def get_absolute_url(self):
      return reverse('detail',kwargs={'pk':self.pk})
```

The `get_absolute_url` method in Django models is used to define a canonical URL for an instance of a model. This method returns the URL to a specific detail view of the instance, which is helpful for creating links to a model's detail page.

Explanation of the Method

In your `Book` model, you have:

```
def get_absolute_url(self):
    return reverse('detail', kwargs={'pk': self.pk})
```

Here's a breakdown of how this works:

1. **Purpose:** The method `get_absolute_url` returns the URL for the detail view of the `Book` instance. This is useful for generating URLs that link to the detail page of a particular book.
2. **Usage of `reverse`:**
   - `reverse` **Function:** The `reverse` function is used to reverse-resolve the URL pattern name to an actual URL. It takes the name of the URL pattern and any parameters required by that pattern.
   - **Parameters:**
     - `'detail'`: This is the name of the URL pattern defined in your `urlpatterns` list that points to the detail view for a book.
     - `kwargs={'pk': self.pk}`: This dictionary provides the parameters needed by the URL pattern. In this case, it provides the `pk` (primary key) of the book instance, which is used to generate the URL for that specific book.
3. **Example URL Pattern:**
   - In your `urlpatterns`, you have a pattern like:

     ```
     path('<int:pk>/', views.BookDetailView.as_view(),
     name='detail')
     ```


Khushal Savalakha

- This pattern expects an integer `pk` to be passed in the URL. The `get_absolute_url` method will use this `pk` to generate the correct URL for the book detail view.

4. **How It Works:**
   - If you have a `Book` instance with a primary key (pk) of `5`, calling `book_instance.get_absolute_url()` would generate a URL like `/5/`, which corresponds to the detail view for that book.

Example Usage

If you have a `Book` instance and you want to generate a URL for its detail page, you can use:

```
book_instance = Book.objects.get(pk=5)
url = book_instance.get_absolute_url()
```

This `url` would be something like `/5/`, assuming that the URL pattern name for the detail view is `'detail'` and it expects a `pk`.

In Templates

You can use `get_absolute_url` in Django templates to create links to the detail view of an instance:

```
<a href="{{ book.get_absolute_url }}">View details</a>
```

This will generate a link to the detail page of the `book` instance.

Summary

The `get_absolute_url` method is a convention in Django to provide a canonical URL for an instance of a model. It uses the `reverse` function to generate URLs based on named URL patterns and parameters like `pk`. This makes it easier to create dynamic links to the detail views of model instances.

▼ Media

urls.py

```
from django.contrib import admin
from django.urls import path
```


Khushal Savalakha

```
from django.conf.urls.static import static
from django.conf import settings
from temp import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('movies/',views.home,name='movie')
]

urlpatterns += static(settings.MEDIA_URL, document_root=sett
```

settings.py

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

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

MovieData.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initi
    <title>Movie Data</title>
</head>
<body>
    <h1>Movie Data</h1>

    <form method="GET" action="{% url 'movie' %}">
        <input type="text" name="searchMovie" placeholder="S
        <button type="submit">Search</button>
    </form>
```


Khushal Savalakha

```
    <ul>
        {% if searchTerm %}
            <p>Search Results for "{{ searchTerm }}":</p>
        {% endif %}
        {% for movie in movies %}
        <li>
            <strong>{{ movie.title }}</strong>
            <p>{{ movie.description }}</p>
            <img src="{{ movie.image.url }}" alt="Movie Imag
        </li>
        {% empty %}
        <p>No movies found.</p>
        {% endfor %}
    </ul>
</body>
</html>
```

views.py

```
from django.shortcuts import render
from temp.models import Movie

def home(request):
    searchTerm = request.GET.get('searchMovie', '')
    if searchTerm:
        movies = Movie.objects.filter(title__icontains=searc
    else:
        movies = Movie.objects.all()
    return render(request,'MovieData.html', {'searchTerm': s
```

admin.py

```
from django.contrib import admin
from temp.models import Movie
# Register your models here.
class MovieAdmin(admin.ModelAdmin):
    fields = ['id', 'title', 'description', 'image', 'url']

admin.site.register(Movie)
```


Khushal Savalakha

```
# from django.contrib import admin
# from temp.models import Movie

# class MovieAdmin(admin.ModelAdmin):
#     # Display the 'id' field along with other fields
#     fields = ['id', 'title', 'description', 'image', 'url'

#     # Make 'id' read-only so it is displayed but not edita
#     readonly_fields = ['id']

# admin.site.register(Movie, MovieAdmin)
```

💡 explain here how search of html and GET.get works explain simply and depthly with example.

Sure, I'll explain how searching in HTML forms and handling `GET` parameters with `request.GET.get` works in Django, including an example for clarity.

HTML Form and GET Method

1. **HTML Form Structure**

   In your HTML template, you have a form that allows users to input a search term:

   ```
   <form method="GET" action="{% url 'movie' %}">
       <input type="text" name="searchMovie" placeholder
   ="Search by Movie Title" value="{{ searchTerm }}">
       <button type="submit">Search</button>
   </form>
   ```

   - `method="GET"`: This specifies that when the form is submitted, the data will be sent to the server using the HTTP GET method. This means that form data will be included in the URL of the request.
   - `action="{% url 'movie' %}"`: This specifies the URL where the form data should be sent. The `{% url 'movie' %}` syntax generates the URL for the view named `'movie'`. If you haven't named your URL pattern `'movie'`, make sure to use the correct name.

Khushal Savalakha

Example URL after form submission: `http://127.0.0.1:8000/movies/?searchMovie=Inception`

Here, `searchMovie` is the name of the form input field, and `Inception` is the value entered by the user.

Handling GET Parameters in Django Views

1. **Handling GET Parameters**

   In your Django view, you use `request.GET.get` to retrieve the value of the query parameter:

   ```
   def home(request):
       searchTerm = request.GET.get('searchMovie', '')
       if searchTerm:
           movies = Movie.objects.filter(title__icontains
   =searchTerm)
       else:
           movies = Movie.objects.all()
       return render(request, 'MovieData.html', {'searchT
   erm': searchTerm, 'movies': movies})
   ```

   - **`request.GET`**: This is a dictionary-like object containing all the GET parameters sent with the request. It holds key-value pairs where the keys are parameter names and the values are parameter values.
   - **`request.GET.get('searchMovie', '')`**: This retrieves the value of the `searchMovie` parameter from the GET request. If `searchMovie` is not present in the request, it returns the default value, which is an empty string `''`.

   Example:

   - If the URL is `http://127.0.0.1:8000/movies/?searchMovie=Inception`, `request.GET.get('searchMovie')` will return `'Inception'`.
   - If the URL is `http://127.0.0.1:8000/movies/` (without `searchMovie`), `request.GET.get('searchMovie')` will return `''`.

2. **Filtering the Movie Queryset**

   - **`Movie.objects.filter(title__icontains=searchTerm)`**: If `searchTerm` is provided, this line filters the `Movie` objects where the title contains the search term (case-insensitive).

Khushal Savalakhia

- `Movie.objects.all()`: If no search term is provided, this retrieves all `Movie` objects.

Summary

- **Form Submission**: The HTML form submits data using the GET method, appending it to the URL as query parameters.
- **Fetching Data in Views**: In the Django view, `request.GET.get('parameter_name')` retrieves the value of the specified query parameter from the URL.
- **Query Filtering**: Based on the retrieved search term, you filter the queryset of `Movie` objects to show relevant results.

Example

1. **HTML Form**: User enters "Inception" and submits the form.
2. **URL**: The browser sends a GET request to `http://127.0.0.1:8000/movies/?searchMovie=Inception`.
3. **View Handling**:
   - `request.GET.get('searchMovie')` returns `'Inception'`.
   - The view filters movies where the title contains `'Inception'`.
4. **Template Rendering**: The filtered movies are passed to the template and displayed.

This flow helps users search for movies by title, and Django handles the search query and displays the results accordingly.

💡 <input type="text" name="searchMovie" placeholder="Search by Movie Title" value="{{ searchTerm }}"> here in input we write name="searchMovie" so we writing searchMovie in searchTerm = request.GET.get('searchMovie', '') and here we are writing ' ' means if data is not present then we assign empty string am i true ?

Yes, you are absolutely right!

Here's a breakdown of your understanding:

1. `name="searchMovie"` **in the input field**:

Khushal Savalakha

- This defines the name of the input field as `"searchMovie"`. When the form is submitted, the data entered by the user will be sent with the key `"searchMovie"`.
- For example, if the user enters "Inception" in the search bar, the form submission URL will look something like this: `http://127.0.0.1:8000/movies/?searchMovie=Inception`.

2. `searchTerm = request.GET.get('searchMovie', '')`:
   - In this line, `request.GET.get('searchMovie')` retrieves the value of the `"searchMovie"` parameter from the URL.
   - The second argument `''` is the **default value**. It means that if no `searchMovie` parameter is found in the URL (i.e., if the user didn't enter anything or if the parameter is missing), `searchTerm` will be assigned an empty string `''`.

   So, you're correct: the empty string is assigned as a fallback when there's no data. This prevents errors and ensures your code runs smoothly when no search term is provided.

Example Scenarios:

- **User searches for "Inception"**:

  `searchTerm = request.GET.get('searchMovie', '')` will return `'Inception'`.

- **User submits the form without entering a search term**:

  `searchTerm = request.GET.get('searchMovie', '')` will return `''` (empty string).

This way, your code handles both the presence and absence of search input gracefully.

sdfgh

```
from django.db import models
class Movie(models.Model):
title = models.CharField(max_length=100)
description = models.CharField(max_length=250)
image = models.ImageField(upload_to='movie/images/')
url = models.URLField(blank=True)
```

here why we have to write image.url white fetching image ?
<img src="{{ movie.image.url }}" alt="Movie Image" style="width:100px;height:auto;">
explain me simply and shortly with example .

Khushal Savalakha

In Django, when you use `models.ImageField`, it stores the **path** to the uploaded image, not the actual image itself.

- `movie.image` refers to the **ImageField** object, which contains metadata like the path.
- `movie.image.url` gives the **actual URL** of the image so it can be displayed in the browser.

Example:

- If an image is uploaded to `movie/images/myimage.jpg`, then:
  - `movie.image` refers to the file.
  - `movie.image.url` will return `"/media/movie/images/myimage.jpg"`, which is the URL path to display the image.

Thus, in the HTML, you need to use `movie.image.url` to get the correct image URL for the browser to load the image.

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

STATIC_URL = 'static/'
STATIC_DIR=os.path.join(BASE_DIR,'static')
STATICFILES_DIRS=[
STATIC_DIR,
]

is it proper ?

Yes, the configuration you provided for serving media and static files in Django looks correct. Here's a brief breakdown of each part:

1. Media Files Configuration

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

- `MEDIA_URL`: This is the URL endpoint where media files (e.g., images, videos) will be accessible. For example, if you upload an image, it will be served at `/media/filename.jpg`.

Khushal Savalakha

- `MEDIA_ROOT`: This is the actual directory on your server or file system where uploaded media files are stored. Here, it will store files in the `media/` folder located at the base directory of your project.

2. Static Files Configuration

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

- `STATIC_URL`: This defines the URL for accessing static files (CSS, JS, images used in templates). For example, `/static/style.css`.
- `STATIC_DIR`: This specifies the directory on your local file system where you are storing your static files, which will be in the `static/` folder inside the project.
- `STATICFILES_DIRS`: This tells Django where to look for additional static files (outside the default `app/static/` directories). Here, you're pointing to the `static/` folder in the project root.

Summary:

- `MEDIA_*` settings are for user-uploaded content.
- `STATIC_*` settings are for static assets like CSS, JavaScript, and images used in your templates.

Both configurations look appropriate for handling media and static files in a Django project.

▼ College Practice Projects

▼ Project7

urls.py

```
from django.contrib import admin
from django.urls import path
from testapp import views

urlpatterns = [
    path('admin/', admin.site.urls),
```


Khushal Savalakha

```
    path('products/',views.product_list,name='product_lis
    path('products/<int:product_id>/',views.product_detai
]
```

views.py

```
from django.shortcuts import render,get_object_or_404
from .models import Product

def product_list(request):
    search_term=request.GET.get('search')
    if search_term:
        products=Product.objects.filter(name__icontains=s
    else:
        products=Product.objects.all()
    return render(request,'product_list.html',{'products'

def product_detail(request,product_id):
    product=Product.objects.get(pk=product_id)
    # product=get_object_or_404(Product,pk=product_id)
    return render(request,'product_detail.html',{'product
```

product_list.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, in
    <title>Product List</title>
</head>
<body>
    <h1>Product List</h1>
    <form method="GET" action="{% url 'product_list' %}">
        <input type="text" name="search" placeholder="Sea
    </form>
    {% for product in products %}
    <li>
        <a href="{% url 'product_detail' product.id%}">
```


Khushal Savalakha

```
            {{product.name}}
        </a>
        -{{product.description}}
    </li>
    {% empty %}
        <li>No products found.</li>
    {% endfor %}
</body>
</html>
```

product_detail.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, in
    <title>Document</title>
</head>
<body>
    <h1>Product Detail</h1>
    <h2>{{product.name}}</h2>
    <p>{{product.description}}</p>
    <a href="{% url 'product_list' %}">Back to Product Li
</body>
</html>
```

This Django application consists of a simple product listing and detail view. Let's break it down step by step:

1. Project Overview:

- **Goal:**
  - Display a list of products.
  - Allow users to search for products by name.
  - Show the details of a specific product when clicked.

---

2. Code Breakdown:

Khushal Savalakha

a. `views.py`:

This file contains two view functions: `product_list` and `product_detail`.

```
from django.shortcuts import render, get_object_or_404
from .models import Product
```

- `render`: Used to render templates (HTML files) and pass data to them.
- `get_object_or_404`: Used to retrieve a single object from the database or return a 404 error if it doesn't exist.

`product_list` **view:**

```
def product_list(request):
    search_term = request.GET.get('search')  # Get the
'search' query from the URL
    if search_term:  # If there is a search term entere
d
        products = Product.objects.filter(name__icontai
ns=search_term)  # Filter products by name, case-insens
itive
    else:
        products = Product.objects.all()  # Otherwise,
return all products
    return render(request, 'product_list.html', {'produ
cts': products})  # Pass products to the template
```

- **Purpose:** This view handles displaying all products or filtering products based on a search term.
- **Example:** If the user visits `/products/?search=mobile`, it will search for products with names containing "mobile" (case-insensitive).

`product_detail` **view:**

```
def product_detail(request, product_id):
    product = Product.objects.get(pk=product_id)  # Get
the product by its primary key (id)
    return render(request, 'product_detail.html', {'pro
duct': product})  # Pass the product to the template
```


Khushal Savalakha

- **Purpose:** This view fetches and displays the details of a single product based on its `product_id`.
- **Example:** If the user clicks on a product with ID `1`, they will be taken to `/products/1/` where the details of that specific product will be displayed.

---

b. `urls.py`:

```
from django.urls import path
from testapp import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('products/', views.product_list, name='product
_list'),  # URL for the product list view
    path('products/<int:product_id>/', views.product_de
tail, name='product_detail'),  # URL for product detail
view
]
```

- **Purpose:** This file maps URLs to the corresponding view functions.
- **Example:**
  - `/products/`: Displays the product list or filtered products.
  - `/products/1/`: Displays the details of the product with ID `1`.

---

c. `product_list.html` (Template for the product list):

```
<h1>Product List</h1>
<form method="GET" action="{% url 'product_list' %}">
    <input type="text" name="search" placeholder="Searc
h by product Name:">
</form>
{% for product in products %}
<li>
    <a href="{% url 'product_detail' product.id %}">
<!-- Link to product detail page -->
        {{ product.name }}
    </a>
```


Khushal Savalakha

```
    - {{ product.description }}
</li>
{% empty %}
    <li>No products found.</li>  <!-- If no products ma
tch the search -->
{% endfor %}
```

- **Purpose:**
  - Displays a form for searching products.
  - Lists all the products or the ones filtered by the search term.
- **Example:**
  - If a product named "Mobile" is in the database, it will be listed with a clickable link to its detail page.

---

d. `product_detail.html` (Template for the product detail):

```
<h1>Product Detail</h1>
<h2>{{ product.name }}</h2>
<p>{{ product.description }}</p>
<a href="{% url 'product_list' %}">Back to Product List
</a>
```

- **Purpose:** Displays the details (name and description) of a single product.
- **Example:** If the user clicks on "Mobile" in the product list, they will see its name and description here.

---

3. How the Application Works:

1. **Product List View (`/products/`):**
   - The `product_list` view checks for a search query in the URL (e.g., `?search=mobile`).
   - If a search term is found, it filters products whose names contain the search term (`icontains` for case-insensitive search).
   - If no search term is provided, it displays all products.

Khushal Savalakha

- The results are passed to the `product_list.html` template, which lists the products.
- Each product is linked to its detailed view.

2. **Search Example:**
   - URL: `/products/?search=mobile`
   - The form at the top of the product list allows users to search by product name. For instance, typing "Mobile" in the search box and submitting the form will reload the page with the URL `?search=mobile`.
   - The `product_list` view will filter the products by name and only show products that have "mobile" in their name.
3. **Product Detail View (`/products/<id>/`):**
   - When a user clicks on a product's name, they are taken to the product's detail page, such as `/products/1/`.
   - The `product_detail` view fetches the product by its `id` and passes it to the `product_detail.html` template.
   - The product’s name and description are displayed, and there’s a link to go back to the product list.

---

4. Example in Action:

- **Step 1:** Visit `/products/` — All products are listed with links.
- **Step 2:** Search for "mobile" using the search form — Only products with "mobile" in their name are shown.
- **Step 3:** Click on a product name (e.g., "Mobile Phone") — Redirected to `/products/1/`, where the details of "Mobile Phone" are displayed.

This setup is simple and efficient for browsing and searching products in a basic product catalog system using Django.

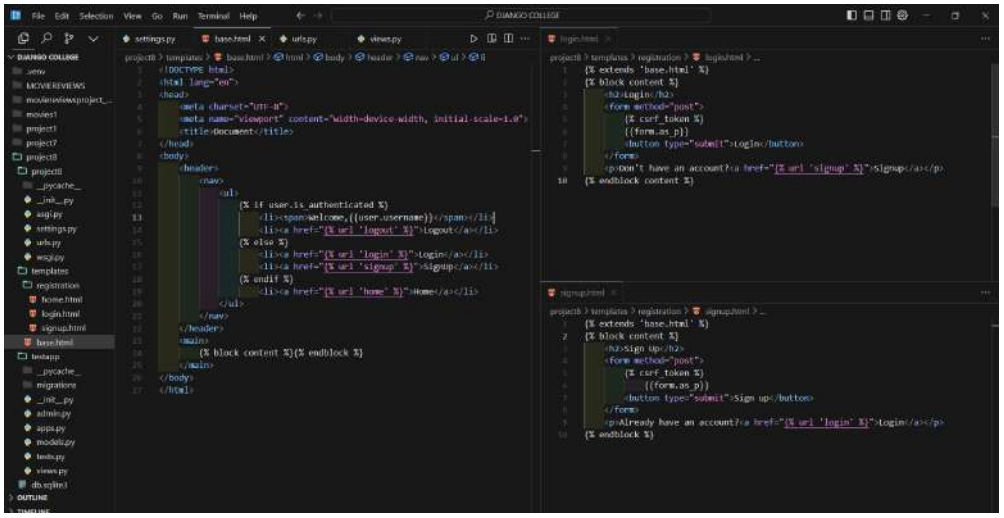
▼ Projects 8

```
from django.shortcuts import render,redirect
from django.contrib.auth.forms import UserCreationForm,Au
from django.contrib.auth import login,logout,authenticate
```


Khushal Savalakha

```
def signup(request):
    if request.method=='POST':
        form=UserCreationForm(request.POST)
        if form.is_valid():
            user=form.save()
            login(request,user)
            return redirect('home')
    else:
        form=UserCreationForm()
    return render(request,'registration/signup.html',{'fo

def user_login(request):
    if request.method=='POST':
        fm=AuthenticationForm(request=request,data=reques
        username=request.POST['username']
        password=request.POST['password']
        user=authenticate(request,username=username,passw
        if user is not None:
            login(request,user)
            return redirect('home')
    else:
        fm=AuthenticationForm()
    return render(request,'registration/login.html',{'for

def user_logout(request):
    logout(request)
    return redirect('login')

def home1(request):
    return render(request,'registration/home.html')
```


Khushal Savalakha



💡 fm=AuthenticationForm(request=request,data=request.POST)
here we are making instance using fm but there is no use in future then why we do that ?
is it like saving data of user which is currently logged in am i true ?

Got it, let's clarify the purpose of `fm = AuthenticationForm(request=request, data=request.POST)` and why it's used in the `user_login` view.

Purpose of `AuthenticationForm`

In Django, the `AuthenticationForm` is a form provided by Django's authentication framework to handle user authentication. It validates user credentials against the database.

`fm = AuthenticationForm(request=request, data=request.POST)`

This line of code does a couple of things:

1. **Form Instantiation**:
   - `AuthenticationForm` is instantiated with `data=request.POST`, which means you are passing the submitted form data (username and password) to the form. This allows the form to process and validate this data.
2. **Validation Preparation**:
   - By creating an instance of `AuthenticationForm` with the data from the request, you prepare the form to perform validation checks. Even if you don't

Khushal Savalakha

explicitly call `fm.is_valid()` in your code (which is unusual), the form instance is essential for:

- **Validation**: Ensuring the submitted credentials are correct.
- **Error Handling**: Collecting and displaying error messages if the login attempt fails.

Why This Step is Important

1. **Data Handling**:
   - **Input Data**: The `AuthenticationForm` instance holds the data provided by the user (username and password). This data is processed to check if it matches any user credentials stored in the database.
2. **Validation**:
   - **Check for Errors**: `AuthenticationForm` will use the provided data to check if the credentials are valid. It can automatically handle common validation tasks, like checking if the username exists and if the password is correct.
3. **Error Messages**:
   - **Form Errors**: If there are issues with the data (e.g., wrong password), the form can generate error messages. These messages can be accessed and displayed to the user to inform them about the problem.

Example Scenario

Consider the following scenario:

```
def user_login(request):
    if request.method == 'POST':
        fm = AuthenticationForm(request=request, data=r
equest.POST)
        if fm.is_valid():  # This checks the validity o
f the data
            username = fm.cleaned_data.get('username')
            password = fm.cleaned_data.get('password')
            user = authenticate(request, username=usern
ame, password=password)
            if user is not None:
                login(request, user)
                return redirect('home')
```


Khushal Savalakha

```
        else:
            # Handle form errors (e.g., invalid credent
ials)
            pass
    else:
        fm = AuthenticationForm()  # Display an empty f
orm for GET requests

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

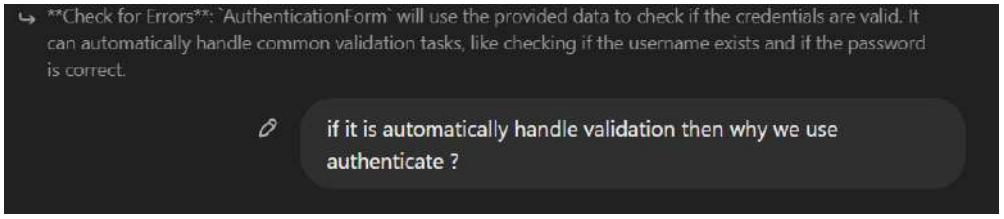
Key Points

- **Form Instance (`fm`)**: It is used to handle the form data and perform validation.
- **Validation Check (`is_valid()`)**: Ensures that the submitted data meets the criteria for successful authentication.
- **Error Handling**: If the data is invalid, error messages are collected and can be displayed to the user.

Not for Storing Logged-In User Data

- **Not for Storing User Data**: The `AuthenticationForm` instance is not used to store data about the currently logged-in user. Its primary role is to process and validate the login credentials.

In summary, the `AuthenticationForm` instance (`fm`) is essential for processing login attempts, validating credentials, and handling any errors related to authentication.



Great question! Let's break down the role of `AuthenticationForm` and `authenticate` in the authentication process:

`AuthenticationForm`

- **Purpose**: `AuthenticationForm` is a Django form designed for user authentication. It is responsible for:

Khushal Savalakha

- **Collecting and Validating Input**: Checking if the provided username and password meet certain criteria (e.g., non-empty fields).
- **Providing Error Messages**: If the credentials are invalid or missing, it will provide error messages.

`authenticate`

- **Purpose**: `authenticate` is a Django function used to verify if the given credentials (username and password) match any user in the database. It performs the actual authentication check:
  - **Authentication**: This function checks the credentials against the stored user data in the database to confirm whether they are valid.

Why Use Both?

1. **Validation vs. Authentication**:
   - **Validation**: `AuthenticationForm` validates the form data (e.g., ensuring fields are filled in). It checks if the form data is in a correct format but does not verify if the credentials match a user in the database.
   - **Authentication**: `authenticate` checks if the validated credentials are correct and match an existing user.
2. **Form Handling and Authentication**:
   - **Form Handling**: `AuthenticationForm` prepares the form data and handles basic validation errors (e.g., empty fields). It's useful for displaying errors in the form.
   - **Authentication**: `authenticate` performs the actual check to determine if the credentials correspond to a valid user.

Example Workflow

Here's how they work together in a login view:

```
from django.contrib.auth import authenticate, login
from django.contrib.auth.forms import AuthenticationFor
m
from django.shortcuts import render, redirect

def user_login(request):
    if request.method == 'POST':
```


Khushal Savalakha

```
        # Create an AuthenticationForm instance with PO
ST data
        fm = AuthenticationForm(request=request, data=r
equest.POST)

        # Check if the form data is valid (e.g., fields
are filled out)
        if fm.is_valid():
            # Extract username and password from the cl
eaned form data
            username = fm.cleaned_data.get('username')
            password = fm.cleaned_data.get('password')

            # Authenticate the user with the provided c
redentials
            user = authenticate(request, username=usern
ame, password=password)

            if user is not None:
                # Log the user in if credentials are co
rrect
                login(request, user)
                return redirect('home')
            else:
                # Handle invalid credentials (e.g., sho
w an error message)
                fm.add_error(None, "Invalid username or
password.")
    else:
        # For GET requests, create an empty form
        fm = AuthenticationForm()

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

Summary

- `AuthenticationForm`: Handles form validation and prepares the form data.
- `authenticate`: Verifies if the credentials match a user in the database.

Khushal Savalakha

Using both allows you to separate concerns: `AuthenticationForm` handles form validation and error handling, while `authenticate` performs the actual authentication check.

▼ 9 marks program

views.py

```
from django.shortcuts import render
from .models import Events

# Create your views here.
def EventList(request):
    list=Events.objects.all().order_by('name')
    if request.method=='GET':
        name=request.GET.get('search')
        if name:
            list=Events.objects.filter(name__icontains=na
    return render(request,'upcomingEvents.html',{'list':l

def EventCategoryList(request):
    category=Events.objects.values('category')
    return render(request,'EventCategory.html',{'category

def home(request):
    return render(request,'mainPage.html')
```

models.py

```
from django.db import models

# Create your models here.
class Events(models.Model):
    name=models.CharField(max_length=40)
```


Khushal Savalakha

```
    Date=models.DateField()
    location=models.CharField(max_length=40)
    category=models.CharField(max_length=40)
```

upcomingEvents.html

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, in
    <title>Upcoming Events</title>
</head>

<body>
    <h1>Upcoming Events</h1>
    <form method="get">
        <input type="text" name="search" placeholder="Sea
        <button type="submit">Search</button>
    </form>
    <table border="1">
        <thead>
            <tr>
                <th>Event Name</th>
                <th>Event Date</th>
                <th>Location</th>
                <th>Category</th>
            </tr>
        </thead>
        <tbody>

                {% for data in list %}
                <tr>
                    <td>{{data.name}}</td>
                    <td>{{data.Date}}</td>
                    <td>{{data.location}}</td>
                    <td>{{data.category}}</td>
                </tr>
```


Khushal Savalakha

```
                    {% empty %}
                        <h1>No Data Found</h1>
                    {% endfor %}

            </tbody>
        </table>
    </body>

    </html>
```

💡 list=Events.objects.filter(name__icontains=name)
here it checks substring or same match am i true ?

Yes, you are correct! The `name__icontains=name` in the query:

```
list = Events.objects.filter(name__icontains=name)
```

is checking for **substring matches**. Specifically:

- `__icontains`: The `icontains` lookup is case-insensitive and checks if the value of the `name` field contains the substring provided in `name`.

For example, if `name="science"`, it will match event names like:

- "Data Science Meetup"
- "Science Fair"
- "Intro to Computer Science"

This means it does not require an exact match; it matches any part of the string in a case-insensitive manner. If you wanted an exact match, you would use `name__iexact=name` for case-insensitive exact matches.

💡 tell me fetch data from user post and get request ?

In Django, you can fetch data from both POST and GET requests in the following ways:

Khushal Savalakha

1. Fetching Data from GET Request

GET requests typically fetch data via URL parameters (also known as query parameters). For example, in the URL `http://example.com/?search=event`, `search` is a query parameter.

To get data from a GET request, you can use the `GET` dictionary from the request object.

Example:

```
def get_data_view(request):
    if request.method == 'GET':
        # Fetch data from GET parameters
        search_query = request.GET.get('search', '')
        # 'search' is the query parameter; '' is the de
fault value if 'search' is not present
    return render(request, 'template.html', {'search_qu
ery': search_query})
```

Here, `request.GET.get('search')` retrieves the value associated with the `search` parameter from the URL.

2. Fetching Data from POST Request

POST requests typically send data via forms or JSON, and the data is not visible in the URL. To fetch the data from a POST request, you can use the `POST` dictionary.

Example:

```
def post_data_view(request):
    if request.method == 'POST':
        # Fetch data from POST form submission
        name = request.POST.get('name', '')
        email = request.POST.get('email', '')
        # 'name' and 'email' are form input names; '' i
s the default value if not found
    return render(request, 'template.html', {'name': na
me, 'email': email})
```

Here, `request.POST.get('name')` retrieves the value associated with the `name` field in the submitted form.

Khushal Savalakha

Combining Both (GET and POST) in One View

You can handle both GET and POST requests in the same view:

```
def handle_get_post(request):
    if request.method == 'POST':
        # Handle POST request data (form submissions)
        name = request.POST.get('name', '')
        email = request.POST.get('email', '')
        print(f'POST data: Name={name}, Email={email}')
    elif request.method == 'GET':
        # Handle GET request data (query parameters)
        search_query = request.GET.get('search', '')
        print(f'GET data: Search Query={search_query}')

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

Key Differences:

- **GET**: The data is sent as URL parameters. You use `request.GET`.
- **POST**: The data is sent as part of the request body (often through a form). You use `request.POST`.

Fetching JSON Data (Common with POST)

If you're sending JSON data, you can parse it like this:

```
import json

def handle_json_post(request):
    if request.method == 'POST':
        body_unicode = request.body.decode('utf-8')
        data = json.loads(body_unicode)
        # Fetch data from JSON body
        name = data.get('name', '')
        email = data.get('email', '')
    return render(request, 'template.html', {'name': na
me, 'email': email})
```

This is commonly used when handling API requests instead of HTML form data.

Khushal Savalakha

In Django, you can fetch data from both POST and GET requests in the following ways:

1. Fetching Data from GET Request

GET requests typically fetch data via URL parameters (also known as query parameters). For example, in the URL `http://example.com/?search=event`, `search` is a query parameter.

To get data from a GET request, you can use the `GET` dictionary from the request object.

Example:

```
def get_data_view(request):
    if request.method == 'GET':
        # Fetch data from GET parameters
        search_query = request.GET.get('search', '')
        # 'search' is the query parameter; '' is the de
fault value if 'search' is not present
    return render(request, 'template.html', {'search_qu
ery': search_query})
```

Here, `request.GET.get('search')` retrieves the value associated with the `search` parameter from the URL.

2. Fetching Data from POST Request

POST requests typically send data via forms or JSON, and the data is not visible in the URL. To fetch the data from a POST request, you can use the `POST` dictionary.

Example:

```
def post_data_view(request):
    if request.method == 'POST':
        # Fetch data from POST form submission
        name = request.POST.get('name', '')
        email = request.POST.get('email', '')
        # 'name' and 'email' are form input names; '' i
s the default value if not found
    return render(request, 'template.html', {'name': na
me, 'email': email})
```


Khushal Savalakha

Here, `request.POST.get('name')` retrieves the value associated with the `name` field in the submitted form.

Combining Both (GET and POST) in One View

You can handle both GET and POST requests in the same view:

```
def handle_get_post(request):
    if request.method == 'POST':
        # Handle POST request data (form submissions)
        name = request.POST.get('name', '')
        email = request.POST.get('email', '')
        print(f'POST data: Name={name}, Email={email}')
    elif request.method == 'GET':
        # Handle GET request data (query parameters)
        search_query = request.GET.get('search', '')
        print(f'GET data: Search Query={search_query}')

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

Key Differences:

- **GET**: The data is sent as URL parameters. You use `request.GET`.
- **POST**: The data is sent as part of the request body (often through a form). You use `request.POST`.

Fetching JSON Data (Common with POST)

If you're sending JSON data, you can parse it like this:

```
import json

def handle_json_post(request):
    if request.method == 'POST':
        body_unicode = request.body.decode('utf-8')
        data = json.loads(body_unicode)
        # Fetch data from JSON body
        name = data.get('name', '')
        email = data.get('email', '')
```


Khushal Savalakha

```
    return render(request, 'template.html', {'name': na
me, 'email': email})
```

This is commonly used when handling API requests instead of HTML form data.

▼ Working With Advanced Model Concepts[page:146-154]

▼ Django MiddleWare[Pending Lec:58- __ page:155-162]

Khushal Savalakha