DJANGO FRAMEWORK

PROGRAMS:

It is a sequence of instruction designed to do a task.

Instruction:

It is a command or an order for a computer to do specific task.

SOFTWARE

Software is a set of instruction or program that are responsible either for Operating a system or performing a specific task.

Software is a collection of programs (interrelated & interdependent) written to achieve a goal (application).

Classifications

It is done based on place where we use the software.

- **System software**
- **Application software**

System software

The software which are responsible for Operating a particular system, such type of software we called it as System software.

System s/w is nothing but set of programs to control and manage the operations of a computer hardware.

System Software required to run the hardware parts of the computer and other application software.

It controls the allocation and usage of different hardware components.

Types: operating system software:
Windows 10, linux, unix, mac os
Utility s/w:
File compression, images, and antivirus programs.
Device drivers:
Keyword devices, mouse touch pen printers.

Ex: - Operating System (OS)

Application software

The software which are responsible for performing specific task, such type of software we called it as application software.

Ex: - Web application, client-server application

1) Stand-alone application

Name itself says that the application will be standing it own.

- **There is no need of internet.**
- **It is independent of OS.**

Ex: - calculator, Ms-office, paint

2) Web application

The application which are access through web browser, such type of application we called it as web application. All the we application informations are present in web servers.

Ex: - Facebook, youtube, skype, pyspider

3) Client-server application

In this type we have to install client software and we will be requesting information through the remote server present in the system and web.

Ex: - Atm,

<u>**Architecture**</u>

It is a design pattern in which application is built. It is used to specify the flow of execution of faces in the application. It is of three types.

- **One-tier Architecture**
- **Two-tier Architecture**
- **N-tier Architecture**

One-tier Architecture

In this type of application, we will be having only one end called as Frontend. By using this frontend only, we'll be doing all the transactions. If we want to change or modify some data then we have to go to respective html page and make the change. We call this architecture as static architecture.

As it is static, it is not preferable in real time.

Two-tier Architecture

In this type of application, we will be having two ends,

- **Frontend**
- **Backend**

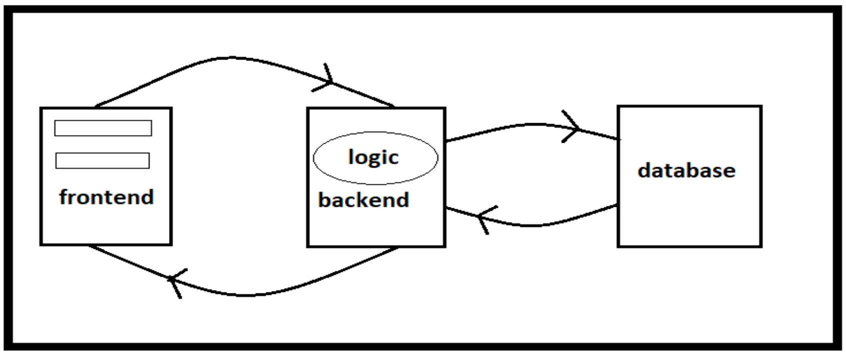
Frontend is used for view of the data to the user, Backend is used for receiving the data from frontend and as well as sending the data to the frontend. We can call this type as medium or semi dynamic architecture.

N-tier Architecture

In this type of application, we will be having three ends,

- **Frontend**
- **Backend**
- **Database**

Frontend is used to view of data to the user, Backend is used for receiving the data from frontend and as well as sending the data to the frontend. Database layer is used to storing the data and as well as sending the data to backend.



(n-tier architecture)

Interaction between Frontend and Backend happens through http (hypertext transfer protocol).

WEB APPLICATION:

- This is the application/software that runs with the help of the browser to do a particular task.
- A web app is a computer program that utilizes web browser and web server/application servers to perform tasks over the internet. (or) A web app is a program that is stored on a remote server and sent over the internet through a browser interface.
- The application will be designed in the below structure

BROWSER:

- A web browser is a software application for accessing information on the World Wide Web.
- Web browser is a client software using which we do the interaction with the servers.
- When a user requests a web page from a particular website, the web browser retrieves the necessary content from a web server and then displays the page on the user's device.

REQUEST:

- This is the set of instruction and information which is send by the browser to the server based on their requirements.

RESPONSE:

- This is the set of instruction and information that we are getting to the server based on the request.

SERVER:

- It is a computer which is installed by the software that is the capability of providing the services based on the users request.

(Or)

- It is a computer with a high configuration and installed with software that makes the computer to provide the service requested.
- Server contains HTML code, CSS Code, JS code, fonts, images, python files, videos, bootstrap, Data base
- The software which will make a normal computer as a server are called Server Software.
  Example:
  - APACHE
  - XAMPP
  - WAMPP
  - TOMCAT
  - ETC…..
- There are set of Operating System that will make a computer as a server.
  Example:
  - Windows server operating system.
  - Linux server operating system
  - Ubuntu server operating system
  - Fedora server operating system
- If ever we are using normal operating system (or) client OS there, we need to installed the server software to give the extra capability of computer to the server but if we installed the OS itself server OS than that particular point of time our computer get the capability of serving the users request.

DATABASE SERVER:

- It is a server containing the database where all required set of data is stored. The db server can be inside an application server (or) it will be outside the application server.
- When the application is very large and contains lot of data in that case we are going to use the database server as a separate and dedicate the server.
- If keep the database server outside the application server than in that case in another computer which we need to be as a db server we are going to install is server s/w that will make our computer to get the capability of serving the server application request (or) the application server db request.
  Example: db server software are
  1. Mysql
  2. Oracle server
  3. Postgre SQL server
  4. Etc….

A server consists of lost of many set of information such as the codes of html, CSS, JavaScript, Bootstrap and Python and data such as images, fonts, videos, PDF etc.. And query language to interact to database (structure query language or non structured query language).

To organize/control/maintain all the set of instruction /information to the server we are going to use the tool called **Framework.**

- That can be framework in development (or) in test automation.

FRAMEWORK:

- Framework is software which is designed to support development of an application.
- It is a set of rules and regulations that helps to organize or structure the files inside a server.
- If we take development framework examples several language will have own language.
  1. Spring and hibernate in JAVA
  2. Lara well in PHP
  3. .Net in C#
  4. Django, Flask, Pyramid in PYTHON.
     Etc…..
- We can development an application even without using framework but it taken a lot of time to get process and get ready or to go for the execution. We don't prefer to create application without using framework.

Web-Framework

Framework which is used for developing web-applications is known as Web-Framework. There are two types of Web-Framework.

- **Micro framework**
- **Major framework**

Micro framework

By using micro frameworks, we can design wed application which are suitable for small-scale industries. By using these only few users can work at a time.

Major framework

By using major frameworks, we can design large-scale application.

Application developed using Python

- Netflix
- Instagram
- Spotify
- Dropbox
- Uber
- Reddit

- Pinterest
- Disqus

DJANGO

- Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design.
- Django is a web development framework that assists in building and maintaining quality web applications.
- Django helps eliminate repetitive tasks making the development process an easy and time saving experience.
- Django makes it easier to build better web apps quickly and with less code.
- It is tells to standardize (or) structure the application related files (python) in an organised manner.
- The django most popular rapid application development where each and every set of instruction need not to completely instead. We can go with the less size of code to do a particular task.
- Rapid development is a process of developing the application quickly and efficiently.
- Some of the large scale applications are Instagram, Udemy, Dropbox, Uber, and Netflix.

History of Django:

- **2003** − Started by Adrian Holovaty and Simon Willison as an internal project at the Lawrence Journal-World newspaper.
- **2005** − Released July 2005 and named it Django, after the jazz guitarist Django Reinhardt.
- **2005** − Mature enough to handle several high-traffic sites.
- **Current** − Django is now an open source project with contributors across the world.

Django comes with the following design philosophies

- **Loosely Coupled:** Django aims to make each element of its stack independent of the others.
- **Less Coding:** Less code so in turn a quick development.
- **Don't Repeat Yourself (DRY):** Everything should be developed only in exactly one place instead of repeating it again and again.
- **Fast Development:** Django's philosophy is to do all it can to facilitate hyper-fast development.

- **Clean Design:** Django strictly maintains a clean design throughout its own code and makes it easy to follow best web-development practices.

Advantages of Django:

Here are few advantages of using Django which can be listed out here −

- **Object-Relational Mapping (ORM) Support** − Django provides a bridge between the data model and the database engine, and supports a large set of database systems including MySQL, Oracle, Postgres, etc. Django also supports NoSQL database through Django-nonrel fork. For now, the only NoSQL databases supported are MongoDB and google app engine.
- **Multilingual Support** − Django supports multilingual websites through its built-in internationalization system. So you can develop your website, which would support multiple languages.
- **Framework Support** − Django has built-in support for Ajax, RSS, Caching and various other frameworks.
- **Administration GUI** − Django provides a nice ready-to-use user interface for administrative activities.
- **Development Environment** − Django comes with a lightweight web server to facilitate end-to-end application development and testing.

Github:

Github is version control tool which is used has a repository to store our projects in centralized memory location so that the people can able to accessing from anywhere any part of the world provide internet is present.

There are two types of github account first one **public account** and second one **private account**.

1) Public account:

It is a type of account which can be created for free of cost and it contain in it can be access by any one.

2) Private account:

It is a type of account which is created by paying some money and the accessibility given only to the given persons.

The pubic account can it not safe to store our project and project related information everybody easy the access of source code.

Configuring github in our system:

To configuration the git in our system we need to download and install the git-scm in our computer.

Steps:

1. Open any browser and go to search bar and type git-scm/download.

2. Display the links of git-scm download and click on it open the git-scm download page.
3. Choose the windows operating system and version number for windows to click on download button.
4. After that downloaded exe file and double click on downloaded file and click on install and given the permissions.
5. Once after installation restart the command prompt and type git and press enter.
6. It you see some information about git which get display on the cmd than its means that git-scm is connected configuration successfully.
7. Login to github account in browser ad display the home page of github your account and click on "+" symbol in right side corner.
8. Select the new repository and specify the repository name and select the public access and click on create repository button.
9. After that it goes to give you some set of commands that you need to use it. When upload your project to git repository.
10. Given the commands:

    The below commands is used to load the project to the newly created repository.

    ```
    >> git init
    >> git add path of the project file (or) .
    >> git commit -m "first commit"
    >> git branch –M main
    >> git remote add origin (giturl)
    >> git push –u origin main
    ```

    The below commands is used to load the project to the existing repository.

    ```
    >> git remote add origin (giturl)
    >> git branch –M main
    >> git push –u origin main
    ```

Git init:

It is used to create the local git container in the present directory.

Git add path of the project:

It is used to add the project files to the local git repository by converting into its own git format.

Git commit –m "first commit":

It is the command is used to save the changes "first commit" we are comment to the passing to name of the commit.

Git branch –M main:

It is used create a main thread of the git hub.

Git remote add origin giturl/repository:

It is used to establish a connection between the local git repository to the git repository.

Git push –u origin main:

It is used to push the code to the repository that is present.

Command prompt commands:

commands that is used in widows for the operation in command prompt
1. mkdir: it is command which is used to create a directory or folder
syntax:

```
mkdir folder_name
```

2. cd : it is command which is used to change the working location from the current directory to the specified path
Syntax:

```
cd path_of_directory
cd.. (will get you out from current directory and keeps control in its parent directory)
cd../..(will get you out from current directory and keeps control in its grand parent directory)
```

3. In unix/windows/linux/mac . means current directory
.. means parent directory

4: rmdir it is a command used to remove directory
syntax:

rmdir foldername	folder should be empty
rmdir /s /q folderpath	folder will be deleted forcefully

5. del: used to delete files

```
del filepath_with_extn
```

6: jumping from drive to drive
synatx: driveletter:
ex:

```
c:\users\venugopal\desktop\akshay_django>D:
d:\>
```

Django frame-work:

1. A web application framework is a toolkit of all components needed for application development.

2. Django is a high-level open source python based frame work which is responsible for rapid development of web application with clean programmatic design with more security.
3. Django follows MVC and MVT DESIGN PATTERNS.

Design pattern:

Design patterns are used describe how the data and code segregation should be done There are two types of Design patterns:

- MVC------->MODEL VIEW CONTROL
- MVT------->MODEL VIEW TEMPLATE

MVC:

It is design pattern which responsible for designing an application with the separation of data happens in the following format

MODEL:

Model is responsible for writing all the operations related to the DATABASE.

VIEW:

This is file where we write the logic by using python program this is used for displaying the content to the user

CONTROL:

Django frame work itself take care of this controlling part

MVT:

While designing an application we have to concentrate on three important points

1. Layout (html, css, js)
2. Logic (python programming language)
3. Data (database operations)

MVT is similar to MVC design pattern but in case of MVT design pattern we will be responding an html layout as a response to the request sent by the client-user

Creation of development environment:

- An environment is a setup that we do for the execution of project successfully. Environment is classified into two categories:
  1. The real time environment (or) actual environment
  2. The virtual environment

  It is the environment with the set it up inside our computer permanently to that whenever are running a project inside our computer think should run in an efficient way every time we use the project in different computer. We can able to create this setup every time because is consume lot of time and effect to set it up everything.

  To avoid this we create to portable environment where everything will install along with the project to that every time. Whenever we wanted to and a project diff computer as the environment is portable. We just need to carry it and activate it. This is going to reduce lot of time and effect.

Virtual environment is a concept of splitting one actual environment into multiple numbers of virtual environments splitting one pc into n virtual pc is a task of virtual environment.

To create a virtual environment are two possible.

1. By utilizing python distribution
2. By utilizing library called virtual environment(env)

Installation of virtual environment:

1 step: first install python software required version and set by click on add python path. Once after installation go to command prompt type PIP (packaging installer python) and press enter.

Note: if pip is not their do like below steps.

2 step: if you get error message taken that python is a neither an internal or external command than go to path where python is install and copy the path.

C: \user\username\AppData\local\programs\python\python version.

3 step: it is the place where ever python interpreter is present than

4 Step: press the start button type ENV and click on edit system environment variable

- ➢ Follow by click on "Environment variable" button inside.
- ➢ Find the path both in USER VARIABLE and SYSTEM VARIABLE environment.
- ➢ Double click on path click on NEW button than paste the path on interpreter is pres
- ➢ One more time click on NEW button paste the path with suffix as scripts file.
- ➢ And click on ok upon ENV window is close.
- ➢ And restart your pc and go to command prompt
- ➢ Type PIP and click enter
- ➢ Check so that the environment setup is ready or not.

Process 1:

Command to create Virtual Environment:

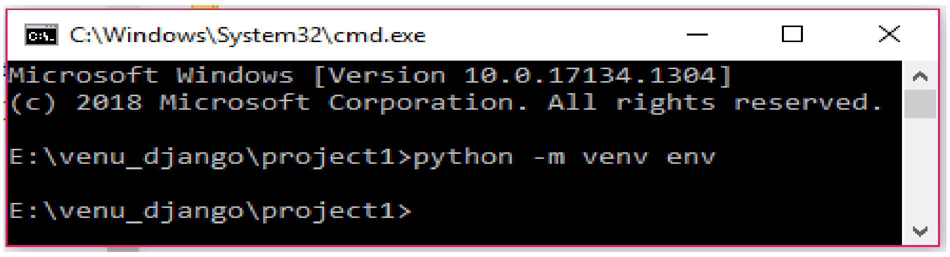
- First select the folder, in that folder path will copy and after that go to search bar in that folder type CMD
- Than open the command prompt after that type the command and press entre

Syntax: python<space>**–m**<space>**venv**<space>**env (or) any_env_name**

Example: cd E:\venu_django\project1\python –m venv env

env means user given environment name

venv means virtual environment



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.17134.1304]
(c) 2018 Microsoft Corporation. All rights reserved.

E:\venu_django\project1>python -m venv env

E:\venu_django\project1>
```

Fig: Environment is created in that folder than

- Immediately after creating virtual environment on brand new system will be created.
- Brand new system which is created will be having the following files:

1. Lib

2. Scripts

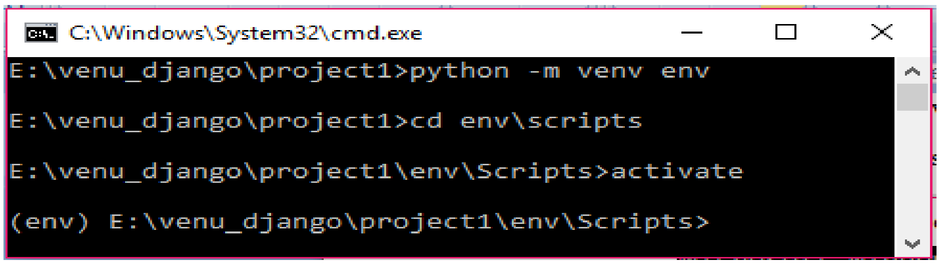
3. Environment configuration

After environment is created select the scripts folder than copy that path

- Open the command prompt after that paste the scripts path to activate the environment type **<path of environment>\env_name\scripts\activate** and press enter.

Example: cd E:\venu_django\env\scripts\activate

- To come to the environment we need to deactivate a environment for that just type **"deactivate"** press enter.



```
C:\Windows\System32\cmd.exe
E:\venu_django\project1>python -m venv env

E:\venu_django\project1>cd env\scripts

E:\venu_django\project1\env\Scripts>activate

(env) E:\venu_django\project1\env\Scripts>
```

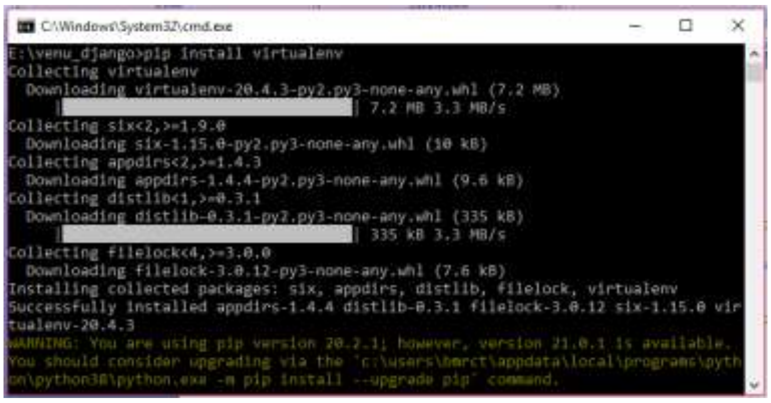
Process 2: Create a third Party Virtual Environment.

1. Step: make sure your computer is connect into the internet (only by installation packages).

- Installation of any library's will be done by using "PIP" with the
  "pip install name_of_library"

2. Step: In the second process the virtual environment is created by using the library "virtualenv" and we need to install the library first by using pip.

"pip install virtualenv" press enter



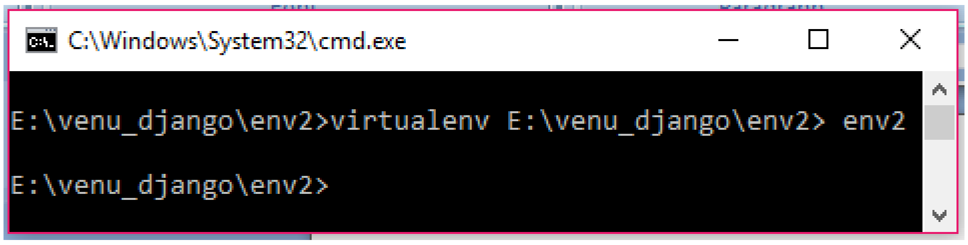
```
E:\venu_django>pip install virtualenv
Collecting virtualenv
  Downloading virtualenv-20.4.3-py2.py3-none-any.whl (7.2 MB)
     |████████████████████████████████| 7.2 MB 3.3 MB/s
Collecting six<2,>=1.9.0
  Downloading six-1.15.0-py2.py3-none-any.whl (10 kB)
Collecting appdirs<2,>=1.4.3
  Downloading appdirs-1.4.4-py2.py3-none-any.whl (9.6 kB)
Collecting distlib<1,>=0.3.1
  Downloading distlib-0.3.1-py2.py3-none-any.whl (335 kB)
     |████████████████████████████████| 335 kB 3.3 MB/s
Collecting filelock<4,>=3.0.0
  Downloading filelock-3.0.12-py3-none-any.whl (7.6 kB)
Installing collected packages: six, appdirs, distlib, filelock, virtualenv
Successfully installed appdirs-1.4.4 distlib-0.3.1 filelock-3.0.12 six-1.15.0 vir
tualenv-20.4.3
WARNING: You are using pip version 20.2.1; however, version 21.0.1 is available.
You should consider upgrading via the 'c:\users\bmrct\appdata\local\programs\pyth
on\python38\python.exe -m pip install --upgrade pip' command.
```

3. Step: once after the install successfully of virtualenv and create a virtual environment using below command.

virtualenv <space><path of environment>\name_of_env press enter

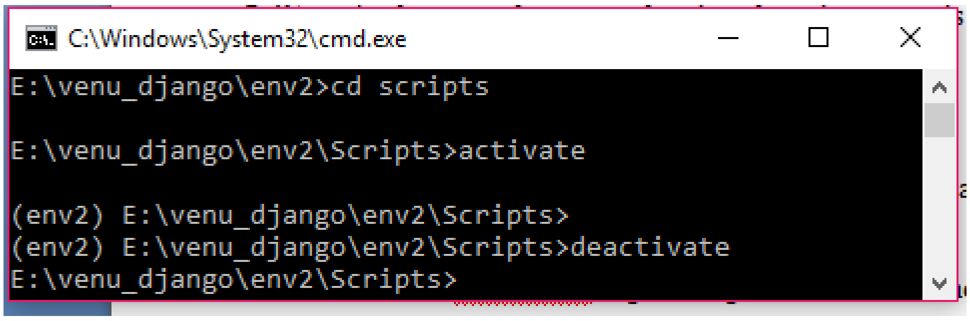
(Or)

virtualenv name_of_env   press enter



```
E:\venu_django\env2>virtualenv E:\venu_django\env2> env2

E:\venu_django\env2>
```

4. Step: Activate and Deactivate environments are same as process1.



```
E:\venu_django\env2>cd scripts

E:\venu_django\env2\Scripts>activate

(env2) E:\venu_django\env2\Scripts>
(env2) E:\venu_django\env2\Scripts>deactivate
E:\venu_django\env2\Scripts>
```

Once activating the environment the changes that we do will have its impact only for that environment.

Note: Once after install/creating the environment we don’t need to have python in the global system because already we have a dedicated interpreter pip everything in each environment. (Note: it is not a best practice).

Installation of any third party library will be done by using: PIP

“pip install (3 diff type) “

There are three ways to install the libraries:

1) Installing the library of current version done by using the command.

"pip install library_name".

2) Installing the library of specific version done by using the command.

"pip install library_name == version_number"

3) Installing from a librarys that is specified in a file will be done by a install command.

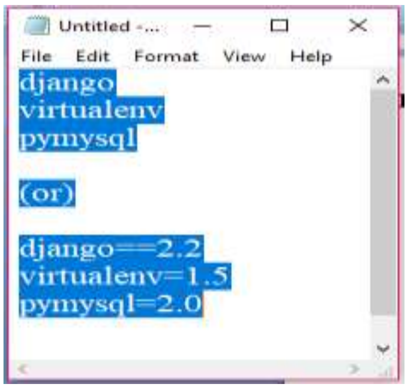
"pip install –r <path of file>"

Note: name of the requirement file with path

Usually when we talk about large set of projects there will be "n" no of library that might be us need but we cannot install all the library's one after another because it is a time consuming process.

The requirement file and that file will be creating and write the only library name that is either library_name or library_name==version.

Requirement.txt



```
django
virtualenv
pymysql

(or)

django==2.2
virtualenv=1.5
pymysql=2.0
```

Installation of DJANGO:

The command of install the django

"pip install django" (current version)

"pip install django==2.2" (specific version)

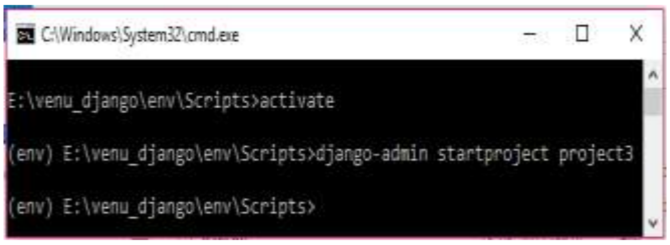
- Once after successfully installation of Django we will get a file called "django-admin".
- To check that type command is **"django-admin"** press enter
- Display the all the django-admin files.
- If you don't get django-admin files not a internal or external command error than your installation is successfully.

How to Create a Django Project:

- To create a Django-project we use a command

"django-admin startproject project_name"

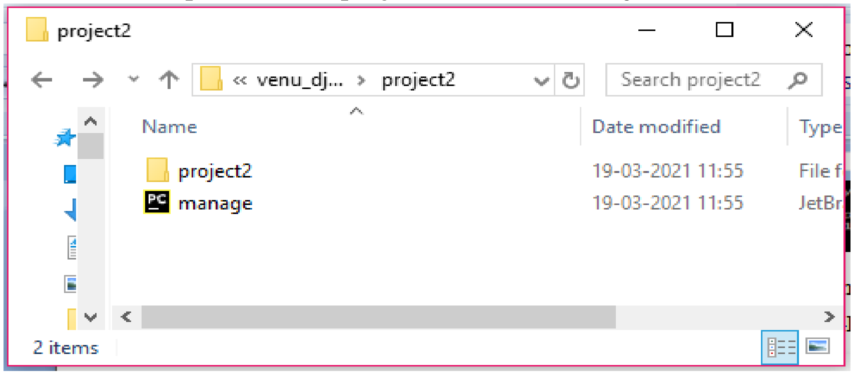
(Create a project in current directory)



```
E:\venu_django\env\Scripts>activate

(env) E:\venu_django\env\Scripts>django-admin startproject project3

(env) E:\venu_django\env\Scripts>
```

- User command Django-admin startproject project_name will create a project in the specified path with the given project name.
- Once after the creation of project to check whether successfully created (or) not
- In that project folder there are automatically created the project files.

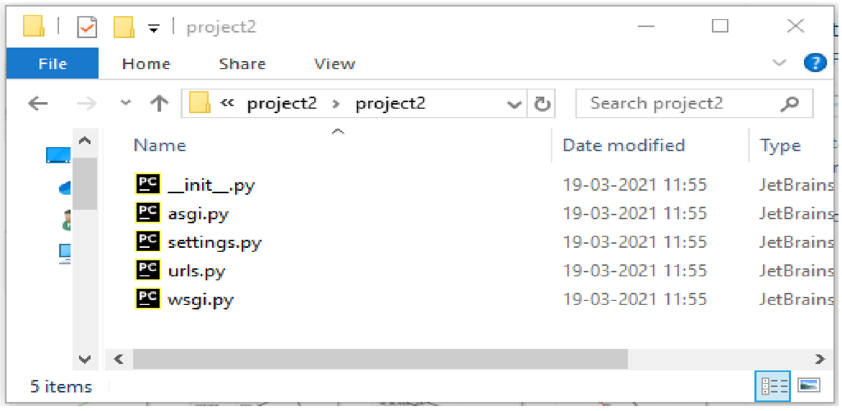
Example 1: After project2 is successfully created



Name	Date modified	Type
project2	19-03-2021 11:55	File f
manage	19-03-2021 11:55	JetBr

Fig: outer project2 directory.

Example 1.1: Project2 inside the files.



Name	Date modified	Type
__init__.py	19-03-2021 11:55	JetBrains
asgi.py	19-03-2021 11:55	JetBrains
settings.py	19-03-2021 11:55	JetBrains
urls.py	19-03-2021 11:55	JetBrains
wsgi.py	19-03-2021 11:55	JetBrains

Fig: outer project2 inside an inner project2 directory.

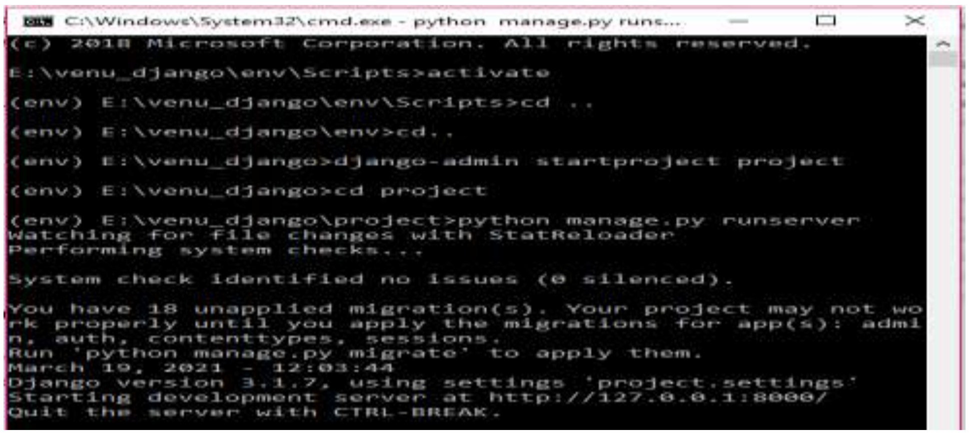
- Where run server using below command

  Get into outer project directory using “cd” command

  “cd project_name”

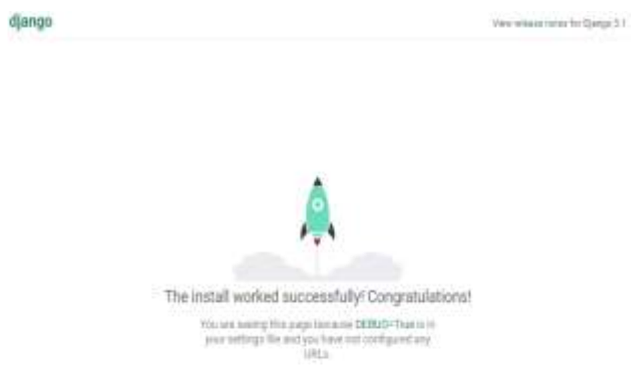
- To run the server use command

  “python manager.py runserver”



```
C:\Windows\System32\cmd.exe - python  manage.py runs...
(c) 2018 Microsoft Corporation. All rights reserved.

E:\venu_django\env\Scripts>activate

(env) E:\venu_django\env\Scripts>cd ..

(env) E:\venu_django\env>cd..

(env) E:\venu_django>django-admin startproject project

(env) E:\venu_django>cd project

(env) E:\venu_django\project>python manage.py runserver
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 wo
rk properly until you apply the migrations for app(s): admi
n, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
March 19, 2021 - 12:03:44
Django version 3.1.7, using settings 'project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

- Copy the URL /IP address that is given and paste it inside the browser and press enter.
- If you get a congratulation page than installation is happen successfully.



django

View release notes for Django 3.1

The install worked successfully! Congratulations!

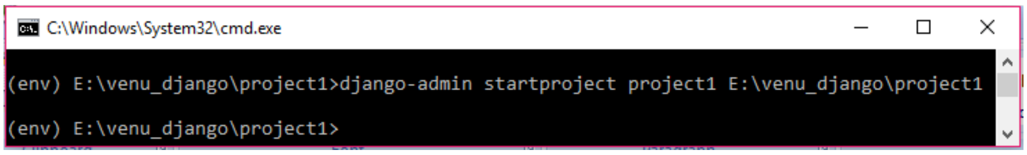
You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.

<u>Three different ways to create a project:</u>

1) **Project in the specifed directory**
2) **Project in the current directory**
3) **Creating a project with the container**

1) Project in the specifed directory:

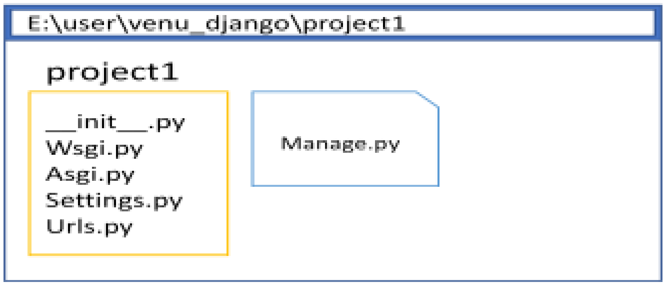
Django-admin startproject project path



```
C:\Windows\System32\cmd.exe

(env) E:\venu_django\project1>django-admin startproject project1 E:\venu_django\project1

(env) E:\venu_django\project1>
```

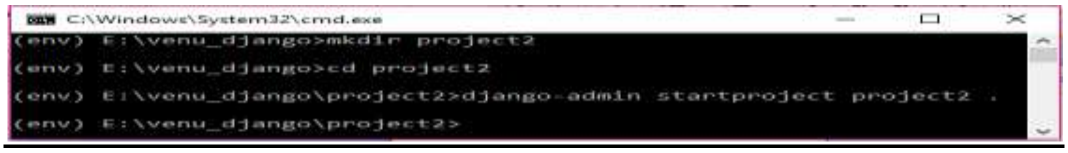
Example: In the specified path the project files will be created.

(env) E:\venu_django\project1>django-admin startproject project1 E:\venu_django\project1



2) Project in the current directory:

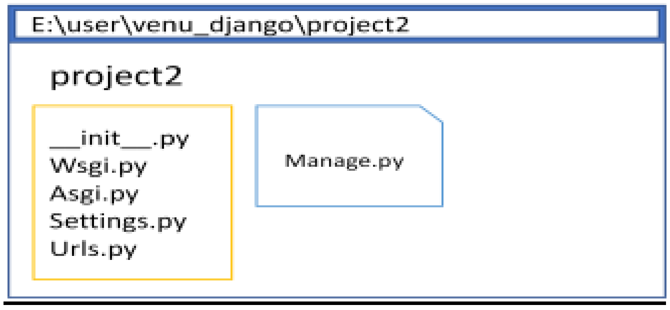
Django-admin startproject projectname .



```
C:\Windows\System32\cmd.exe
(env) E:\venu_django>mkdir project2
(env) E:\venu_django>cd project2
(env) E:\venu_django\project2>django-admin startproject project2 .
(env) E:\venu_django\project2>
```

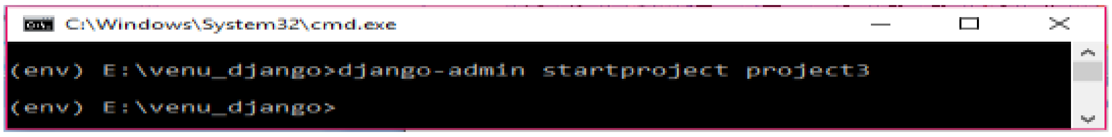
Example: In the current directory to which the "**cmd prompt"** is pointing to project file will be created in "django-admin startproject project2 ."

(env) E:\venu_django\project2>django-admin startproject project2 .



3) Creating a project with the container.

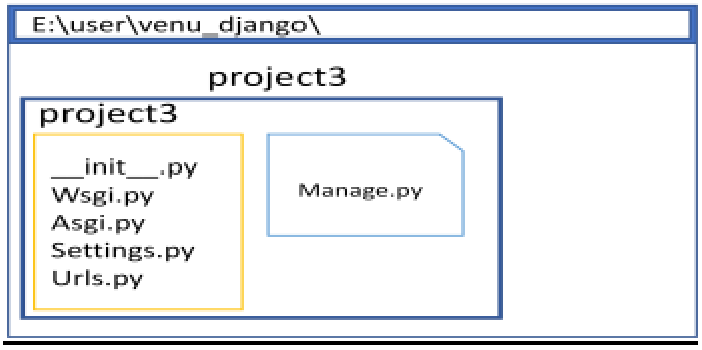
Django-admin startproject projectname



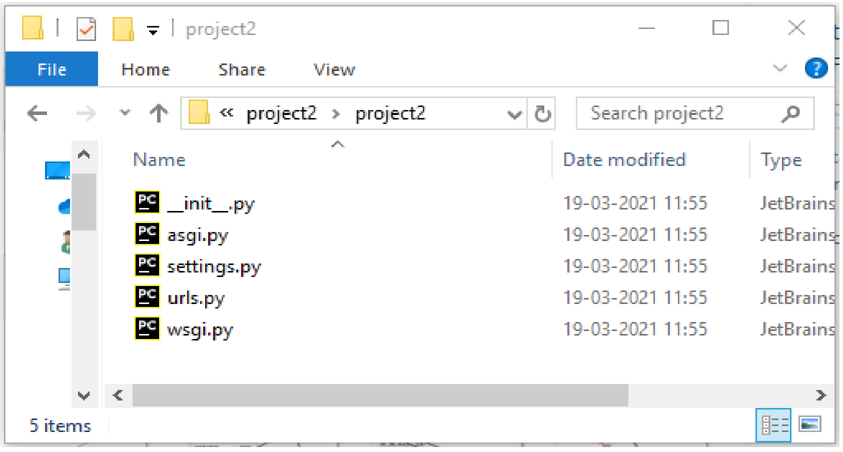
```
C:\Windows\System32\cmd.exe
(env) E:\venu_django>django-admin startproject project3
(env) E:\venu_django>
```

Example: In this project created by the specified directory but in this project created by outer project as well as inner project with manage.py

(env) E:\venu_django>django-admin startproject project3



Describe the project files inside the project.



Name	Date modified	Type
__init__.py	19-03-2021 11:55	JetBrains
asgi.py	19-03-2021 11:55	JetBrains
settings.py	19-03-2021 11:55	JetBrains
urls.py	19-03-2021 11:55	JetBrains
wsgi.py	19-03-2021 11:55	JetBrains

__init__.py:

Initialization file in which contains the statement that is necessary for the project pre-setup. It is blank python script. Because of this special file name, django treated this folder as python package

Setting.py:

This file is responsible for the integration of the project related settings and configurations such as DB setting, media settings, IP settings, other settings and installed app settings.

Urls.py:

This is the file which is responsible for url navigations. Here we have to store all our url-patterns of our project. For every view (webpage), we have to define separate url-pattern. End user can use url-patterns to access our web pages.

Wsgi.py: [Web Server Gateway Interface]

Wsgi is like watchman/gatekeeper. It will be checked & authenticated the browser requests come to the file called wsgi.py in the server. There server response will also passing through the wsgi only. Wsgi has the capability to handle only synchronous request [browser request]. We can use this file while deploying / hosting our application in production on online server

Asgi.py: [Asynchronous Server Gateway Interface]

Similar to wsgi.py but it will handle only asynchronous request that comes from sockets (or) web scraping algorithms. Asgi will be present only in Django 3.X above versions.

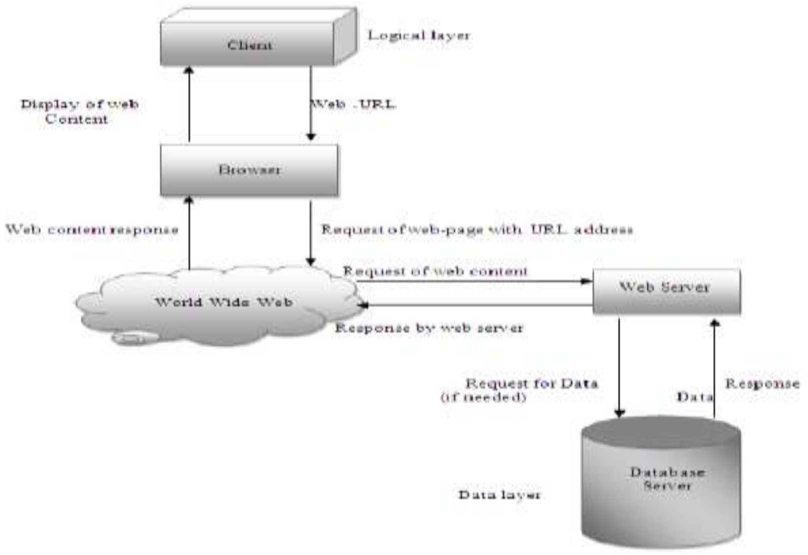
Manage.py:

Maintains & manage all the operations of a project or in a project. It is a command line utility to interact with django project in various ways like to run development server, run tests, create migrations etc

Web Scraping:

It is phenomenal of extracting the information from a web application without using browser.

Web Application working flow in internet using url's:

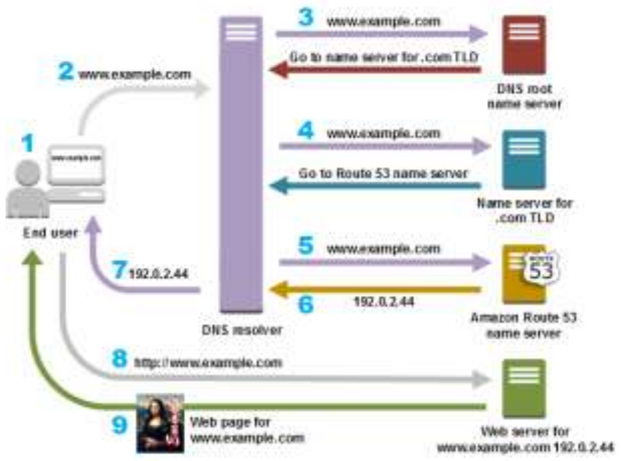


Whenever we need to enter into the server we should know some set of basic things and there are:

1) Request should be readable (or) encrypted format
2) What is the address of the server
3) Entry point (or) port number
4) Where to navigate and what to display

To know about all this information the first format thing is how we do carry this information the answer is using "URLs"

"URLs" expanded as Uniform Resource Locator that helps as to locate the required set of resource from the specific area or specific location.



The url is return as: http://www.example.com

http://www.facebook.com/venugopal

http://www.facebook.com/venugopal/profile

http://www.pyspider.com/

The format of url will be protocol://baseurl/primarysuffix/secondary suffix/……..

Protocol:

It is a set of rules and regulations using which request will be sent to the server.

There are two types of protocols:

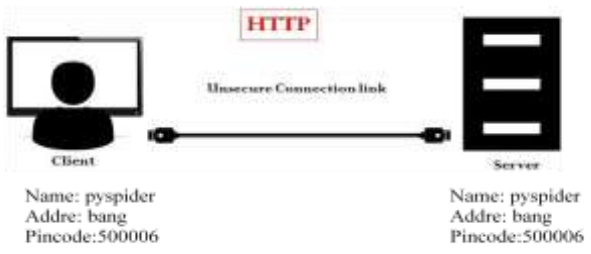
1) http (hyper text transfer protocol)
2) https (hyper text transfers protocol security)

HTTP:

HTTP (Hypertext Transfer Protocol) is the base of the data communication for the web this is how the internet works when it comes to delivering the web pages. It is TCP/IP based protocol and things like text, audio, videos, images can be transmitted through it.

HTTP works on **request** and **response** cycle where the client requests a web page. Suppose, if you browse to google.com, you are requesting a web page from the server, and the server will deliver you response.

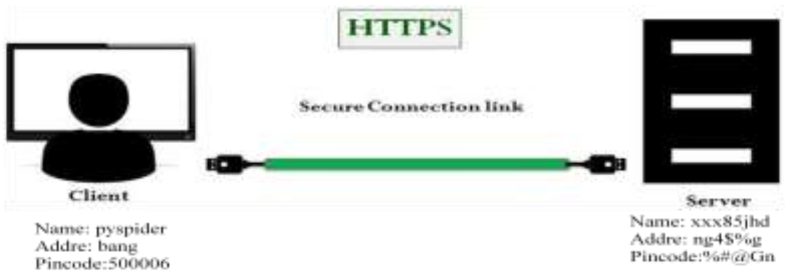
It is basic protocol that we use to send a request from the browser to a server. In this type request will be in readable format and it is a default protocol that developer going to used and it is a free protocol as well.



HTTPS:

HTTPS (Hypertext Transfer Protocol Secure) is nothing but the HTTP working in tandem with **SSL (Secure Socket Layer)** that is the "S" in HTTPS. SSL takes care of ensuring that the data goes securely over the internet. The alternative names given to HTTPS are HTTP over TLS, HTTP over SSL and HTTP secure.

This protocol was designed to increase primarily on the internet when communicating with web sites and sending sensitive data. This made man-in-the-middle attack increasingly difficult as the data send is no longer in plain text.



It is protocol we use to whenever we wanted to send the data in the encrypted format using request. Whenever we wanted to extract or sent the useful information and important information in that case go for 'https' protocol. It is paid protocol.

Diff between HTTP and HTTPS:

1. HTTP URL in your browser's address bar is http:// and the HTTPS URL is https://.
2. HTTP is unsecured while HTTPS is secured.
3. HTTP doesn't require domain validation, where as HTTPS requires at least domain validation and certain certificates even require legal document validation.
4. No encryption in HTTP, with HTTPS the data is encrypted before sending on the other hand encryption and decryption is used in https.
5. HTTP is subject to man-in-the middle and eavesdropping attacks(hacking) and HTTPS is designed to resist man-in-the middle and eavesdropping attacks and is considered secure against such attacks.
6. HTTP is no paid protocol and HTTPS is paid protocol.

Base URL:

A **base URL** is, basically, the consistent part of your web address. For example, throughout this training site, you'll note that the address section http://pyspider.com always appears in the address bar. This is the **base URL**. Everything that follows it is known as a **URL path**.

Base URL will decide where will go server (or) it contains the address of the server will the entry point address. The address of the server is called as IP address entry point address is called as port number.

BaseURL format : "http://IP address : port number"

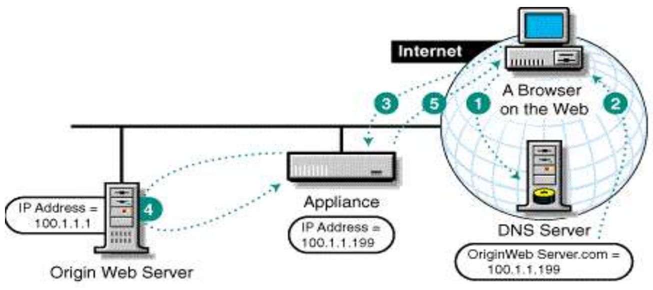
Ex: 120.0.0.1:8000

By default the IP address is development server is 127.0.0.1: and the port number is address in django is 8000.

Suffix:

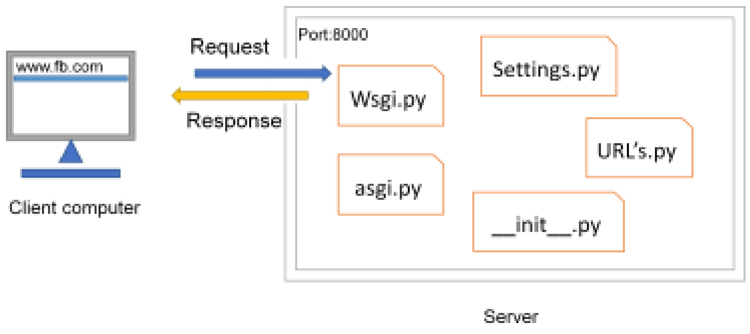
The suffix will decide to which particular function the control should be navigated to the suffix is segregated into various sub parts is primary suffix, secondary suffix and so on…

How to URL navigation in client and server:



1. A browser on the web requests an origin web server web page. This generates a request to DNS for the numeric IP address of the web server.
2. Instead of returning the origin web server numeric IP address, DNS returns the numeric IP address of the accelerator service on the application.
3. The browser requests the web page using the numeric IP address of the accelerator service.
4. The accelerator service obtains the web page objects from the origin web server.
5. The accelerator returns copies of the objects to the browser.

Coming to Django server Url Navigation:



- The process of navigating the request comes from the browser to a specific function or a class which is responsible to render the response to the client is called as URL navigation (URL mapping).
- URL's.py is a file which is responsible for url navigation.
- Based on the suffix of the url the mapping will be decided (or) the further navigation will be decided.
- Url's will map each and every url to a particular function or a class and this is called URL mapping.

- “Path” is a name of the function which is responsible for the url navigation(url mapping).

  Syntax:

  path(‘suffix/’, address_function, name = mapping_name),

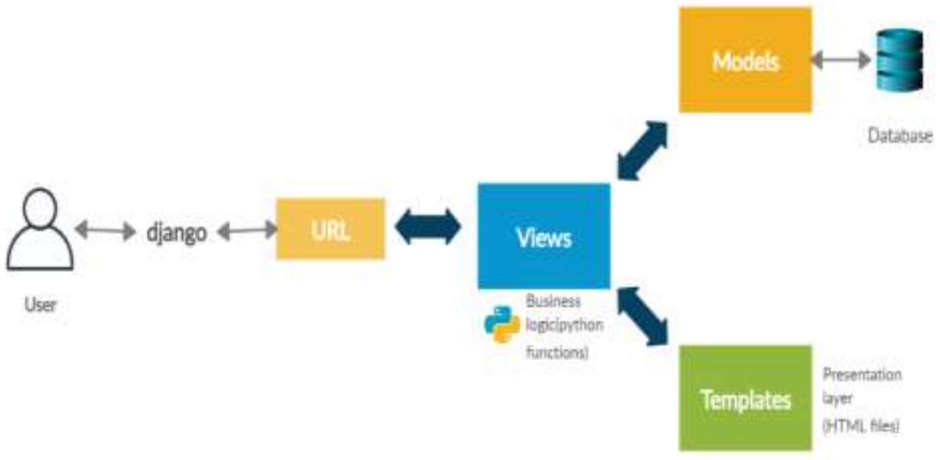
Design patterns:

It is the structure that decides where to keep what (it means in which file what contains should be return what is the purpose of the file or folder).

A Design pattern is also called as “Architect of the project”.

Django was initially using MVC (Model View Controller) as its design pattern.

- “M” stands for Model. It is section of in structure that is the responsible working with the database.
- “V” stands for View. It is section which consists of business logic (python files and instructions) which is responsible for the view rendering operation.
- “C” stands for Controller. It is section which consists of the instructions responsible to maintain the project or control the project.
- The controller part is taken care by Django application itself.


Models
Database
User
django
URL
Views
Business logic(python functions)
Templates
Presentation layer (HTML files)

Note:

The Model-View-Template (MVT) is slightly different from MVC. In fact the main difference between the two patterns is that Django itself takes care of the Controller part (Software Code that controls the interactions between the Model and View), leaving us with the template. The template is a HTML file mixed with Django Template Language (DTL).

How to open the django project in visual studio code:

By open the django project directly opening to the visual studio code.

1. Open visual studio code application in your system.
2. Click on file menu.
3. Click on open folder.
4. Select the path of the project folder and click on select folder button.
5. After doing the before step whatever you select project files will open into the visual studio code application.

How to create view in your project:

- A view is nothing but a function, or it is simply called a Python function that takes a web request and returns a web response.
- This response can be the HTML contents of a Web page, or a redirect, or a 404 error, or an XML document, or an image, etc. Example: You use view to create web pages, note that you need to associate a view to a URL to see it as a web page.

Creating the Custom View and Rendering Response:

Steps:

1. Create a file called **views.py** inside the project folder.

- This is file which is responsible for storing the functions that are responsible for rendering contents.
- To render the response to the http request we need to import "**HttpResponse**" from **django.http.**

  Syntax:

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

- To render the response to the http response we need to return the "**HttpResponse**" with HTML Tags or HTML file.

  Syntax:

  ```
  return HttpResponse("HTML tags or HTML file")
  ```

2. The function that we write must accept at least an argument to store the request which comes from the front end.

We prefer to give the name of the argument as "request" only.

3. Ones after writing the function we should do the URL mapping so go to URL's.py file and import the views from the project using the syntax.

Syntax:

```
from projectname import views
```

- Next the go to the URL patterns variable inside URL's.py file
- And set path function is return as

  path("suffix/", views. Name_of_the function, mapping_name),

4. After that save the all files and run that server (python manage.py runserver), copy the IP address of that server.
5. Open the browser and paste the IP address like (127.0.0.1:8000/home/) and press enter, display the HttpResponse of the screen.

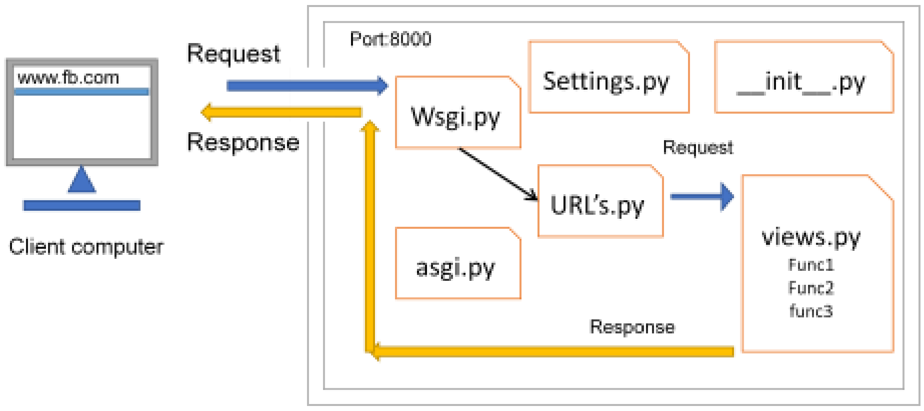


Fig: Work flow of client request and response of urls's.py and views.py

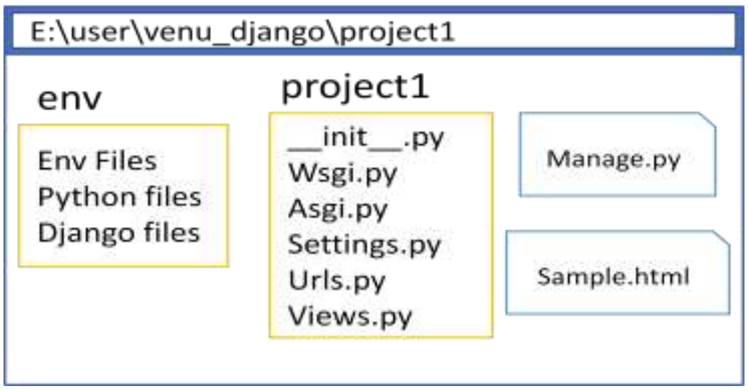


Fig: After create a views.py file inside the project files

Example for Sample view:

We will create a simple view in project1 to say "welcome to pyspider!"

See the following contents of views.py:

```
from django.http import HttpResponse

def hello(request):
  return HttpResponse("hello world")

def hello1(request):
  data = """<h1>welcome to pyspider !</h1>"""
  return HttpResponse(data)
```

```
def hii(request):
  return HttpResponse(“<h1>welcome to pyspider !</h1>”)
```

In this view, we use HttpResponse to render the HTML (as you have probably noticed we have the HTML hard coded in the view). To see this view as a page we just need to map it to a URL.

Now, we want to access that view via a URL. Django has his own way for URL mapping and it's done by editing your project url.py file **(project1/urls.py)**. The url.py file looks like:

See the following contents of urls.py:

```
from django.contrib import admin
from project1 import views

urlpatterns=[
          path(‘admin/’, admin.site.urls),
          path(‘home/’, views.hello, name=’home’),
          path(‘home1/’, views.hello1, name=’home1’),
          path(‘home2/’, views.hii, name=’home2’),
]
```

When a user makes a request for a page on your web app, Django controller takes over to look for the corresponding view via the urls.py file, and then return the HTML response or a 404 not found error, if not found. In url.py, the most important thing is the **"urlpatterns"** tuple. It’s where you define the mapping between URLs and views.

Note:

We used HttpResponse to render the HTML in the view before. This is not the best way to render pages. Django supports the MVT pattern so to make the precedent view, Django - MVT like, we will need

The process of rendering the complete html file:

Steps:

1. Create a html file inside a project, type your html data in that file
2. Read the html file and returns it as a response

Example:

Below program for views .py

```
from django.http import HttpResponse


def hml_respo(request):
  file_addr=open(r’ C:\Users\BMRCT\Desktop\venu_django\project1\sample.html’,”r”)
  data =file_addr.read()
  return HttpResponse(data)
```

if the project is system to system the path will be keep of changing and every time cannot sort and correcting the path so instead of we creating path we are go to generating the path dynamically using “OS” module.

Step:

1. To use “OS” module first we need to import it using the statement.
   Syntax:
   import os

2. Some of the functions use in that are:
   __file__: is variable that gives the path of the where it is present (current file).

a) **os.path.abspath(file)** (absolute path)
   This is a function that gives the absolute path
   Example:
   FILE_PRO = os.path.abspath(__file__)

b) os.path.dirname(absolute path)
   This is a function that path of parent directory by eliminating the child from abs path
   Example:
   DIR_PRO = os.path.dirname(FILE_PRO)

c) os.path.join(path1, path2)
   This is functions which is joining the directory file and add the new file.

   Example:
   FILE_PATH = os.path.join(DIR_PRO,”sample.html”)

Example:

```
import os

print(__file__)

FILE_PRO = os.path.abspath(__file__)
print(FILE_PRO)

DIR_PRO = os.path.dirname(FILE_PRO)
print(DIR_PRO)

FILE_PATH = os.path.join(DIR_PRO,”sample.html”)
print(FILE_PATH)
```

output:

```
C:/Users/BMRCT/AppData/Local/Programs/Python/Python38/programs/rendering.py
C:\Users\BMRCT\AppData\Local\Programs\Python\Python38\programs\rendering.py
C:\Users\BMRCT\AppData\Local\Programs\Python\Python38\programs
C:\Users\BMRCT\AppData\Local\Programs\Python\Python38\programs\sample.html
```

Example for project1:

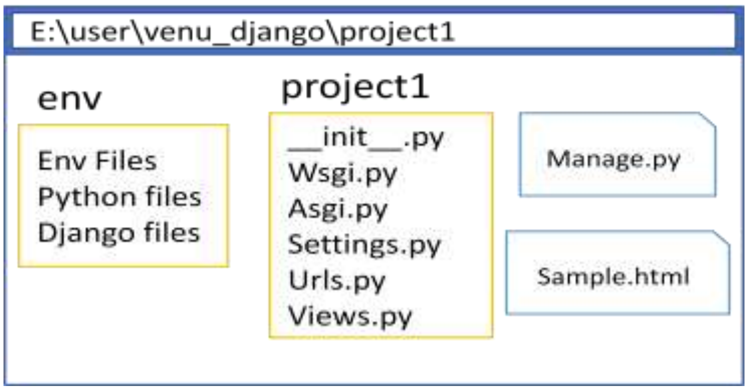


Fig: After the creating a views.py file inside the project1

See the below following contents of sample.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Document</title>
</head>
<body>
   <h1>hello world</h1>
   <marquee direction=\"right\"><h1>welcome to my world</h1></marquee>
</body>
</html>
```

See the below following contents of views.py:

```
from django.http import HttpResponse
import os

FILE_PRO = os.path.abspath(__file__)
DIR_PRO = os.path.dirname(FILE_PRO)
```

```
def index(request):
     return HttpResponse("<h1>hello world</h1>")

def home(request):
    FILE_PATH = os.path.join(DIR_PRO,"sample.html")
    Fp = open(FILE_PATH,"r")
    Data = Fp.read()
     return HttpResponse(Data)
```

See the below following contents of urls.py:

```
from django.contrib import admin
from project1 import views

urlpatterns=[
            path('admin/', admin.site.urls),
            path('index/', views.index, name='index'),
            path('home/', views.home, name='home'),
]
```

Organizing the html file:

Instead of writing an html where ever we want the group everything and we store it inside the directory called templates.

Templates are a directory containing all the html files that are related to the project.

When we follow this approach than the architecture of the framework will be changed as MVT (it is container of html files) from MVC.

Once after create a template directory we need to register the templates directory with the project using settings.

What are Django templates?

1. Django templates are a combination of static HTML layout/pages and django syntax (statements/commands) which is essentially python code.
2. Both of these components help in generating html pages dynamically and make your website more user-engaging.
3. The purpose of the templates severing html pages (static/dynamic)to the users screen.
4. The main purpose of django templates is to separate the data representation with the data itself
5. That means the browser part is only to render the html sent to it by the server, and all the relevant data is given to the template by django itself.

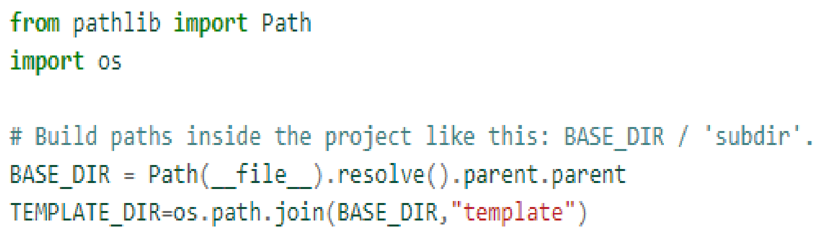
6. This makes the process mush easier and page render easily as there is less clutter in both the front end and back end
7. Django templates is done via template engine, there are multiple templates engines(JINJA template).
8. Django template engine as you can see in your projects' setting.py file.

Step: <u>Register the Template Directory:</u>

1. Open the settings.py
2. Type the **import os**
3. After the write **BASE_DIR** statement type below syntax.

   TEMPALTE_DIR = os.path.join(BASE_DIR, ”templates”)

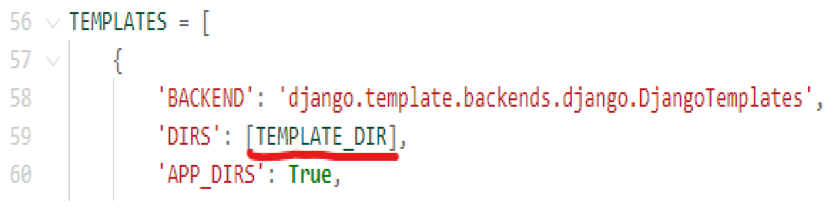
   BASE_DIR contains the address of the project connect the templates along with **os.path.join function.**



```
from pathlib import Path
import os

# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent
TEMPLATE_DIR=os.path.join(BASE_DIR,"template")
```

Fig: connect the template folder into the project.

4. Store the join the path in the variable with the name “**TEMPLATE_DIR”.**
5. Scroll down find templates list I which find the key called “**DIRS**” with a list value append your **TEMPLATE_DIR** to that list.



```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [TEMPLATE_DIR],
        'APP_DIRS': True,
```

Fig: connect the template folder into the project directory.

6. Restart the server.

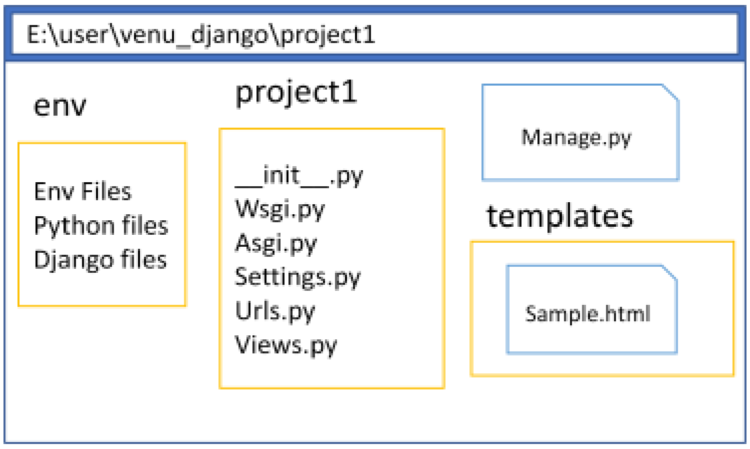


Fig: After creating a template folder inside the project1

Once after restarting the server application will be knowing the address of templates directory because of that in our render function we do the specify the complete path of file.

<u>Sending html file as a response:</u>

Rendering:

- It is phenomenal of sending the html file as a response.
- Rendering operation done by using render function called **"render ()"**.
- Render function is associated with return statement.

Syntax:

return render(request, "path_of_html_file")

Example:

return render(request,"sample.html")

- To use the render function we need to import function called django shortcuts.

Syntax: from django.shortcuts import render

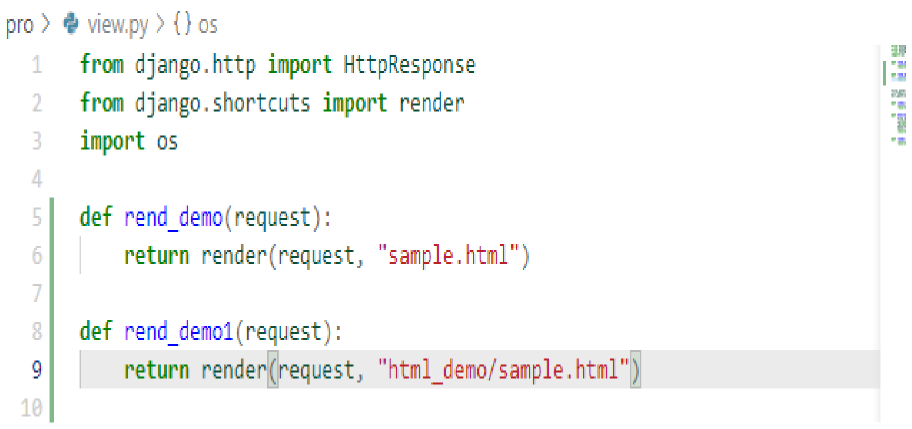
When we wanted sample.html in the template we just need to specify the name of the file with extension. For example,

```
def  rend_demo(request):
        return render(request, "sample.html")
```

If we wanted to render present in the another directory for example html_demo folder than we should specify like below example.

```
def  rend_demo1(request):
        return render(request, "html_demo/sample.html")
```

*** In views.py file in project1**



```
pro > view.py > {} os
from django.http import HttpResponse
from django.shortcuts import render
import os

def rend_demo(request):
    return render(request, "sample.html")

def rend_demo1(request):
    return render(request, "html_demo/sample.html")

```

Fig: Example for view.py in the project1 with render functions.

Changing the IP address of DJANGO SERVER:

We can change the IP address of the server only to the system IP address. Find the system IP address by using keyword called **IPCONFIG,** search in command prompt.

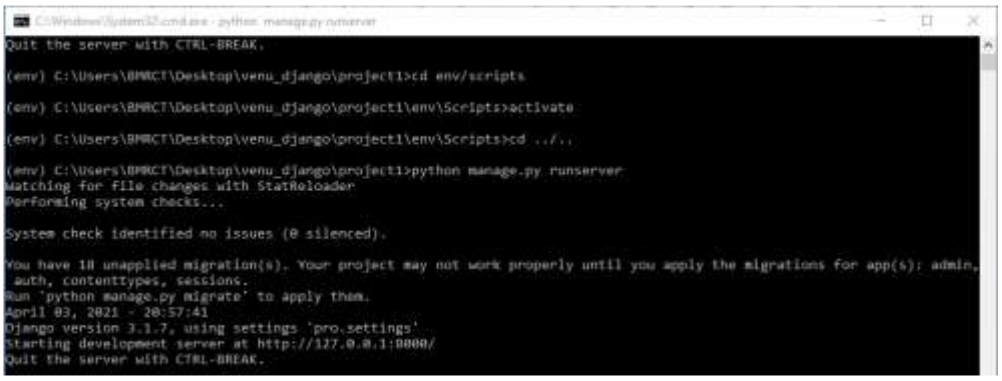
Steps that we follow to change the IP address:

1. Go to setting.py find the variable called "**ALLOWED_HOSTS**".
2. In that insert a string containing the IP address, save it.
   Example: IP: **192.127.1.131**
   (Or)
   Insert a string containing the "*". Allow for any IP address, save it.

3. Go to command prompt and type the below syntax.
   python manage.py runserver IP_address : portnumber

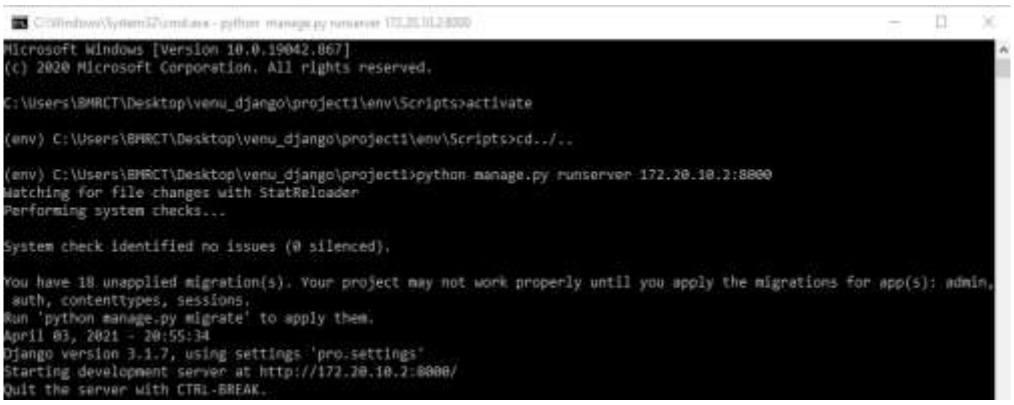
Example: python manage.py runserver 197.127.1.131:8000

Note: once we change the IP address from development environment IP address to any other IP address again if we want to run it development environment we should we specify the development environment IP address in the ALLOWES_HOST.



```
C:\Windows\System32\cmd.exe - python manage.py runserver
Quit the server with CTRL-BREAK.

(env) C:\Users\BMRCT\Desktop\venu_django\project1>cd env/scripts

(env) C:\Users\BMRCT\Desktop\venu_django\project1\env\Scripts>activate

(env) C:\Users\BMRCT\Desktop\venu_django\project1\env\Scripts>cd ../..

(env) C:\Users\BMRCT\Desktop\venu_django\project1>python manage.py runserver
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 work properly until you apply the migrations for app(s): admin,
 auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
April 03, 2021 - 20:57:41
Django version 3.1.7, using settings 'pro.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Fig: Default IP address of the django server



```
C:\Windows\System32\cmd.exe - python manage.py runserver 172.20.10.2:8000
Microsoft Windows [Version 10.0.19042.867]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\BMRCT\Desktop\venu_django\project1\env\Scripts>activate

(env) C:\Users\BMRCT\Desktop\venu_django\project1\env\Scripts>cd../..

(env) C:\Users\BMRCT\Desktop\venu_django\project1>python manage.py runserver 172.20.10.2:8000
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 work properly until you apply the migrations for app(s): admin,
 auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
April 03, 2021 - 20:55:34
Django version 3.1.7, using settings 'pro.settings'
Starting development server at http://172.20.10.2:8000/
Quit the server with CTRL-BREAK.
```

Fig: After changing the IP address of the django server

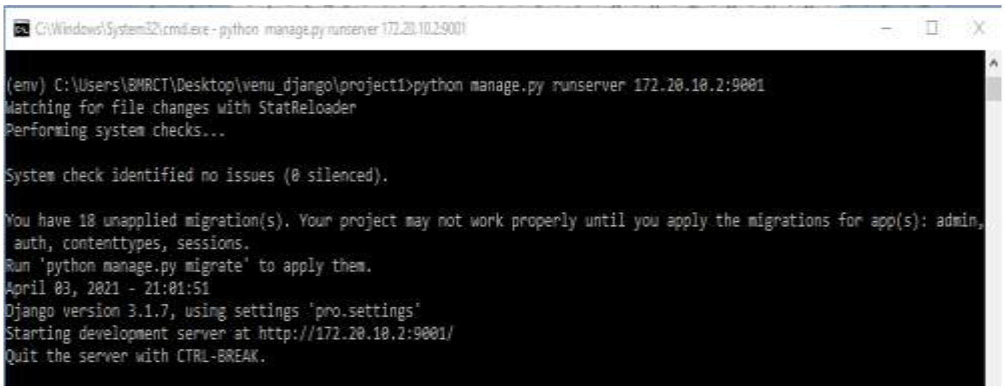
Changing the port number of DJANGO SERVER:

The default django server port number is **“8000”**.

When we change the port number in your server just follow the below commands and press enter button.

Python manage.py runserver IP_address:new_port_number

Note: the port number that we are specify must not be occupied by any other server.



```
C:\Windows\System32\cmd.exe - python manage.py runserver 172.20.10.2:9001

(env) C:\Users\BMRCT\Desktop\venu_django\project1>python manage.py runserver 172.20.10.2:9001
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 work properly until you apply the migrations for app(s): admin,
 auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
April 03, 2021 - 21:01:51
Django version 3.1.7, using settings 'pro.settings'
Starting development server at http://172.20.10.2:9001/
Quit the server with CTRL-BREAK.
```

Fig: After changing the port number is **“9001”**

In django we can develop the projects and application.

What is project?

It is collection of front end, business logic, back end.

A project refers to the entire application and all its parts.

A projects can contain several smaller application that serve a particular function or purpose.

Example for project

Facebook, youtude, gmail, google

Project :

its simply name of the your website(facebook). Django will create a python package and give it a name that you have provided. Lets say we name it (facebook).

Method1:

```
Django-admin startproject facebook

{Or}

Method2:
    Python manage.py startproject facebook
```

Django project is a collection of applications and configuration.

What is django application?

A django application refers to a sub module of the django project.
An application serves one basic functionalities or purpose.
Example for application:

Facebook:
Login, register, homepage, about us, profile, massages, newsfeed, friends list.

This will create (facebook) directory in your working directory and the structure will look like this:

Apps : (applications)
Those little sub modules/components that together make up your project. They are the features of your project in our case (facebook):

Login: This would have logic for user authentication information.
Register: This would have logic for user related information.
Home page: This would have logic for some website information/ posts.

Why you are going to django applications.

When we consider a large -project there will be lot of features/views that will be present inside the project.

If ever we write a set of features inside the project. it will be every difficult for us to handling the project (error, debugging, any modification, new features add, updating) and it is going to increase the time taken for the rendering the things.

To avoid this problem we have segregating our project into set of functionalities.
Where each and every functionalities (or) collecting of interrelated functionalities are been segregated into something we called as application (app's).

Segregating the project into App’s:

When we consider a large project there will be lot of views that will be present inside the views.py file.

If ever we write a one location it will be very difficult for us to handling (error comes) and it is going to increase the time taken for the rendering things.

To avoid this problem we have segregated our project into set of functionalities. Where each and every functionality (or) collection of interrelated functionalities are been segregated into something is called "**app's**".

AN APPLICATION (APP):

An application is a large feature (or) a collection of interrelated features in a project.

We segregated this because we need to increase the efficiently (or) reduce our time consumes.

Note:
Always the django application should be inside the project (project container).

Always the app should be inside the project to create an app we use syntax is:

Python manage.py startapp app_name

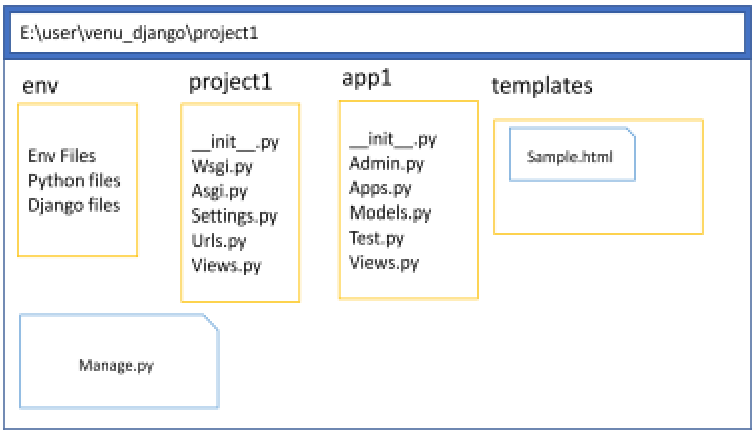


FIG: after the creating a app in project1

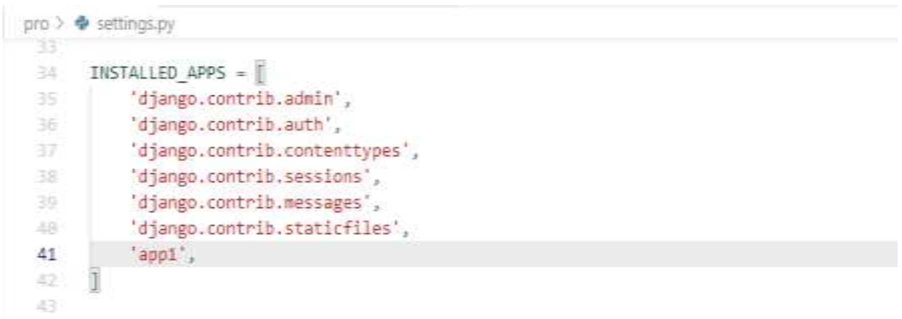
When we create app folder (or) app package will be created containing the directory as migration and such files as,

a) __init__.py
b) Admin.py
c) App.py
d) Models.py
e) Tests.py
f) Views.py

After creating an app connected to the project by using "setting.py".

Steps:

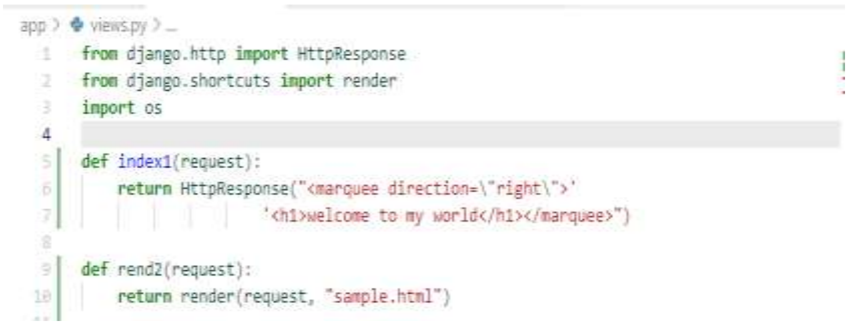
1. Open setting.py file in project.
2. Scroll down the file and find the section called **INSTALLED_APPS** list.
3. In that section append the **APP_NAME** as a string.
4. And to check the app is successfully created or not.



```
pro > settings.py

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'app1',
]
```

Fig: append the app_name into INSTALLED_APPS variable.

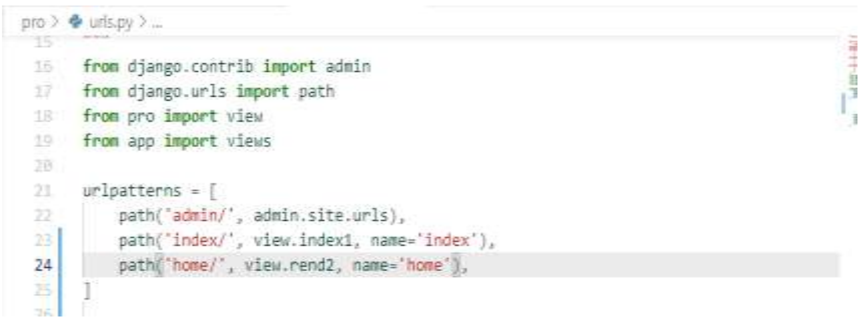
In a views.py file of an app folder in a project:



```
app > views.py > ...
from django.http import HttpResponse
from django.shortcuts import render
import os

def index1(request):
    return HttpResponse("<marquee direction=\"right\">"
                        '<h1>welcome to my world</h1></marquee>")

def rend2(request):
    return render(request, "sample.html")
```

Fig: views.py file present in the app folder

In a urls.py file of an project folder:



```
pro > urls.py > ...
from django.contrib import admin
from django.urls import path
from pro import view
from app import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('index/', view.index1, name='index'),
    path('home/', view.rend2, name='home'),
]
```

Fig: urls's.py file present in the project1 folder

NOTE: run server; copy the IP address of that server. And open any browser paste the ip address with suffix and check the app is working or not

Describe about files present in app:

__init__.py:

It is initialization file. It is blank python script file. Python treated this file as a package.

Admin.py:

In this file contains the code which is reasonable for giving the feature to the admin. If ever we want to give any access to the admin the code with respect to that will be written in admin.py

App.py:

It is file containing the configuration of an "app".

Models.py:

It is the database related file was we write our code to create a table inside the database.

Test.py:

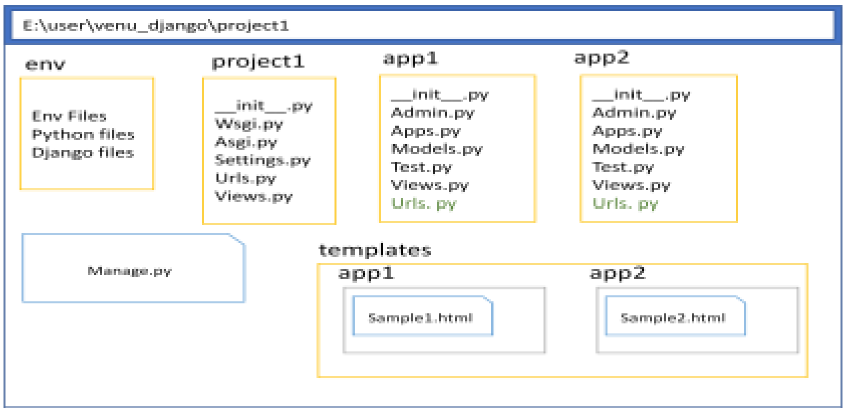
It is file that contains the test cases (unit test cases) that are written to automate the project.

Views.py:

It is file containing the functions (or) the classes that are reasonable for views operations rendering the response to the front end.

Creating and connecting a specific urls's directory for "app's":

When ever we have multiple apps at that time importing the views in the URL's file will create a problem where the views of apps1 will get to override with views of app2 because both are having the same name as views only. To avoid this problem we came up with a solution of creating the alternate variables and storing the address of views inside so that it wills not a problem. so



<u>Create and connect the specific url's file(child url.py file in each app).</u>

Steps to:

1. Create a file called urls.py in each app.
2. In that urls.py file import the path module and import views.py file inside the app.
   Example:

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

3. Create a variable app_name for the unique identification
4. And store it with the value as string containing appname for further purpose.
   Example:

   userapp = "USERAPP"

5. Create a list with the name called urlpatterns write your path function.
6. Their path function is connecting the app specific urls to the project urls.
7. To go project urls.py file (parent urls.py file) and import the include module (django.urls).
   Example:
   from django.urls import path,include

8. And import the app
   Example:
   Import appname
9. in urlpatterns list to create a path function with the below syntax:
   Syntax:
   Path("appsuffixname/", include("appname.urls")),

10. Based on the app suffic name the application is going to decide whether the url navigation should be done views or app specific urls.

See the below following contents of templates/app1/sample1.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>sample1</title>
</head>
<body>
   <h1>welcome to sample1 of app1</h1>
   <marquee direction=\"right\"><h1>welcome to my world</h1></marquee>
</body>
</html>
```

See the below following contents of templates/app2/sample2.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>sample1</title>
</head>
<body>
   <h1>welcome to sample2 of app2</h1>
   <marquee direction=\"right\"><h1>welcome to my world</h1></marquee>
</body>
</html>
```

Views.py in app1 folder:

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

def index1(request):
     return HttpResponse("<h1>hello world</h1>")

def home1(request):
     return render(request,'app1/sample1.html')
```

Views.py in app2 folder:

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

def index2(request):
     return HttpResponse("<h1>hello world</h1>")

def home2(request):
     return render(request,'app2/sample2.html')
```

urls.py file in project1 folder:

```
from django.shortcuts import render
from django.urls import path
from pro import views

from app1 import views
views_app1="views"

from app2 import views
views_app2="views"
urlpatterns =[
                path('admin/'admin.site.urls),
                path('index1/', views_app1.index1, name='index1'),
                path('home1/', views_app1.home1, name='home1'),

                path('index2/', views_app2.index2, name='index2'),
                path('home2/', views_app2.home2, name='home2'),

]
```

Note:

Now if the control wanted sample2 to it need get its comparison done with suffix of 5 path function so to reduces it time consumption and increase the efficient we come up the approach of creating the specific urls.py file for each and every app's.

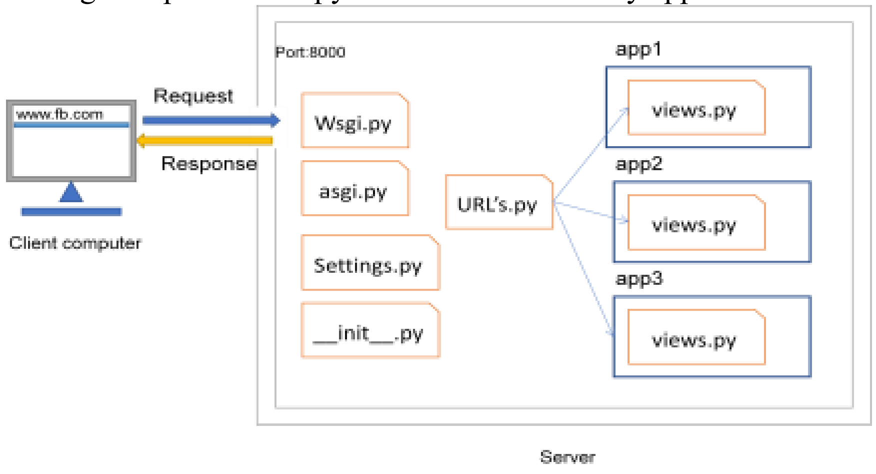


Fig: Generic urls.py in all the app's (general url mapping).

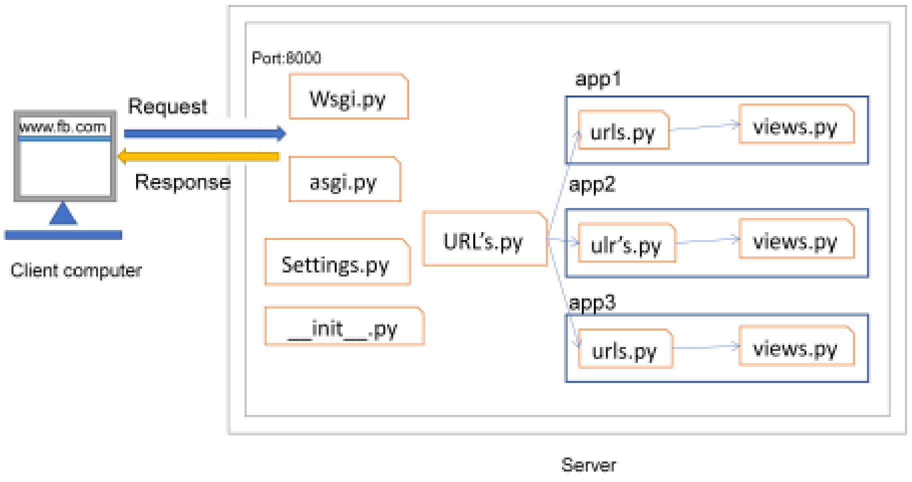


Fig: specific urls.py in all the app's (specific url mapping).

App1 specific url's content (app specific url's of app1)

```
from django.shortcuts import render
from django.urls import path

from app1 import views

urlpatterns =[
            path('admin/'admin.site.urls),
            path('index1/', views.index1, name='index1'),
            path('home1/', views.home1, name='home1'),

]
```

App2 specific url's content (app specific url's of app2)

```
from django.shortcuts import render
from django.urls import path

from app2 import views

urlpatterns =[
            path('admin/'admin.site.urls),
            path('index2/', views.index2, name='index2'),
            path('home2/', views.home2, name='home2'),

]
```

Project1 specific url's content (project1 specific url's)

```
from django.shortcuts import render
from django.urls import path, include

import app1
import app2

urlpatterns =[
            path('admin/'admin.site.urls),
            path('app1/', include("app1.ulr's")),
            path('app2/', include("app2.ulr's")),

]
```

Creating a specific templates directory for each app:

Whenever we have multiple html files (or) multiple templates it will be very difficult to handle it in one template directory.

Every time when we create a new app we need to disturb that particular template directory that causes the inefficient problem so we have come up with a concept of creating the specific template directory for each app.

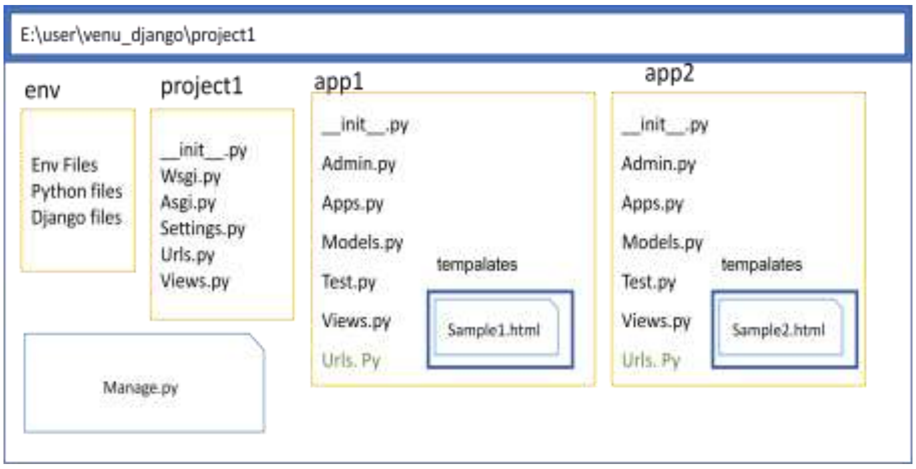


Fig: After create specific templates in each app.

Steps to create and concept the specific templates directory and connecting it:

1. Create a directory called templates in each app.
2. Store the path of the app specific templates directory in a variable "TEMPLATE_DIR_APPNAME".

   <u>Project1/template:</u>
   TEMPLATE_DIR = os.path.join(BASE_DIR,"templates")

   <u>Project1/app1/template:</u>
   App1=os.path.join(BASE_DIR,"app1")
   TEMPLATE_DIR_APP1 = os.path.join(app1,"templates")

   <u>Project1/app2/template:</u>
   App2=os.path.join(BASE_DIR,"app2")
   TEMPLATE_DIR_APP2 = os.path.join(app2,"templates")

3. Append the path variable to the list of template DIRS.

   DIRS= [TEMPLATE_DIR, TEMPLATE_DIR_APP1, TEMPLATE_DIR_APP2]

4. Save the all the files in project. And run server check whatever we doing process are working or not.

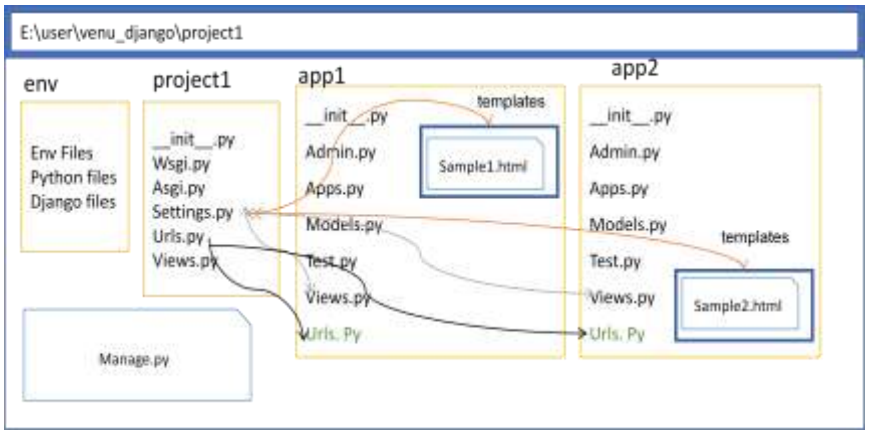


Fig: After connect the app’s template, views, urls mapping to the project1 files.

What is Django template language (DTL)?

Django needs a simple method for dynamically generating HTML. Templates are the most popular method in django framework. A template contains the static sections of the desired HTML output, as well as some special syntax for inserting dynamic content.

A Django project can be configured with one or several template engines (or even zero if you don’t use templates). Django includes backends for both its own template scheme and DTL (Django template language), and for the popular alternative Jinja2. Backends for other template languages may be available from third-parties.

The Django template language is Django’s own template system. A front end developer can simply leave html comments.(whenever he wants db and other information from django). Later a programmer can simply replace them with a template language is known as DTL

Django templates are text documents, frontend related documents (html, css, javascripts), Python strings that have been marked up with the Django template language. The template engine recognizes and interprets certain constructs. That is Variables and/or expression, which gets replaced with values when a templates rendering and tags are the most important. Which control the logic of the templates

A template is rendered with a context. Rendering replaces variables with their values, which are looked up in the context, and executes tags. Everything else is output as is.

The syntax of the Django template language (jinja2 Tags) involves four constructs.

a) Templates Variable:
b) Templates Tags
c) Templates Filters
d) Templates Comments

1) Templates Variable:

Template variable are similar to the variable used in python.
Templates variable are defined by the context dictionary passed to the templates.
A variable is to retrieve the data from the context and gives it.
Variables are surrounded 2pair of flower brackets by like this:

Syntax:

{{Value or Variable}}

Example:

Simple variable →{{variable}}, {{value}}
List attributes →{{list.0}}
Object attributes →{{name. title}}
Dictionary attributes →{{dict.key}}

Note: list.0 is nothing but list[0] in python but in jinja tag we are using list.0 to access first element of a python list.

2) Templates Tags:

- Templates tags to perform specific task.
- This are called as 3rd party tags
- It is used with the html code to perform some operation of the data that we send it from back-ends.
- To do some operations {% operation syntax %}. The specified operations will be performs again in this we have 2 types.

1) Single tags:

- We will just specify {% operation syntax %}
- No closed tags or end tags.

Example: {% static ‘path’ %}

Content import:

```
{% include “header.html” %}
```

Inheritance:

```
{% extends “base.html” %}
```

2) Dual tags:

- We will just specify {% operation syntax %} with closed tags or end tags.
- We specify the boundary to utilize these tags.

Syntax:

```
{% operation syntax %}
    TRUE STATEMENT BLOCK
{% end syntax %}
```

Condition statement/display logic:

Simple if condition:

```
{% if <condition>  %}
        TSB
{%endif%}
```

Simple if else condition:

```
{% if <condition>  %}
        TSB
{%else%}
        FSB
{%endif%}
```

Simple elif if else condition:

```
{% if <condition> %}
        TSB
{% elif <condition> %}
```

```
        TSB
{% else%}
        FSB
{%endif%}
```

Simple nested if condition:

```
{% if <condition> %}
        {% if <condition> %}
                {% if <condition> %}
                        TSB
                {%endif%}

        {%endif%}
{% elif <condition> %}
        TSB
{% else%}
        FSB
{%endif%}
```

Looping statement/display logic:

```
{% for x in collection %}
        TSB
{% endfor %}
```

Block Declaration:

```
{% block content %}
        Statements
{% endblock %}
```

3) Template Filter:

- A template filter is used for filtering the variable/attributes.
- The symbol pipe (|) is used for indicating filters.
- Django Template Engine provides filters which are used to transform the values of variables and tag arguments. Tags can't modify value of a variable whereas filters can be used for incrementing value of a variable or modifying it to one's own need.

  Syntax:

  ```
  {{ variable_name | filter_name }}
  ```

Some Template filters are:

Add : it is used to add an argument/value to the initial value

Syntax: {{ value | add: " value"}}

Title: It is used to convert a string into title case by making word start with an uppercase and remaining characters is lower case

Syntax:{{ value | title}}

Upper: It is used to convert a string into all uppercase.

Syntax: {{value | upper}}

Lower: It is used to converts a string into all lowercase.
Syntax: {{value | lower}}

Word count : It is used to return the number of words.
Syntax: {{value | wordcount}}

Length: It is used to return the length of the value. This works for both strings and lists.
Syntax: {{value | length}}

Line numbers: It is used to display text with line numbers.
Syntax: {{value | linenumber}}

Randow: It is used to return a random item from the given list.
Syntax: {{value | random}}

Slice: It is used to return a slice of the list.
Syntax: {{Value | slice : "startpoint : endpoint : updation"}}

Unordered list: It is used to recursively take a self-nested list and returns an HTML unordered list – WITHOUT opening and closing <ul> tags.
Syntax: {{value | unorderd_list}}

First: It is used to return the first item in a list.
Syntax: {{value | first}}

Last: It is used to return the last item in a list.
Syntax: {{value | last}}

Join: It is used to join a list with a string, like Python's str.join(list)
Syntax: {{value | join:"value"}}

Default: It is used to to give a default value to a variable. If variable evaluates to False, it will use the default argument given else the variable value itself.
Syntax: {{value | default:" value"}}

Cut: It is used to remove all values of arg from the given string.
Syntax: {{value | cut:" value"}}

Center: It is used to center the value in a field of a given width.
Syntax: {{value | center}}

Truncation:
Syntax: {{name | truncatewords :80}}
This means that from the name, show only first 80 i.e, truncate the name to the first 80 characters.

Capfirst: It is used to capitalize the first character of the value. If the first character is not a letter, this filter has no effect.
Syntax: {{value | capfirst}}

4) Template Comments:

{# comments#} for comments not included in the templates output.

Sending the data from the backend to it front end:

- To send the data from the backend to the frontend we are going to utilize an argument called "context" in the render function.
- To the "context" argument we should end the dictionary of the data the keys in the dictionary must be the variable names which we use it in the frontend values is nothing but that data that I wanted to send.
- In the html we use 1st category of jinja tag to retrieve the value present in the variable.

Render function:

Django relies on the render function and the django function.

This function takes three parameters.

1) Request
2) The path of the templates(html file)
3) Dictionary of parameters(context)

Syntax:

Return render(request, path_of_template, context)

- Request: the initial request to the user.
- The path of the templates is the path relatives to the TEMPLATES_DIRS option in the project settings.py variables.
- The dictionary of parameters that contains all variables needed in the templates. This variable can be created on dictionary (to pass local variable declared in the function/view).

Example:

```
Def sample (request):
        Data="hello world"
        return render(request, sample.html, {"data":Data})
```

In view.py of app:

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

def index2(request):
    return HttpResponse("<h1>hello world</h1>")

def home2(request, email):
    return render(request,'sample.html',context={'email:'email,'name':"pyspider"})
```

In sample.html file of templates:

```
<!Doctype html>
<html>
<body>
            <h1>welcome to html page</h1>
            <h1>my email is {{email}}</h1>
            <h1>my name is {{name}}</h1>

</body>
</html>
```

In urls.py in app:

```
from django.urls import path
from dboyapp import views
urlpatterns = [
    path("sample/",views.sample),
    path("home2/",views.sample1),
]
```

1) **One more example:** Application to take the login data through url and check user login authentication and render the home page content like data and time and student information from views.py file to templates(html)file.

In login.html file inside the app templates:

```
 <!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1>this is login page </h1>
    {%if email == 'raja@gmail.com'%}
        {%if pass == '12345' %}
            {#<h1>current {{dt}}</h1>#}
            <h1>student information</h1>
            <h2>student name is{{sname|add:"hello world"}}</h2> <br>
            <h2>student name is{{sname|title}}</h2> <br>
            <h2>student roll is{{sroll}}</h2><br>
            <h2>student address is{{saddress|wordcount}}</h2><br>
            <h2>student branch is{{sbranch|length}}</h2><br>
                        <h2>it is list random value is display {{list1|random}}</h2><br>
            <h2>it is list slicing value is display {{list1|unordered_list}}</h2><br>
            <h2>list first value is {{list1|first}}</h2><br>
            <h2>list last value is {{list1|last}}</h2><br>
            <h2>student address is{{saddress|join:"+"}}</h2><br>
            <h2>list last value is {{list1|join:"+"}}</h2><br>
            <h1>{{val|default:"100"}}</h1> <br>
            <h2>student address is{{saddress|cut:" "}}</h2>
        {%else%}
            <h1>invaild password</h1>
        {%endif%}
    {%else%}
        <h1>invaild username</h1>
    {%endif%}

</body>
</html>
```

In view.py of app:

```
from django.shortcuts import render
import datetime

def sample(request):
    return render(request, "sample.html")

def login(request, email, password):
    date=datetime.datetime.now()
    return render(request, "login.html",{'email':email, 'pass':password, 'dt':date, 'sname':'raja',
        'sroll' : 'pys123', 'sbranch':'django' ,'saddress':'bangalore btm basvanagudi chennai hyd ',
        'list1':[10,20,30,'helloworld','haii'],'val':3000})
```

In urls.py of app:

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

urlpatterns = [
    path('sample/', views.sample),
    path('login/<email>/<password>/', views.login),
```

In urls.py of project:

```
from django.contrib import admin
from django.urls import path,include
import ola

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

Search the url inside the browser

```
http://127.0.0.1:8000/ola/login/raja@gmail.com/12345/
```

The process of creating url navigation from one file to another file.

The process of creating a url navigation from one file to another file or one page to another page is called as url navigation it is done by using the anchor tag <a href “ ” > and template navigation tag.

The <a> tag defines a hyperlink, which is used to link from one page to another. The most important attribute of the <a> element is the href attribute, which indicates the link's destination.

Whenever we specify the anchor tag should specify the reference of the page to which the navigation should happen. But in changes when we have the project from system to system to avoid lot of many changes we have come up with a jinja tag where navigation path will be taken dynamically.

Syntax of jinja tag is:

{% url “name of mapping” %} --→ (parent urls.py)

It is mapping has used only when the mapping has to the parent (or) root url file.

{% url “appname : name of mapping ” %} --→ (child urls.py)

It is mapping has used only when the mapping has to the child (or) root url file.

Example:

Generic url path(parent url)

```
<a href = "{% url 'sample1'%}" >Sample1</a>
```

Specific app url path(child url)

```
<a href = "{% url 'myapp:sample3' %}" >Sample3</a>
```

Whenever url carrying the data in that time we use below syntax:

Syntax:

{% url “name of mapping” var = value %} --→ (parent urls.py)

It is mapping has used only when the mapping has to the parent (or) root url file with carrying the data.

{% url “appname : name of mapping ” var = value%} --→ (child urls.py)

It is mapping has used only when the mapping has to the child (or) root url file with carrying the data.

Whenever the name of the url space is not specified then use the jinja tags same as the tag for parent file url navigation.

Example:

This is sample1.html file inside the project/templates folder.

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Sample1</title>
</head>
<body>
   <h1>Welcome to sample.html Page</h1>
   <a href="{% url 'myapp:sample2' data='Hello World' %}">Sample2</a>
   <br><a href="{% url 'myapp:sample3' %}">Sample3</a>
</body>
</html>
```

This is sample2.html file inside the project/templates/myapp folder.

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Sample2</title>
</head>
<body>
   <h1>Welcome to sample2.html Page</h1>
   <h2>I am Carrying the data in the URL and the data is {{data}}</h3>
   <a href="{% url 'sample1' %}">Sample1</a><br>
   <a href="{% url 'myapp:sample3' %}">Sample3</a>
   </body>
</html>
```

This is sample3.html file inside the project/templates/myapp folder.

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
    <title>Sample3</title>
</head>
<body>
    <h1>Welcome to sample3</h1>
    <h2>Data Carried from Backend {{name}}</h2>
    <a href="{% url 'sample1' %}">Sample1</a>
    <br><a href="{% url 'myapp:sample2' data=name %}">Sample2</a>
</body>
</html>
```

This is views.py file inside the project/myapp folder.

```
from django.shortcuts import render

# Create your views here.
def sample1(request):
    return render(request,"sample1.html")

def sample2(request,data):
    return render(request,"myapp/sample2.html",context={'data':data})

def sample3(request):
    return render(request,"myapp/sample3.html",context={'name':"akshay"})
```

This is urls.py file inside the project/myapp folder.

```
from django.urls import path
from myapp import views
app_name="myapp"

#name of the urlspace
#name of the url space is not mandatory
#if it is not given navigation jinja tag will be {% url 'mappingname' %}

urlpatterns = [
    path('sample2/<data>',views.sample2,name="sample2"),
    path('sample3/',views.sample3,name="sample3")
```

This is urls.py file inside the project folder.

```
from django.contrib import admin
from django.urls import path,include
import myapp
```

```
from myapp import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('',include("myapp.urls")),
    path('sample1/',views.sample1,name="sample1"),
]
```

Templates inheritance:

The process of deriving the features of one template (html page) to other template (html page) is called as template inheritance.

Whenever we work with respect to the real time application there are many features like top bar, navigation bars, footer bar, some others features, will be same in the each and every templates (html file). Instead of repeating the code again and again we came up with the concept of templates inheritance and we utilizing in all the situations.

If one project multiple templates files have some common code, it is not recommended to write that common code in every template html file. It increases length of the code and reduces readability. It also increases development time.

We have to separate that common code into a new templates file, which is also known as base template html file. The remaining template files should required extending base template so that the common code will be inherited automatically.

Inheriting common code from base templates to remaining templates is nothing but template inheritances.

Templates inheritances tags:

Extends jinja tag:

{% extends "filename_with_path .extension" %}

In general, *extend* a file to use its code as a base for multiple templates

Django extend template tag:

Using the extends tag in Django requires several things.

(1) First, you need a Django template to extend. This is the template whose base code you want to use for other templates.

(2) Next, you need to add the Django extend block content tags where each of the other templates will be loaded in.

(3) Finally, you need to extend the first template at the top of the other templates and add a matching set of Django extend block tags to the other templates

Block contents jinja tags:

The block of code which we wanted to override will be return inside the block jinja tag. Anything which is return inside the block jinja tag can be override in the child html file by repeating the creating the same block with the block and with the diff content.

Block names:

Block name can be title if we wanted to override the title content of title tag.

Syntax:

```
{% block bodyname %}
        <title>
                Body content
        </title>
{% endblock %}
```

Body block:

If we wanted to override the entire body then we go for body block and body tag will be specified inside that.

Syntax:

```
{% block body_name %}
        <body>
                Body content
        </body>
{% endblock %}
```

Body content:

Whenever we don’t want to override the entire body instead we wanted to override the position of the body in that case we are going to utilize the content block.

Syntax:

```
{% block content %}
        Override contents inside the body tags
{% endblock %}
```

Example:

Parent html file:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
          {% block title %}
          <title>Sample3</title>
          {% endblock %}
</head>
<body>
{%block content%}
   <h1>Welcome to sample3</h1>
   <h2>Data Carried from Backend {{name}}</h2>
   <a href="{% url 'sample1' %}">Sample1</a>
   <br><a href="{% url 'myapp:sample2' data=name %}">Sample2</a>
{% endblock %}
</body>
</html>
```

Child html file:

```
{% extends “parentfile.html” %}

{% block title %}

<title> sample2 </title>

{% endblock %}

{% block content %}

          content

{% endblock %}
```

Partials: (Templates Include)

When we are doing the real time projects our html file will be having lot of features and the code related to it.

As exists lot of features the size of the code will get increase. Due to the debugging becomes a problem to avoid at that problem. We are going to segregate each and every individual as one component and we segregated the code related that separately and we call it as partials.

Partials are return written inside a folder called partials and the file name should start will ‘underscore’.

Example:

_topbar.html

Partials folder should be create inside the templates folder.

To include the partials files in main file we use jinja tag is called **“include**”.

Syntax:

{%include “partials/partial_name” %}

Include a file to load a single section of code within the context of another template

Django include template tag:
Using the include tag in Django requires fewer tags than using the Django extends tag.
(1) First, create the Django template you want to be included in other templates.
(2) Then, add the DTL include tag to the exact location you want the file added in the template

Example:

This is _topbar.html file inside the templates/partials.

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
   header {
       background-color: rgb(201, 76, 76);
       padding: 20px;
       text-align: center;
       font-size: 25px;
       color: white;  }
</style>
</head>
<body>
   <header>
       <h2>pyspider</h2>
     </header>
</body>
</html>
```

This is _navbar.html file inside the templates/partials.

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
*{
   box-sizing: border-box;
  }
  body {
    font-family: Arial, Helvetica, sans-serif;
  }
/* Create two columns/boxes that floats next to each other */
nav {
  float: left;
  width: 30%;
  height: 300px; /* only for demonstration, should be removed */
  background: #ccc;
  padding: 20px;
}
```

```
/* Style the list inside the menu */
nav ul {
  list-style-type: none;
  padding: 0;
}
article {
  float: left;
  padding: 20px;
  width: 70%;
  background-color: #6e59cf;
  height: 300px; /* only for demonstration, should be removed */
}
/* Clear floats after the columns */
section::after {
  content: "";
  display: table;
  clear: both;
}
</style></head>
<body>
<section>
    <nav>
      <ul>
        <li><h1><a href="#">home</a></h1></li>
        <li><h1><a href="#">about</a></h1></li>
        <li><h1><a href="#">gallary</a></h1></li>
      </ul>
    </nav>
    <article>
      <h1>pyspider</h1>
    <p>PySpiders is the world's ace Python development training organization with an aim to bridge the gap bet
ween the demands of the industry and the curriculum of educational institutions.</p>
</article>
  </section>
</body></html>
```

This is _footbar.html file inside the templates/partials.

```
<!DOCTYPE html>
<html lang="en">
<head>
<style>
    footer {
        background-color: rgb(115, 230, 153);
        padding: 10px;
        text-align: center;
        color: white;}
</style></head>
<body>
    <footer>
        <p>thank you </p>
      </footer>
</body></html>
```

This is part.html file inside the templates.

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

```
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>part</title>
</head>
<body>
    {%include "partials/_topbar.html"%}
    {%include "partials/_navbar.html"%}
    {%include "partials/_footbar.html"%}
</body>
</html>
```

This is views.py file inside the project.

```
from django.shortcuts import render

def part(request):
    return render(request,"part.html")
```

This is urls.py file inside the project.

```
from django.contrib import admin
from django.urls import path
from pro6 import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('part/',views.part, name="part" ),
]
```

Search the url inside the browser

```
http://127.0.0.1:8000/part/
```

Static files.

What are Static files?

Static files are those files which cannot be processed, generated or modified by the server. This are the type of files which will not be changed till the end of the project execution because this are the type of files which will created at the time of project creation and will be display to the client before the project execution has the prototype ones after the client confirmation itself we go to the development of the application.

In prototype the styling will be shown the images are we put for the display will be shown will fonts and pop ups related information.

These files don’t change until the developer replaces them with a new one. Thus, the server just fetches, taking a minimum amount of time. Static file management is an important factor in web development. When the website undergoes frequent UI changes and graphics are updated constantly, this condition requires developers to manage static files.

As this we fixed we are going to put it inside the static directory which can be common for the entire project (or) the specific static directory which is unique for each and every app.

Creating a generic static directory:

It is a directory which we created inside the project folder which further contains a directory called css, images, fonts, javascripts file in it.

Once after creating of the directory we need to connect this directory to the project.

Steps to create a static directory to the project:

1) Create a directory called static inside the project directory.
2) Create a variable called STATIC_DIRS and create a path of static directory to it.
   Syntax:
   STATIC_DIRS =os.path.join(BASE_DIR, “static”)

3) Create the list with the name called “STATICFILES_DIRS” and put a path of static directory in it.
   Syntax:
   STATICFILES_DIRS = [STATIC_DIRS,]

4) To check whether the static directory is connected property or not. To place a image inside the imgs directory.
5) Run the server and specify the url like.
   http://127.0.0.1:8000/static/imgs/images_name.extension

note:

Project loads every static files from root directory which we have specified as STATIC_ROOT = path of the root directory.

Creation of specific static directory:

1) Create a directory called static directory inside each app.
2) Create a variable called STATIC_APP_DIRS and put a path of static directory to it.
   Syntax:

   STATIC_APP_DIRS =os.path.join(os.path.join (BASE_DIR, “appname”),’static’)

3) Place the path of the all this directory inside the list called STATICFILES_DIRS. And this type of directory is called as APP specific static directory / child static directory.
   Syntax:
   STATICFILES_DIRS = [STATIC_DIRS, STATIC_APP_DIRS,]

   Example:

   STATIC_ROOT =os.path.join (BASE_DIR, ’static’)

   STATIC_APP_DIRS =os.path.join(os.path.join (BASE_DIR, “appname”),’static’)

   STATIC_URL =’/static/’

STATICFILES_DIRS = [STATIC_APP_DIRS,]

collectstatic command

This command is very useful when the website is in production state. When this command is executed, Django performs these operations:

- It looks for static files in all the directories listed in STATICFILES_DIRS
- The static-files are then copied and saved in STATIC_ROOT directory.
- When the server is requested for static content, it will fetch a file from STATIC_ROOT.
- And, that file will have its URL modified with STATC_URL.

Always the project loads the static content from the root directory (or) generic static directory (or) parent directory.

"Python manage.py collectstatic"

Static templates tags.

Load tag: It load the static resources under *static* directory.

Syntax:

```
{% load static %}
```

{% load static %} tag will inform Django that there are static files in this template. Django will then check for the related static files. Then send them together with the HTML code.

After that, example we gave the src of the image tag. The source of image tag is a Django tag (jinja tag). Here we used static keyword and the path to the image. The path is not specified from the root directory. The STATIC_URL value is automatically added to the source of the image file, even though it is coming from a different directory altogether.

Static tag: Here we used static keyword and the path to the static file links with extension.

Syntax:

```
{% static "path_of_the_template_file.extension"%}
```

Example:

Loading images:

```
<img src= " {%static ' img/imagename.extension ' %}" alt=" " >
```

Loading css file:

```
<link href= " {% static 'css/cssfilename.extension'%}" rel=" " type=" text/css">
```

Loading javascript file:

```
<script src="{%static 'js/jsfilename.extension' %}" type="text/javascript " ></script>
```

Working with Models and Databases:

- As the part of web application development, compulsory we required to interact with database to store our data and to retrieve our stored data.
- It comes with easy yet powerful tools for performing database queries using django.

Database:

- A **database** is an organized collection of data, so that it can be easily accessed and managed. You can organize data into tables, rows, columns, and index it to make it easier to find relevant information. How to interact the database. When you interact the db by using query language(sql).
- SQL (Structured Query Language) is complex and involves a lot of different queries for creating, deleting, updating or any other stuff related to database.
- Django provides a big in-built support for database operations.
- Django provides one inbuilt database like sqlite3.
- For small to medium applications this database is more enough. Django can provide support for other databases also like oracle, mysql, postgre sql etc,

Connection between mysql server to django by traditional way.

- Now we are seen the db base connection with mysql, oracle, others db's.
- We written all things by hard coding the text directly within views. There's a way to insert and retrieve data from the database in a view.
- In this example view, we use the PYMYSQL library to connect to a MySQL database(server), retrieve some records, and feed them to a template for display as a html page:

```
from django.shortcuts import render
import pymysql
def login(request):
        db = pymysql.connect(host='localhost', user='root', passwd='root', db='djnagop')
        cursor = db.cursor()
        val=[venu@gmail.com]
        key="select name from customer where email=%s"
        cursor.execute(key,val)
        names = cursor.fetchall()
        db.commit()
        return render(request, 'book_list.html', {'names': names})
```

- We're hard-coding the database connection parameters. Ideally, these parameters would be stored in the each every view
- We have to write a fair bit of creating a connection, creating a cursor, executing a statement, save the transition, closing the connection. Ideally, all we'd have to do last session. Specify which results we wanted.
- As you might expect, Django's database layer solves these problems.

Let’s start Django’s database layer:

Database Configuration:

- the first configuration that was added to settings.py when we created the django project:
- Django by default provides sqlite3 database(server). If we want to use this database, we are not required to do any configurations.
- The default sqllite3 configurations in settings.py file are declared as follows.

settings.py

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

- If we don't want sqlite3 database 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

Note: Most of the times HOST and PORT are optional.

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.
  C:\user\gvg\djangoprojects\Project> **python manage.py shell**
- When type above command automatically open the python shell in command prompt and after that use below commands for checking purpose.

```
>>> from django.db import connection

>>> c = connection.cursor()
```

- If we are not getting any error means our database configurations are proper.

Installation procedure of mysql

1. double click on the downloaded file and give the permission for the installation by clicking on yes
2. Select "develop default "and click on next.
3. Click on execute so that all the required things will be configured if it asks for visual c++ give permission to install it.
   Note: mysql python connector is not needed and Microsoft visual studio is not needed even if u get they have not installed click on proceed
4. Click on next and click on execute (it takes a bit time for its execution) let it get installed.
5. Post installation click on next until u get post num make sure you keep the port no 3306 and click on next.
6. Configure password for "root" click on next.
   Note: if you want to add the user then click on add user details.
7. Click on and click on execute then click on finish.
8. In the other window do the server setup by clicking on next and next for some time till it asks the root password.
9. Give the root password and click on check if connection is successfully then click on next then execute and finish.
10. Given next and click on finish

Connecting and working with various database management systems

1. All the settings that are used to connect the DBMS will inside the settings.py file of project
   a) Open those file and scroll down to find the section called database.
   b) By default sqlite3 db is connected to our project in settings.
      Note: sqlite3 is a very light weight DB it can used for learning purpose but not for an enterprise application dev.
   c) So that we use MYSQL or POSTGRESQL to work with the Enterprise applications.
   d) To do that we need to download and install either MYSQL or POSTGRESQL in our PC based on our requirements
   e) MYSQL DOWNLOAD LINK https://dev.mysql.com/downloads/installer/
   f) POSTGRESQ1 DOWNLOAD LINK https://www.postgresql.org/download/
   g) To configure mysql with django we need two libraries to be installed
      1. pymysql ==> pip install pymysql
      2. mysql_connector_python ==> pip install mysql_connector_python
   h) To configure postgresql we need a library
      a. 1. psycopg2 ==> pip install psycopg2

2. **Configuration of mysql in django 3.x a.**
    a) Press start button and type **mysql command line client** give the password of **root.**
    b) Create a database using a command CREATE DATABASE DBNAME.
    c) If you find 1 row affected it means database created successfully.
    d) Configuration in 3.x
        1. open settings.py and type the below code import PM. pymysql.version_info = (1, 4, 2, "final", 0)
        2. pymysql.install_as_MySQLdb()
    e) Scroll down to database section and change the current code with below code **settings.py**

```
DATABASES = {
        'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'employeedb',
        'USER': 'root',
        'PASSWORD': 'root'
        ‘HOST’: “localhost”
        ‘port’: ‘3306’
                        }}
```

Checking Configurations:

```
<Env> C:\djangoprojects\Project>py manage.py shell

>>> from django.db import connection

>>> c = connection.cursor()
```

3. **Configuration in django 2.x:**
    1. Install mysql and create a database.
    2. Install pymysql.
    3. Install mysql_connector_python.
    4. Go to env folder and find the file called lib and open that lib file → site_packages→ db→ backends→mysql open it.
    5. Open find base.py file in any editor like (idle or notepad or edit++, vscode) and after find out below code.

        If version < (1, 3, 13):
            Raise improperlyConfigured
    6. Above code replace to below code
        If version < (1, 3, 13):
            pass
    7. Find operations.py file open it and search decode and replace decode with encode save file.
    8. Open settings.py file and write below code.
    9. Got settings.py write a code
        a. Import pymysql

   b. Pymysql.version_info=(1,4,2,’final’,0)
   c. Pymysql.install_as_MySQLdb()

10. Scroll down to find the section called database and replace that code with the below code.

```
DATABASES = {
          'default': {
          'ENGINE': 'django.db.backends.mysql',
          'NAME': 'employeedb',
          'USER': 'root',
          'PASSWORD': 'root'
          ‘HOST’: “localhost”
          ‘port’: ‘3306’
                              }}
```

11. Save the settings.py that we have done.
12. Python manage.py runserver to check the success configuration or not.

Installation of Postgresql

1. Download the postgresql server in this link https://www.postgresql.org/download/.
2. Give the permission for installation.
3. Keep on pressing the next button until it asks the password for the super-user.
4. In postgres name of super user is **postgres.**
5. Keep on pressing next until the install extraction is started.
6. Once the installation is done click on finish, but unselecting stack builder (stack builder is not needed.
7. Click and open the **pgadmin4** app from the start menu and provide the password.
8. Click on server expand postgresql and right click on database click on create choose database.
9. Provide the name for database server and click on save.
10. Install **psycopg2** from command prompt using command **pip install psycopg2**

Settings that we need to do in settings.py in project

1. Go to project and open settings.py file and scroll down the file find out database section and change the database values with the below values.

```
DATABASES = {
          'default': {
          'ENGINE': 'django.db.backends.postgresql',
          'NAME': 'employeedb',
          'USER': 'postgres',
          'PASSWORD': 'root'
          ‘HOST’: “localhost”
          ‘port’: ‘5432’
                              }}
```

Checking Configurations:

```
<Env> C:\djangoprojects\Project>py manage.py shell

>>> from django.db import connection

>>> c = connection.cursor()
```

Defining Models in django:

- The "M" in "MVT" stands for "Model."
- A Django model is a Python code representation of the data in your database.
- The data format in the model is the same as our SQL CREATE TABLE statements.
- It's written in Django rather than SQL, and it goes beyond database column descriptions.
- Django uses a model to run SQL code in the background and return results.
- Rows in your database are represented by Python data structure a table.
- **Models** define the structure of stored data, including the field types and possibly also their maximum size, default values, selection list options, helps text for documentation, label text for forms, etc.
- Django's models provide an *Object-relational Mapping* (ORM) to the underlying database. ORM is a powerful programming technique that makes working with data and relational databases much easier.
- An ORM tool simplifies database programming by providing a simple mapping between an object (the 'O' in ORM) and the database.
- Django's models are written in Python code and provide a mapping to the underlying database structure. Django uses a model to execute SQL behind the scenes to return Python data structures—which Django calls *QuerySets*.
- Finally, Django provides a utility that can create models. This is useful for quickly bringing legacy data up and fast performance.

In the django we call the table as model.

- You all know that our DBMS will understand only one language that is SQL but we will be writing the sql queries here.
- Django will allow us to write the code to the interaction with the dbms only in python but our dbms will not understand the python code to resolve this problem we introduced a concept called SQL parsing.

SQL Parsing:

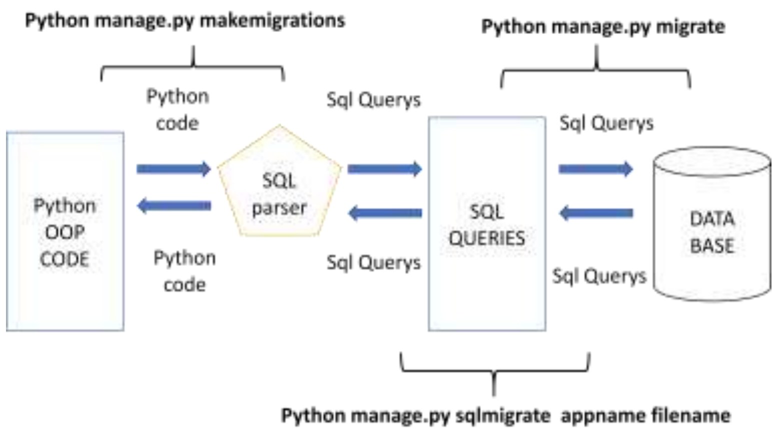
- It is a process of converting the non SQL code (Python OOP Code) to the sql instruction with the help of the parser called SQL parser.
- Sql parser is a translator who will convert our python oop code into sql queries this process is called as migration.

- when we give our python code to the parser it will give me intermediate SQL Queries that need be further given for DBMS Manually.
- Giving the python code to the parser will be done by using the command **python manage.py makemigrations.**
- giving the sql converted queries to the database will be done by the command **python manage.py migrate.**

ORM:

The process of writing the python code to do the interaction with the database is called as ORM (object relational mapping) in this phenomenon we write our python code to interact with the database.

As we are working on ORM we need to follow some conversions
1 table ==> 1 Model or 1 class
1 record or 1 tuple ==> 1 object
Collection of records or tuples ==> query set
According to the rules of the Framework we have to write our oop code to create a model we use a file called models.py
models.py will be there inside the app



Creating models in django:

To use this database layout with Django, steps:

1. We must first create a django application inside the project.

   Django-admin startapp appname

2. After creating application open the application directory than select models.py file. (When creating time automatically generate a models.py file inside the application)
3. open models.py file then create a model class

   For example: **models.py**

```
from django.db import models
class personal(models.Model):
```

```
name = models.CharField(max_length=30)
address = models.CharField(max_length=50)
city = models.CharField(max_length=60)
state = models.CharField(max_length=30)
country = models.CharField(max_length=50)
email = models.EmailField()
phone =models.IntegerField()
```

Model class (python class):

- A Model is a Python class which contains database information.
- A Model is a single, definitive source of information about our data. It contains fields and behavior of the 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)
- Each attribute of the model represents a database table field(column).
- We have to write all model classes inside 'models.py' file.

Converting Model Class into Database specific SQL Code:

Once we write Model class, we have to generate the corresponding SQL Code. For this, we have to use "makemigrations" command.

Syntax:

Python manage.py makemigrations

Output:

```
It results the following:

        Migrations for 'myapp':

        myapp/migrations/0001_initial.py

- Create model Employee
```

How to see corresponding SQL Code of Migrations:

To see the generated SQL Code, we have to use the following command "sqlmigrate"

python manage.py sqlmigrate myapp 0001

```
1) BEGIN;

2) --

3) -- Create model Employee
```

```
4) --

5) CREATE TABLE "myapp_personal" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "name" varchar(30) NOT NULL, "address" varchar(30) NOT NULL, "city" varchar(60) NOT NULL, "state" varchar(30) NOT NULL, "country" varchar(30) NOT NULL, "email" varchar(60) NOT NULL, "phone" int NOT NULL);

6) COMMIT;
```

Note: Here 0001 is the file passed as an argument "id" field:

1) For every table (model), Django will generate a special column named with "id".
2) ID is a Primary Key. (Unique Identifier for every row inside table is considered as a primary key).
3) This field(id) is auto increment field and hence while inserting data, we are not required to provide data for this field.
4) This id field is of type "AutoField"
5) We can override the behavior of "id" field and we can make our own field as "id".
6) Every Field is by default "NOT NULL".

How to execute generated SQL Code (migrate Command):

After generating sql code, we have to execute that sql code to create table in database. For this, we have to use 'migrate' command. As soon as we give below command all the tables that are predefined in the application will be migrated to our database internally the sql queries will get triggered and executed to do the changes in the database.

Syntax:

python manage.py migrate

Output:

```
Operations to perform:
Apply all migrations: admin, auth, contenttypes, sessions, testApp
Running migrations:
     Applying contenttypes.0001_initial... OK
     Applying auth.0001_initial... OK
     Applying admin.0001_initial... OK
     Applyingadmin.0002_logentry_remove_auto_add... 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... OK
     Applying auth.0008_alter_user_username_max_length... OK
     Applying sessions.0001_initial... OK
     Applying testApp.0001_initial... OK
Note: Now tables will be created in the database.
```

The Advantage of creating Tables with 'migrate' Command

If we use 'migrate' command, then all Django required tables will be created in addition to our application specific tables. If we create table manually with sql code, then only our application specific table will be created and django may not work properly. Hence it is highly recommended to create tables with 'migrate' command.

Difference between makemigrations and migrate:

"makemigrations" is responsible to generate SQL code for Python model class whereas "migrate" is responsible to execute that SQL code so that tables will be created in the database.

Basic model data types fields list:

Field Name	**Description**
AutoField	It An IntegerField that automatically increments.
BinaryField	A field to store raw binary data.
BooleanField	A true/false field. The default form widget for this field is a CheckboxInput.
CharField	A field to store text based values.
DateField	A date, represented in Python by a datetime. date instance it is used for date and time, represented in Python by a datetime.datetime instance.
EmailField	It is a CharField that checks that the value is a valid email address.
FileField	It is a file-upload field.
FloatField	It is a floating-point number represented in Python by a float instance.
ImageField	It inherits all attributes and methods from FileField, but also validates that the uploaded object is a valid image.
IntegerField	It is an integer field. Values from -2147483648 to 2147483647 are safe in all databases supported by

	Django.
TextField	A large text field. The default form widget for this field is a Textarea.
TimeField	A time, represented in Python by a datetime.time instance.
URLField	A CharField for a URL, validated by URLValidator.

Relationship Fields

Django also defines a set of fields that represent relations.

Field Name	**Description**
ForeignKey	A many-to-one relationship. Requires two positional arguments: the class to which the model is related and the on_delete option.
ManyToManyField	A many-to-many relationship. Requires a positional argument: the class to which the model is related, which works exactly the same as it does for ForeignKey, including recursive and lazy relationships.
OneToOneField	A one-to-one relationship. Conceptually, this is similar to a ForeignKey with unique=True, but the “reverse” side of the relation will directly return a single object.

Example:

1) **Create a models in models.py file**

```
class Publisher(models.Model):
        name = models.CharField(max_length=30)
        address = models.CharField(max_length=50)
        city = models.CharField(max_length=60)
        state_province = models.CharField(max_length=30)
        country = models.CharField(max_length=50)
        website = models.URLField()

class Author(models.Model):
        first_name = models.CharField(max_length=30)
        last_name = models.CharField(max_length=40)
        email = models.EmailField(max_length=50, unique =True)

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

```
authors = models.ManyToManyField(Author)
publisher = models.ForeignKey(Publisher)
publication_date = models.DateField()
```

2) python manage.py makemigrations books

```
Migrations for 'books':
0001_initial.py: -
        --Create model Author
        --Create model Book –
        --Create model Publisher –
        --Add field publisher to book
```

3) Python manage.py sqlmigrate myapp 0001

```
Result on above command

BEGIN;
CREATE TABLE "books_author" (
        "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
        "first_name" varchar(30) NOT NULL,
        "last_name" varchar(40) NOT NULL,
        "email" varchar(254) NOT NULL, UNIQUE      );

CREATE TABLE "books_book" (
        "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
        "title" varchar(100) NOT NULL,
        "publication_date" date NOT NULL           );

CREATE TABLE "books_book_authors" (
        "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
        "book_id" integer NOT NULL REFERENCES
        "books_book" ("id"), "author_id" integer NOT NULL REFERENCES
"books_author" ("id"), UNIQUE ("book_id", "author_id")           );

CREATE TABLE "books_publisher" (
        "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
        "name" varchar(30) NOT NULL,
        "address" varchar(50) NOT NULL,
        "city" varchar(60) NOT NULL,
        "state_province" varchar(30) NOT NULL,
        "country" varchar(50) NOT NULL,
        "website" varchar(200) NOT NULL            );

CREATE TABLE "books_book__new" (
        "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
        "title" varchar(100) NOT NULL,
        "publication_date" date NOT NULL,
        "publisher_id" integer NOT NULL REFERENCES "books_publisher" ("id") );
```

4) Python manage.py migrate

```
Operations to perform:
Apply all migrations:
```

```
books Running migrations:
Rendering model states...
DONE # ...
 Applying books.0001_initial...
OK # ...
```

Basic Data Access:

Once you’ve created a model, Django automatically provides a high-level Python API for working with those models.

Insert the values: To insert a row into your database, first create an instance of your model using keyword arguments, like so:

1) Try it out by running **python manage.py shell** and typing the following.
   Syntax: **var=modelname(key=value, key=value,--------,key=value)**
   Example:

```
>>> from books.models import Publisher
>>> p1 = Publisher(    name='django',
            address='basavanagudi',
            city='bangalore',
            state_province=’karnataka',
            country='india',
            website='http://www.pyspiders.com/')

>>> p1.save()
>>>p2 = Publisher(     name="python",
            address='basavanagudi',
            city='bangalore',
            state_province='karnataka',
            country='india',
            website='http://www.pyspiders.com/')

>>> p2.save()
 >>> publisher_list = Publisher.objects.all()
 >>> publisher_list
[<publisher: publisher object(1)> , [<publisher: publisher object(2)>]
```

2) If you want to create an object and save it to the database in a single step, use the **objects.create()** method. This example is equivalent to the example above.
   Syntax: **var=modelname.objects.create(key=value, key=value,--------,key=value)**
   Example:

```
>>> p1 = Publisher.objects.create(name='django',
            address='basavanagudi',
            city='bangalore',
            state_province=’karnataka',
            country='india',
            website='http://www.pyspiders.com/')

>>> p2 = Publisher.objects.create(name="python",
            address='basavanagudi',
            city='bangalore',
            state_province='karnataka',
            country='india',
            website='http://www.pyspiders.com/')
     >>> publisher_list = Publisher.objects.all()
>>> publisher_list
[<publisher: publisher object(1)> , <publisher: publisher object(2)>]
```

Note:

Until you call save, the record isn't saved in the database ()
In SQL, this can roughly be translated into the following:

INSERT INTO books_publisher (name, address, city, state_province, country, website) VALUES ('python', 'basavanagudi', 'banagolre', 'karnataka', 'india', 'http://www.pyspiders.com/');

3) To Add Multiple Records at a Time:
By using bulk_create() method.
Example:

```
        Employee.objects.bulk_create([
        Employee(eno=1,ename='DDD',esal=1000,eaddr='Hyd'),
        Employee(eno=2,ename='HHH',esal=1000,eaddr='Hyd'),
        Employee(eno=3,ename='MMM',esal=1000,eaddr='Hyd')])
```

4) get_or_create():

- it is method which is used to get the details of the objects which is present if it is not present then it will create it and gets the details it can be used instead of get method because if it is present it gets the object.
- This method is used almost situation to avoid error from create.
- It returns a tuple containing 2 values first one is object and second one is creation status.
- If creation status is true then object is created now else object is already is created

Syntax:
var=modelname.objects.get_or_create(key=value, key=value,.....,key=value)

- After creation we save using savefunction

Syntax:

Varname[0].save{}

Example:

```
def publish(request):
        p2 = Publisher.objects.get_or_create(name="python",
        address='basavanagudi',
        city='bangalore',
        state_province='karnataka',
        country='india',
        website='http://www.pyspiders.com/')

        if p2[1]==True:
                    p2[0].save()
                    return HttpResponse("creation successful")
        return HttpResponse("already creation successful")
```

5) **update_or_create():**
   - It is method which is used to update or create the records if the specified key is matched than it updated the existing key with the specified value else it create it.
   - Like get_or_create method it will also return tuple of arguments.
   - It returns a tuple containing 2 values first one is object and second one is creation status.
   - If creation status is true then object is created now else object is already is created

   Syntax: for creation syntax is

   var=modelname.objects.update_or_create(key=value, ..., key=value)

   Syntax: for updation syntax is

   var=modelname.objects.update_or_create(key=value, ..., key=value, defaults={key:newvalue, ... , key:newvalue})

   - After creation we save using savefunction

   Syntax:

   Varname[0].save{}

Example: for creation

```
def publish(request):
            p2 = Publisher.objects.update_or_create(name="python",
            address='basavanagudi',
            city='bangalore',
            state_province='karnataka',
            country='india',
            website='http://www.pyspiders.com/',)
            p2[0].save()
            if p2[1]==True:
                        return HttpResponse("creation successful")
            return HttpResponse("already creation successful")
```

Example: for updation

```
def publish(request):
        p2 = Publisher.objects.update_or_create(name="python",
        defaults={'name'='full python'})
        p2[0].save()
        if p2[1]==True:
                return HttpResponse("creation successful")
        return HttpResponse("already creation successful")
```

Note: after creation we save using save() function.

Syntax: var.save()

Retrieving the data: (Selecting Objects)

Retrieving the all the records

```
>>> Publisher.objects.all()

[<publisher: publisher object(1)> ,  <publisher: publisher object(2)>]
```

This roughly translates to this SQL:

SELECT id, name, address, city, state_province, country, website FROM books_publisher;

Filtering Data: (filter the objects data)

In the Django API, you can filter your data using the filter() method:

- filter() accepts keyword arguments, which are then converted into SQL WHERE clauses.

```
>>> Publisher.objects.filter(name='python')
[<publisher: python]
```

This roughly translates to this SQL:
SELECT id, name, address, city, state_province, country, website FROM books_publisher WHERE name = 'Apress'

- You can pass multiple arguments into filter() .

```
>>> Publisher.objects.filter(country="india", state_province="karnataka")
 [<publisher: django> ,  <publisher: python>]
```

This roughly translates to this SQL:
SELECT id, name, address, city, state_province, country, website FROM books_publisher WHERE country = 'U.S.A.' AND state_province = 'CA';

- Those multiple arguments get translated into SQL AND clauses.
- Filter using like operations

```
>>> Publisher.objects.filter(name__contains="p")
[<publisher: python>]
```

This roughly translates to this SQL:
SELECT id, name, address, city, state_province, country, website FROM books_publisher WHERE name LIKE '%p%';

- Between name and contains, there's a double underscore. Django, like Python, employs the double underscore to indicate that something "magical" is happening - in this case, the __contains section is translated into a SQL LIKE statement by Django

Retrieving Single Objects:

- It's sometimes more convenient to retrieve a single object rather than a collection. That's why there's a get() method:

```
>>> Publisher.objects.get(name="Apress")
[<publisher: python>]
```

- Note: Only a single item is returned rather than a list (rather, QuerySet). As a result, if a query returns several objects, an exception will be raised:

Updating Multiple objects(data) in One Statement:

- You might simply want to update a subset of the columns. For example, suppose we want to modify the name of the python Publisher from 'python' to 'python full stack' It would look like this if you used save():

```
>>> p = Publisher.objects.get(name='python')
 >>> p.name = 'python full stack'
>>> p.save()
```

This roughly translates to the following SQL:
SELECT id, name, address, city, state_province, country, website FROM books_publisher WHERE name = 'python';

```
 UPDATE books_publisher SET name = 'python full stack',
                        address = 'basavanagudi',
                         city = 'banaglore',
                        state_province = ‘karnataka',
                         country = 'india',
                         website = 'http://www.pyspiders.com'
                        WHERE id = 01;
```

- It's better to only change the columns that need to be changed. Use the update() method on QuerySet objects to achieve this. Here's an example.

```
>>> Publisher.objects.filter(id=01).update(name='python full stack')
```

This roughly translates to the following SQL:

UPDATE books_publisher SET name = 'python full stack' WHERE id = 01;

- The update() method can be used on any QuerySet, allowing you to alter multiple records at once. Here's how to modify the country in each Publisher record from 'india' to Bharath:

```
>>> Publisher.objects.all().update(country='Bharath')
2
```

Deleting the records (Objects):

- To delete an object from your database, simply call the object's delete() method:

```
>>> p = Publisher.objects.get(name="django")
>>> p.delete()
>>> Publisher.objects.all()
[publisher:<python>]
```

- You can also delete objects in bulk by calling delete() on the result of any QuerySet. This is similar to the update() method we showed in the last section:

```
>>> Publisher.objects.filter(country='india').delete()
```

- You can delete all the records at a time.

```
>>> Publisher.objects.all().delete() >>> Publisher.objects.all()
[]
```

Django ORM:

- ORM full form **Object Relation Mapping**

⇨ In general we can retrieve data from the database by using the following approach Employee.objects.all()

The return type of all() method is QuerySet.

```
>>>qs=Employee.objects.all()

>>>print(type(qs))

<class 'django.db.models.query.QuerySet'>
```

⇨ If we want to get only one record then we can use get() method.
`The return type of get() method is Employee Object.

```
>>>emp = Employee.objects.get(id=1)

>>>print(type(emp)) #<class 'testapp.models.Employee'>
```

How to find Query associated with QuerySet

qs = Employee.objects.all()

print(qs.query)

How to select only some columns in the queryset (3 Ways)

1) By using values_list()

q1 = Student.objects.all().values_list('name','mailid','aadharnumber')

2) By using values()

q1 = Student.objects.all().values('name','mailid','aadharnumber')

3) By using only():

q1 = Student.objects.all().only('name','mailid','aadharnumber')

Note: Difference between values() and only() Methods

In the case of values() only specified columns will be selected. But in the case of only() in addition to specified columns 'id' column also will be selected.

How to Filter Records based on some Condition

- **Greater than**

  employees = Employee.objects.filter(esal__gt=15000)

  It returns all employees whose salary greater than 15000

- **Greater than or equal to**

  employees = Employee.objects.filter(esal__gte=15000)

  It returns all employees whose salary greater than or equal to 15000

- Similarly we can use __lt and __lte

  Example: **Relational <, >, <=, >=**

Sql querys	Model language
Select * from customer where id<4:	res=Customer.objects.filter(id__lt=7)
Select * from customer where id<=4:	res=Customer.objects.filter(id__lte=4)
Select * from customer where id>4:	res=Customer.objects.filter(id__gt=4)
Select * from customer where id>=4:	res=Customer.objects.filter(id__gte=4)

How to Various possible Field Look ups ():

1) **exact**: Exact Match

student.objects.get(id__exact=14)

2) **iexact**: Case-insensitive exact Match

student.objects.get(name__iexact='venu') student.objects.get(name__iexact=None)

3) **contains**: Case-sensitive Containment Test

student.objects.get(name__contains='sam')

4) **icontains**: Case-insensitive Containment Test

student.objects.get(name__icontains='venu')

5) **startswith** :Case-sensitive starts-with.

student.objects.filter(name__startswith=vgopal)

6) **istartswith**: Case-insensitive starts-with.

student.objects.filter(name__istartswith='Venugopal')

7) **endswith**: Case-sensitive ends-with.

student.objects.filter(name__endswith='sam')

8) **iendswith:** Case-insensitive ends-with.

student.objects.filter(name__iendswith='Loin')

Example: **Special operators (like operator)**

Sql query	**Model language**
Select * from customer where name like '%a%'	**res= Customer.objects.filter(name__contains='a')**
Select * from customer where name like 'v%'	**res= Customer.objects.filter(name__startswith='v')**
Select * from customer where name like '%v'	**res= Customer.objects.filter(name__endswith='k')**
Select * from customer where name not like '%a%'	**res= Customer.objects.exclude(name__contains='a')**
Select * from customer where name not like 'v%'	**res= Customer.objects.exclude(name__startswith='v')**
Select * from customer where name not like '%v'	**res= Customer.objects.exclude(name__endswith='k')**
Case insensitive:	**res= Customer.objects.filter(name__icontains='a')**
Select * from customer where name not like 'v%'	**res= Customer.objects.exclude(name__istartswith='v')**
Select * from customer where name like '%v'	**res= Customer.objects.filter(name__iendswith='k')**

9) **in:** In a given iterable; often a List, Tuple OR queryset.

student.objects.filter(id__in=[1, 3, 4])

student.objects.filter(name__in='sai')

10) **range**: Range test (inclusive)

student.objects.filter(emarks__range=(76,90)

Example

import datetime
start_date = datetime.date(2005, 1, 1)
end_date = datetime.date(2005, 3, 31)
student.objects.filter(joindate=(start_date, end_date))

How to implement logical operators in Django ORM

- **Logical OR operator (2 ways)**

1) queryset_1 | queryset_2
2) filter(Q(condition1)|Q(condition2))

Example: To get all employees whose name starts with 'A' OR salary < 15000.
Method1:
employees = Employee.objects.filter(ename__startswith='A') |
Employee.objects.filter(esal__lt=15000)

Method2:
from django.db.models import Q
employees= Employee.objects.filter(Q(ename__startswith='A') |
Q(esal__lt=15000))2 ways are available

- **Logical AND operator (3 ways)**

1) filter(condition1,condition2)
2) queryset_1 & queryset_2
3) filter(Q(condition_1)&Q(condition_2))

Example: To get all employees whose name startswith 'J' AND salary < 15000
Method 1:
employees= Employee.objects.filter(ename__startswith='J '**,** esal__lt=15000)

Method 2:
employees= Employee.objects.filter(ename__startswith='J') &

```
Employee.objects.filter(esal__lt=15000)
```

Method 3:

```
from django.db.models import Q
employees= Employee.objects.filter(Q(ename__startswith='J') &
 Q(esal__lt=15000))
```

- **Logical NOT operators (2 ways)**
  1) exclude(condition)
  2) filter(~Q(condition))

Example: To select all employees whose name not starts with 'J':

Method1:

```
employees= Employee.objects.exclude(ename__startswith='J')
```

Method2:

```
from django.db.models import Q
employees= Employee.objects.filter(~Q(ename__startswith='J'))
```

How to do Aggregate Functions in django ORM

- Django ORM defines several functions to perform aggregate operations.

1) **Avg(),**

```
avg1=Employee.objects.all().aggregate(Avg('esal'))
```

2) **Max(),**

```
max=Employee.objects.all().aggregate(Max('esal'))
```

3) **Min(),**

```
min=Employee.objects.all().aggregate(Min('esal'))
```

4) **Sum(),**

```
sum=Employee.objects.all().aggregate(Sum('esal'))
```

5) **Count()**

```
count=Employee.objects.all().aggregate(Count('esal'))
```

How to Order query set in Sorting Order:

- All records will be arranged according to ascending order of eno
- Default sorting order is ascending order

```
employees = Employee.objects.all().order_by('eno')
```

- For Descending order we have to use '-'

```
employees = Employee.objects.all().order_by('-eno')
employees = Employee.objects.all().order_by('-esal')[0]
```

- Returns highest salary employee object

```
employees = Employee.objects.all().order_by('-esal')[1]
```

- Returns Second highest salary employee object

```
employees = Employee.objects.all().order_by('-esal')[0:3]
```

- Returns list of top 3 highest salary employees info but in the case of strings for alphabetical order:

```
employees = Employee.objects.all().order_by('ename')
```

- In this case, case will be considered.
- If we want to ignore case then we should use Lower() Function

```
from django.db.models.functions import Lower

employees=Employee.objects.all().order_by(Lower('ename'))
```

Model Inheritance

- It is very useful and powerful feature of django.
- There are 5 types of Model Inheritance.
  1) Abstract Base Class Model Inheritance
  2) Multi table inheritance
  3) Multi level inheritance
  3) Multiple inheritance
  5) Proxy model inheritance:

1) Abstract Base Class Model Inheritance:

- If several model classes having common fields, then it is not recommended to write these fields separately in every model class. It increases length of the code and reduces readability.
- We can separate these common fields into another Model class, which is also known as Base Class. If we extend Base class automatically common fields will be inherited to the child classes.

Example:

```
class Contact(models.Model):
        name=models.CharField(max_length=64)
        email=models.EmailField()
        address=models.CharField(max_length=256)
        class Meta:
                abstract=True

class Student(Contact):
        rollno=models.IntegerField()
        marks=models.IntegerField()

class Teacher(Contact):
        subject=models.CharField(max_length=64)
        salary=models.FloatField()
```

- In this case only Student and Teacher tables will be created which includes all the fields of Contact.
- Note: Contact class is an abstract class and hence table won't be created.It is not possible to register abstract model classes to the admin interface. If we are trying to do then we will get error.

2) Multi Table Inheritance:

- If the base class is not abstract then such type of inheritance is called multi table inheritance. In Multi table inheritance, inside database tables will be created for both Parent and Child classes.
- Multi table inheritance uses an implicit OneToOneField to link Parent and Child. That is by using one-to-one relationship multi table inheritance is internally implemented.
- Django hides internal structure and creates feeling that both tables are independent.

Example:

```
class Base(models.Model):
        name=models.CharField(max_length=64)
        phone=models.IntegerField()
        address=models.CharField(max_length=64)

class Child(BasicModel):
        aadhar=models.IntegerField()
        email=models.EmailField(max_length=64)
```

3) Multi Level Inheritance:

- Inheritance at multiple levels.

Example:

```
class Person(models.Model):
        name=models.CharField(max_length=64)
        age=models.IntegerField()

class Employee(Person):
        eno=models.IntegerField()
        esal=models.FloatField()

class Manager(Employee):
        exp=models.IntegerField()
        team_size=models.IntegerField()
```

- Note: Multilevel inheritance internally multi table inheritance only.

4) Multiple Inheritance:

- If model class extends multiple parent classes simultaneously then such type of inheritance is called Multiple Inheritance.

Example:

```
class Parent1(models.Model):
        f1=models.CharField(max_length=64)
        f2=models.CharField(max_length=64)
class Parent2(models.Model):
        f3=models.CharField(max_length=64)
        f4=models.CharField(max_length=64)
class Child(Parent1,Parent2):
        f5=models.CharField(max_length=64)
        f6=models.CharField(max_length=64)
```

- **Note:**

  1. Multiple inheritances are also internally multi table inheritance only.
  2. In multiple inheritances Parent classes should not contain any common field, otherwise we will get error.

5) Proxy Model Inheritance:

- For the same Model we can provide a customized view without touching the database is possible by using Proxy Model Inheritance.
- In this inheritance a seperate new table won't be created and the new model also pointing to the same old table.

  Example:

```
class Employee(models.Model):
        fields
class ProxyEmployee(Employee):
        class Meta:
        proxy=True
```

- Both Employee and ProxyEmployee are pointing to the same table only.

- In the admin interface if we add a new record to either Employee or ProxyEmployee, then automatically those changes will be reflected to other model view.

Admin Panel:

- Django has its own built-in admin panel.
- Which we can use after the migrate command to login to the admin panel.
- We need to create the superuser (admin).
- We need to create the login credentials for the admin for that we use the command **python manage.py createsuperuser** and enter.
- It will ask for username, email, password and reenter the password and press enter once we get the message called superuser *created successfully.*
- Then run the server and go to the **url ipaddress:portnum/admin** it will take you to the built in admin site
  Ex: 127.0.0.1:8000/admin

Creating the user from the admin panel:

1. Login to the admin panel and click on Users and click on add user.
2. Fill your information and click on save.
3. If the information is valid then user will be saved and a new form will be displayed fill out the details in the form.
4. Click on staff status and superuser checkbox these will give the admin rights.
5. Save it and logout from the current user and login as with the new user information.

Query set:

A **QuerySet** represents a collection of objects from your database. It can have zero, one or many filters. Filters narrow down the query results based on the given parameters. In SQL terms, a **QuerySet** equates to a SELECT statement, and a filter is a limiting clause such as WHERE or LIMIT .

Model Manager:

The manager is the interface that interacts with the database in a Django model. For example, if you want to get objects from your database, you'll need to use a Manager on your model class to create a QuerySet.

Every model in a Django application has at least one Manager, and objects is the default manager for every model, retrieving all objects from the database.

However, one model can have several model managers, and we can extend the base manager class to create our own custom model managers.

How to define our own Custom Manager:

- We have to write child class for models.Manager.
- Whenever we are using all() method, internally it will call get_queryset() method.
- To customize behavior we have to override this method in our Custom Manager class.

Example:
To retrieve all employees data according to ascending order of ename, we have to define **Custom Manager class as follows models.py**

```
from django.db import models
class CustomManager(models.Manager):
        def get_queryset(self):
                return super().get_queryset().order_by('ename')

# Create your models here.
class Employee(models.Model):
        eno=models.IntegerField()
        ename=models.CharField(max_length=64)
        esal=models.FloatField()
        eaddr=models.CharField(max_length=256)
        emply=CustomManager()
```

- As we know objects is the default model manager for every model, therefore **employee.objects.all()** will return all post objects. The objects method is capable of doing all basic QuerySets then.
- Whenever we are using employee.emply.all() method it will always get employees in ascending order of ename based on our requirement. we can define our own new methods also inside Custom Manager class.

Example: for custom manager class

```
class CustomManager(models.Manager):
        def get_queryset(self):
                return super().get_queryset().order_by('email')

        def get_emp_sal_range(self,esal1,esal2):
                return super().get_queryset().filter(esal__range=(esal1,esal2))

        def get_employees_sorted_by(self,param):
                  return super().get_queryset().order_by(param)
```

Why would we need a custom model manager?
There are two reasons you might want to customize a Manager –

- To add extra Manager methods.
- To modify the initial QuerySet the Manager returns.

We will build a custom model manager called CustomManager to retrieve the employee objects.

Django ORM:
The **Django** web framework includes a default object-relational mapping layer (**ORM**) that can be used to interact with application data from various relational databases such as SQLite, PostgreSQL and MySQL. The **Django ORM** is an implementation of the object-relational mapping (**ORM**) concept.

Django Forms:

- It is the very important concept in web development.
- The main purpose of forms is to take user input.
  Eg: login form, registration form, enquiry form etc.
- From the forms we can read end user provided input data and we can use that data based on requirement. We may store in the database for future purpose. We may use just for validation/authentication purpose etc.
- Here we have to use Django specific forms but not HTML forms.

Advantages of Django Forms over HTML Forms:

1) We can develop forms very easily with python code.
2) We can generate HTML Form widgets/components (like textarea, email, pwd etc) veryquickly.
3) Validating data will become very easy.
4) Processing data into python data structures like list, set etc will become easy
5) Creation of Models based on forms will become easy etc.

Request method:
Whenever we do any operations on the web –application the response for those operations in a website should come from server to get the response out browser should send some request.
The request will be of 2 types:

1. GET request:
   - when ever we wanted any information from the server then use GET request.
   - By default all the request type which comes from the browser will be GET request only.
   - And the data
2. POST request:
   - When ever we wanted to store any data in thnesrver or when ever we wanted to send any data tothe server in that case we should specify the request type as POST request.
   - Until we specify the request type will be the get request only whenever we wanted to have the post request in that case we should specify the method attribute in the with value as POST.
     <form method="POST">
   - When ever we wanted to store the data inside the server first we should have the clear data but whenever we use post request there is chance that the hackers might corrupt out data through the cross site request and to avoid to it we should use a token called csrf_token within the jinja tag immediately after the form tag{%csrf_token%}

- If you don’t specify then the form submission will not happen instead you will get a error message called CSRF verification failed.
- The data that comes from the front end with request will be stored inside the request. POST in the form of a dictionary in that the keys willbe the name of the element and value will be tha data that we have sent.
- If the element has the single value to get that we us the get method from request.POST
- Syntax: request.POST.get(“name of the element”)
- This scene
- If the lement carries morethan 1 value then we should use get_list method to get the list of values.
- Syntax: request.POST.get

Process to generate Django Forms:

Step-1: Creation of forms.py file in 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 and marks are the field names which will be available in html form

Step-2: usage of forms.py inside views.py file:

views.py file is responsible to send this form to the template html file

views.py:

```
from django.shortcuts import render
from user import forms
# Create your views here.
def studentinputview(request):
        form=forms.StudentForm()
        my_dict={'form':form}
        return render(request,'testapp/input.html',context=my_dict)
```

Note: context parameter is optional.We can pass context parameter value directly without using keyword name 'context'

Step-3: Creation of html file to hold form:

- Inside template file we have to use template tag to inject form {{form}}
- It will add only form fields. But there is no <form> tag and no submit button.
- Even the fields are not arranged properly. It is ugly form.

We can make proper form as follows

```
<h1>Registration Form</h1>
<div class="container" align="center">
<form method="post">
        {{form.as_p}}
        <input type="submit" class="btn btn-primary" name=""
value="Submit">
</form>
</div>
```

Note: If we submit this form we will get 403 status code response Forbidden (403)
CSRF verification failed. Request aborted.
Help Reason given for failure:
    CSRF token missing or incorrect.

CSRF (Cross Site Request Forgery) Token:

- Every form should satisfy CSRF (Cross Site Request Forgery) Verification, otherwise Django won't accept our form.
- It is meant for website security. Being a programmer we are not required to worry anything about this. Django will takes care everything.
- But we have to add csrf_token in our form.

```
<h1>Registration Form</h1>
<div class="container" align="center">
<form method="post">{% csrf_token %}
{{form.as_p}}
<input type="submit" class="btn btn-primary" name="" value="Submit">
</form>
</div>
```

- If we add csrf_token then in the generate form the following hidded field will be added,which makes our post request secure.

```
<input type='hidden' name='csrfmiddlewaretoken'
value='1ZqIJJqTLMVa6RFAyPJh7pwzyFmdiHzytLxJIDzAkKULJz4qHcetL
oKEsRLwyz4h'/>
```

- The value of this hidden field is keep on changing from request to request. Hence it is impossible to forgery of our request.
- If we configured csrf_token in html form then only django will accept our form.

Output form fields: {{form.as_table}}, {{form.as_p}}, {{form.as_ul}} :

- Django forms offer three helper methods to simplify the output of all form fields.
- The syntax {{form.as_table}} outputs a form's fields to accommodate an HTML <table> tag.

Django form output with form.as_table

```
<tr>
   <th><label for="id_name">Name:</label></th>
   <td><input id="id_name" name="name" type="text" /></td>
</tr>\n
<tr>
   <th><label for="id_email">Your email:</label></th>
   <td><input id="id_email" name="email" type="email" required/></td>
</tr>\n
```

```
<tr>
   <th><label for="id_comment">Comment:</label></th>
   <td><textarea cols="40" id="id_comment" name="comment" rows="10"
required>\r\n</textarea></td>
</tr>
```

- The syntax {{form.as_p}} outputs a form's fields with HTML <p> tag.

Django form output with form.as_p

```
<p>
   <label for="id_name">Name:</label>
   <input id="id_name" name="name" type="text" />
</p>\n
<p>
   <label for="id_email">Your email:</label>
   <input id="id_email" name="email" type="email" required/>
</p>\n
<p>
   <label for="id_comment">Comment:</label>
   <textarea cols="40" id="id_comment" name="comment" rows="10"
required>\r\n</textarea>
</p>'
```

- The syntax {{form.as_ul}} outputs a form's fields to accommodate an HTML <ul> list tag.

Django form output with form.as_ul

```
<li>
   <label for="id_name">Name:</label>
   <input id="id_name" name="name" type="text" />
</li>\n
<li>
   <label for="id_email">Your email:</label>
   <input id="id_email" name="email" type="email" required/>
</li>\n
   <li><label for="id_comment">Comment:</label>
   <textarea cols="40" id="id_comment" name="comment" rows="10"
required>\r\n</textarea>
</li>
```

Note:

If you use form.as_table, form.as_p, form.as_ul you must declare opening/closing HTML tags, a wrapping <form> tag, a Django {% csrf_token %} tag and an <input type="submit"> button.

Django form field types, generated HTML, default widget and validation behavior

Django form field type	HTML output	Default Django widget	Validation behavior
forms.BooleanField()	<input type='checkbox'	forms.widgets.Checkb	Generates HTML checkbox input

	...>	oxInput()	markup to obtain a boolean True or False value; returns False when the checkbox is unchecked, True when the checkbox is checked.
forms.NullBooleanField()	`<select> <option value="1" selected="selected"> Unknown </option> <option value="2"> Yes </option> <option value="3"> No </option> </select>`	forms.widgets.NullBooleanSelect()	Works just like BooleanField but also allows "Unknown" value; returns None when the Unknown(1) value is selected, True when the Yes(2) value is selected and False when the No(3) value is selected.
forms.CharField()	`<input type="text" ...>`	forms.widgets.TextInput()	Generates HTML text input markup.
forms.EmailField()	`<input type="email" ...>`	forms.widgets.EmailInput()	Generates HTML email input markup. Note this HTML5 markup is for client-side email validation and only works if a browser supports HTML5. If a browser doesn't support HTML5, then it treats this markup as a regular text input. Django server-side form validation is done for email irrespective of HTML5 support.
forms.GenericIPAddressField()	`<input type="text" ...>`	forms.widgets.TextInput()	Works just like CharField, but server-side Django validates the (text) value can be converted to an IPv4 or IPv6 address (e.g.192.46.3.2, 2001:0db8:85a3:0000:0000:8a2e:0370:7334).
forms.RegexField(regex='regular_expression')	`<input type="text" ...>`	forms.widgets.TextInput()	Works just like CharField, but server-side Django validates the (text) value complies with the regular expression defined in regex. Note regex can be either a string that represents a regular expression (e.g. \.com$ for a string that ends in .com) or a compiled Python regular expression from Python's re package (e.g. re.compile('\.com$'))
forms.SlugField()	`<input type="text" ...>`	forms.widgets.TextInput()	Works just like CharField, but server-side Django validates the (text) value can be converted to slug. In Django a 'slug' is a value that contains only lower case letters, numbers, underscores and hyphens, which is typically used to sanitize URLs and file names (e.g. the slug representation of 'What is a Slug ?! A sanitized-string' is what-is-a-slug-a-sanitized-string.
forms.URLField()	`<input type="url" ...>`	forms.widgets.URLInput()	Generates HTML url input markup. Note this HTML5 markup is for client-side url validation and only works if the browser supports HTML5. If a browser doesn't support HTML5 then it treats this markup as regular text input. Django server-side form validation is done for a

			url irrespective of HTML5 support.
forms.UUIDField()	<input type="text" ...>	forms.widgets.TextInput()	Works just like CharField, but server-side Django validates the (text) value is convertable to a UUID (Universally unique identifier).
forms.ComboField(fields=[field_type1,field_type1])	<input type="text" ...>	forms.widgets.TextInput()	Works just like CharField, but server-side Django enforces the data pass rules for a list of Django form fields (e.g.ComboField(fields=[CharField(max_length=50), SlugField()]) enforces the data be a slug field with a maximum length of 50 characters)
forms.MultiValueField(fields=[field_type1,field_type1])	*Varies depending on field list (e.g. for three CharField : <input type="text" ...><input type="text" ...><input type="text" ...>; for two CharField: <input type="text" ...><input type="text" ...>*	forms.widgets.TextInput()	Designed to create custom form fields made up of multiple pre-exisiting form fields (e.g. Social Security form field made up of three CharField()). It requires a subclass implementation (e.g. class SocialSecurityField(MultiValueField):) to include the base form fields and validation logic of the new field.
forms.FilePathField(path='directory')	<select > <option value="directory/file_1"> file_1 </option> <option value="directory/file_2"> file_2 </option> <option value="directory/file_3"> file_3 </option> </select>	forms.widgets.Select()	Generates an HTML select list from files located on a server-side path directory. Note value is composed of path+filename and just displays filename
forms.FileField()	<input type="file" ...>	forms.widgets.ClearableFileInput()	Generates HTML file input markup so an end user is able to select a file through his web browser. In addition, it provides various utilities to handle post-processing of files
forms.ImageField()	<input type="file" ...>	forms.widgets.ClearableFileInput()	Generates HTML file input markup so an end user is able to select an image file through his web browser. Works just like FileField but provides additional utilities to handle post-processing of images using the Pillow package. Note this field forces you to install Pillow (e.g. pip install Pillow).
forms.DateField()	<input type="text" ...>	forms.widgets.DateInput()	Works just like CharField, but server-side Django validates the (text) value can be converted to a datetime.date, datetime.datetime or string formatted in a particular date format (e.g. 2017-12-25, 11/25/17).
forms.TimeField()	<input type="text" ...>	forms.widgets.TextInput()	Works just like CharField, but server-side Django validates the (text) value

			can be converted to a datetime.time or string formatted in a particular time format (e.g. 15:40:33, 17:44).
forms.DateTimeField()	<input type="text" ...>	forms.widgets.DateTimeInput()	Works just like CharField, but server-side Django validates the (text) value can be converted to a datetime.datetime, datetime.date or string formatted in a particular datetime format (e.g. 2017-12-25 14:30:59, 11/25/17 14:30).
forms.DurationField()	<input type="text" ...>	forms.widgets.TextInput()	Works just like CharField, but server-side Django validates the (text) value can be converted to a timedelta. Note Django uses the django.utils.dateparse.parse_duration() method as a helper, which means the string must match the format DD HH:MM:SS.uuuuuu (e.g. 2 1:10:20 for a timedelta of 2 days, 1 hour, 10 minutes, 20 seconds).
forms.SplitDateTimeField()	<input type="text" name="_0" ...> <input type="text" name="_1" ...>	forms.widgets.SplitDateTimeWidget	Works similar to DateTimeField but generates two separate text inputs for date & time, unlike DateTimeField which expects a single string with date & time. Validation wise Django enforces the date input can be converted to a datetime.date and the time input can be converted to a datetime.time.
forms.IntegerField()	<input type="number" ...>	forms.widgets.NumberInput()	Generates an HTML number input markup. Note this HTML5 markup is for client-side number validation and only works if the browser supports HTML5. If a browser doesn't support HTML5 then it treats this markup as a regular text input. Django server-side form validation is done for an integer number irrespective of HTML5 support.
forms.DecimalField()	<input type="number" ...>	forms.widgets.NumberInput()	Generates an HTML number input markup. Note this HTML5 markup is for client-side number validation and only works if a browser supports HTML5. If a browser doesn't support HTML5 then it treats this markup as a regular text input. Django server-side form validation is done for a decimal number irrespective of HTML5 support.
forms.FloatField()	<input type="number" ...>	forms.widgets.NumberInput()	Generates an HTML number input markup. Note this HTML5 markup is for client-side number validation and only works if a browser supports HTML5. If a browser doesn't support HTML5 then it treats this markup as a regular text input. Django server-side form validation is done for a float number irrespective of HTML5 support.
forms.ChoiceField(choices=tuple_of_tuples)	<select> <option value="tuple1_1" selected="selected">	forms.widgets.Select()	Generates an HTML select list from a tuple of tuples defined through choices (e.g.((1,'United States'),(2,'Canada'),(3,'Mexico'))). Note with ChoiceField if no value is selected

	tuple1_2 </option> <option value="tuple_2_1"> tuple_2_2 </option> <option value="tuple_3_1"> tuple_3_2 </option> </select>		an empty string '' is passed for post-processing and if a value like '2' is selected a literal string is passed for post-processing, irrespective of the original data representation. See TypeChoiceField or clean form methods to override these last behaviors for empty values and string handling.
forms.TypeChoiceField(choices=tuple_of_tuples, coerce=coerce_function, empty_value=None)	<select> <option value="tuple1_1" selected="selected"> tuple1_2 </option> <option value="tuple_2_1"> tuple_2_2 </option> <option value="tuple_3_1"> tuple_3_2 </option> </select>	forms.widgets.Select()	Works just like ChoiceField but provides extra post-processing functionality with the coerce and empty_value arguments. For example, with TypeChoiceField you can define a different default value with the empty_value arguement (e.g.empty_value=None) and you can define a coercion method with the coerce argument so the selected value is converted from its string representation (e.g. with coerce=int a value like '2' gets converted to 2 (integer) through the built-in int function).
forms.MultipleChoiceField(choices=tuple_of_tuples)	<select multiple='multiple'> <option value="tuple1_1" selected="selected"> tuple1_2 </option> <option value="tuple_2_1"> tuple_2_2 </option> <option value="tuple_3_1"> tuple_3_2 </option> </select>	forms.widgets.SelectMultiple()	Generates an HTML select list for multiple values from tuple of tuples defined through choices (e.g.((1,'United States'),(2,'Canada'),(3,'Mexico'))). It works just like ChoiceField but allows multiple values to be selected, which become available as a list in post-processing.
forms.TypedMultipleChoiceField(choices=tuple_of_tuples, coerce=coerce_function, empty_value=None)	<select multiple='multiple'> <option value="tuple1_1" selected="selected"> tuple1_2 </option> <option value="tuple_2_1"> tuple_2_2 </option> <option value="tuple_3_1"> tuple_3_2	forms.widgets.Select()	Works just like MultipleChoiceField but provides extra post-processing functionality with the coerce and empty_value arguments. For example, with TypedMultipleChoiceField you can define a different default value with the empty_value argument (e.g.empty_value=None) and you can define a coercion method with the coerce argument so the selected value is converted from its string representation (e.g. with coerce=int a value like '2' gets converted to 2 (integer) through the built-in int function).

	</option> </select>		

Above table shows all the built-in form fields in Django, which can mislead you to believe the same table also shows all Django built-in form widgets. The widget column in above table only shows the default widgets assigned to all built-in form fields.

There are a few more Django built-in widgets you can use that are also included in the **forms.widgets.package:**

Widget	Description
PasswordInput	Widget for password field (e.g. displays **** as a user types text). Also supports re-display of a field value after validation error.
HiddenInput	Widget for hidden field (e.g. <input type='hidden'...>
MultipleHiddenInput	Like HiddenInput but for multiple values (i.e. a list).
Textarea	Widget for text area field (e.g.<textarea></textarea>).
RadioSelect	Like the Select widget, but generates a list of radio buttons (e.g. <ul><li><input type="radio"></li>.. </ul>)
CheckboxSelectMultiple	Like the SelectMultiple widget, but generates a list of checkboxes (e.g. <ul><li><input type="checkbox"></li>. </ul>)
TimeInput	Like the DateTimeInput widget, but for time input only (e.g. 13:54, 13:54:59)
SelectDateWidget	Widget to generate three Select widgets for date (e.g. select widget for day, select widget for month, select widget for year).
SplitHiddenDateTimeWidget	Like the SplitDateTimeWidget widget, but uses Hidden input for date and time.
FileInput	Like the ClearableFileInput widget, but without a checkbox input to clear the field's value.

Examples:

Django forms fields:

Froms.py

```
from django import forms
class RegisterF(forms.Form):


        #characters inputs
        name=forms.CharField()
        address=forms.CharField(widget=forms.Textarea(attrs={'rows':3}))
        firstname=forms.CharField(widget=forms.TextInput)
        email=forms.EmailField()
        password=forms.CharField(widget=forms.PasswordInput)
```

```
            #hiddenpassword=forms.CharField(widget=forms.HiddenInput)
             #date inputs
            birthdate=forms.DateField()
            birth=forms.DateField(widget=forms.NumberInput(attrs={'type':'date'}))
            bi=['2019','2020','2021','2022']
            dateofbirth=forms.DateField(widget=forms.SelectDateWidget(years=bi))
            #number input
            intvalue=forms.IntegerField()
            floatvalue=forms.FloatField()
            decvalue=forms.DecimalField()
            #boolean input
            agree=forms.BooleanField()
             #choice inputs
            list1=[('Male','MALE'),('Female','FEMALE')]
            gender=forms.ChoiceField(choices=list1)
            list2=[('Male','MALE'),('Female','FEMALE')]
            person=forms.ChoiceField(widget=forms.RadioSelect, choices=list2)
            fav_color=[('Red','RED'),('Green','GREEN'), ('Pink','PINK'),('White','WHITE')]
            color=forms.MultipleChoiceField(choices=fav_color)
            fav=[('Red','RED'),('Green','GREEN'),('Pink','PINK'),('White','WHITE')]
            colorfav=forms.MultipleChoiceField(widget=forms.CheckboxSelectMultiple,choices=fav)
            #files input
            images=forms.ImageField()
            files=forms.FileField()
```

views.py

```
from django.shortcuts import render
from registerapp.forms import RegisterF
# Create your views here.
def registerp(request):
    form = RegisterF
    return render(request,"register.html",{'form':form})
```

Register.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Document</title>
</head>
<body>
    <form method="POST">
        {% csrf_token %}
        {{form.as_p}}
        <input type="submit" value="register">
    </form>
</body>
</html>
```

Urls.py

```
from django.urls import path
from registerapp import views
my_app='registerapp'
urlpatterns = [
    path('sample/',views.registerp,name="sample"),
]
```

Common arguments(attributes):

Required:

The required argument determines if the fields are required for form submission. It is a Boolean (true or false only) and creates the asterisk mark next to the field.

The argument is assigned to every field and is true by default.

email_address = forms.EmailField(required = False)

Label:

Django creates the field label by taking the designated field name, changing underscores to spaces, and the first letter to uppercase.

If you wish to create a custom label of a field, add the label argument.

email_address = forms.EmailField(label="Please enter your email address",)

Initial:

To add pre-loaded information to input, use the initial argument.Import datetime at the top of the file. Then you can set the initial input as the current date by using the import datetime. Add initial=True to the BooleanField() to set the default as a clicked checkbox in the HTML template.

first_name = forms.CharField(initial='Your name')

agree=forms. BooleanField(initial=True)

forms.DateField(initial=datetime.date.today)

Min_length and Max_length:

A minimum character length is assigned using min_length. A maximum character length is assigned to the user input using max_length. Only **#charfield, numberfields**

message = forms.CharField(min_length = 3, max_length = 10,)

Min_value, Max_value:

A minimum range is assigned using min_value. A maximum range is assigned to the user input using max_value. Only **#numberfields**

age = forms.IntegerField(min_value = 18, max_length = 35,)

How to process Input Data from the form inside views.py File:

- We required modifying views.py file. The end user provided input is available in a dictionary named with 'cleaned_data'

views.py:

```
from django.shortcuts import render
from . import forms
# Create your views here.
def studentinputview(request):
        form=forms.StudentForm()
        if request.method=='POST':
                form=forms.StudentForm(request.POST)
                if form.is_valid():
                        print('Form validation success and printing data')
                        print('Name:',form.cleaned_data['name'])
                        print('Marks:',form.cleaned_data['marks'])
        return render(request,'testapp/input.html',{'form':form})
```

Validating form values : is_valid().

- The is_valid() method is one of the more important parts of Django form processing.
- Once you create a bound form with request.POST, you call the is_valid() method on the instance to determine if the included values comply with a form's field definitions (e.g. if an EmailField() value is a valid email).
- Although the is_valid() method returns a boolean True or False value, it has two important side-effects:
- Calling is_valid() also creates the cleaned_data dictionary on the form instance to hold the form field values that passed validation rules.
- Calling is_valid() also creates the errors dictionary on the form instance to hold the form errors for each of the fields that didn't pass the validation rules.

Form Validations:

Once we submit the form we have to perform validations like
1) Length of the field should not be empty
2) The max number of characters should be 10
3) The first character of the name should be 'd' etc

We can implement validation logic by using the following 2 ways.
1) Explicitly by the programmer by using clean methods
2) By using Django inbuilt validators

Note: All validations should be implemented in the forms.py file

1) **Explicitly by the Programmer by using Clean Methods:**
    - The syntax of clean method: **def clean_fieldname(self)** (or) **def clean(self)**

- In the FormClass for any field if we define clean method then at the time of submits the form.
- Django will call this method automatically to perform validations. If the clean method won't raise any error then only form will be submitted.
  Example: the length of the max 6 characters

```
from django import forms
class Student(forms.Form):
        name = forms.CharField()
        rollno = forms.IntegerField()
        email = forms.EmailField()
        phone = forms.CharField(widget=forms.TextInput)

        def clean_name(self):
                inputname=self.cleaned_data['name']
                if len(inputname) > 6:
                        raise forms.ValidationError('The Minimum no of
                                                                characters should be 4')
                 return inputname
```

- The returned value of clean method will be considered by Django at the time of submitting the form.

Note:

- Django will call these filled level clean methods automatically and we are not required to call explicitly.
- Form validation by using clean methods is not recommended.

2) Django's Inbuilt Core Validators:

- Django provides several inbuilt core validators to perform very common validations.
- We can use these validators directly and we are not required to implement.
- Django's inbuilt validators are available in the django.core module.
    from django.core import validators
- To validate Max number of characters in the feedback as 40, we have to use inbuilt validators as follows.

forms.py

```
from django import forms
from django.core import validators

class Student(forms.Form):
        name=forms.CharField()
        rollno=forms.IntegerField()
        email=forms.EmailField()
        phone=forms.CharField(widget=forms.TextInput, validators=
                                        [validators.MaxLengthValidator(10)])
```

Note:

- We can use any number of validators for the same field.
  feedback= forms.CharField(widget = forms.Textarea,validators = [validators.MaxLengthValidator(40), validators.MinLengthValidator(10)])

- Usage of built in validators is very easy when compared with clean methods.

3) How to implement Custom Validators by using the same Validator Parameter:

- The value of name parameter should starts with 'd' or 'D'. We can implement this validation as follows

Forms.py

```
From django import forms
from django.core import validators

def starts_with_d(value):
        if value[0].lower() != 'd':
            raise forms.ValidationError('Name should be starts with d | D')

class FeedBackForm(forms.Form):
        name = forms.CharField(validators=[starts_with_d])
        rollno = forms.IntegerField()
```

Validation of Total Form directly by using a Single Clean Method:

- Whenever we are submitting the form Django will call clean() method present in our Form class. In that method we can implement all validations.

Forms.py

```
from django import forms
from django.core import validators

class Student(forms.Form):
        name=forms.CharField()
        rollno=forms.IntegerField()
        email=forms.EmailField()
        phone=forms.CharField(widget=forms.Textarea)

        def clean(self):
            print('Total Form Validation...')
            total_cleaned_data=super().clean()

            inputname=total_cleaned_data['name']
            if inputname[0].lower() != 'd':
                raise forms.ValidationError('Name parameter should starts with d')

            inputrollno=total_cleaned_data['rollno']
            if inputrollno <=0:
                raise forms.ValidationError('Rollno should be > 0')
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1>welcome to user form page</h1>
    {% if form.error %}
         {{form.error}
    {% else %}

    <form method="POST">
        {% csrf_token %}
        {{form.as_p}}
        <input type="submit" value="register">
    </form>
    {% endif %}
</body>
</html>
```

```
from django import forms
from django.core import validators

def start_with_d(value):
    print(value)
    if value[0].lower()!='d':
        raise forms.ValidationError('USERNAME SHOULD BE STARTS WITH D or d')

def phonenumber(value):
    if len(str(value))!=10:
        raise forms.ValidationError('PHONE NUMBER SHOULD BE 10 CHARACTERS')

class userform(forms.Form):
    username=forms.CharField(widget=forms.TextInput,validators=[start_with_d])
    password=forms.CharField(widget=forms.PasswordInput)
    email=forms.EmailField()
    phone=forms.IntegerField(widget=forms.TextInput,validators=[phonenumber])
    address=forms.CharField(widget=forms.Textarea)
    age=forms.IntegerField(widget=forms.TextInput)
class registerform(models.Model):
    username=models.CharField(max_length=30)
    email=models.EmailField(unique=True)
    phone=models.BigIntegerField(unique=True)
    address=models.CharField(max_length=100)
    age=models.IntegerField()
    password=models.CharField(max_length=30)

from django.shortcuts import render
from myapp.forms import userform
from myapp.models import registerform
# Create your views here.
def user(request):
    if request.method=='POST':
        form=userform(request.POST)
        if form.is_valid():
            username=request.POST['username']
            email=request.POST['email']
            address=request.POST['address']
            phone=request.POST['phone']
            age=request.POST['age']
            password=request.POST['password']
            p1=registerform.objects.create(username=username,password=password
,email=email,phone=phone,age=age,
                                          address=address)
            p1.save()
        else:
            return render(request,"user.html",{'form':form})
    form=userform()
    return render(request,"user.html",{'form':form})
```

Render Forms with Django Crispy Forms

- Django Crispy Forms is a term that refers to a type of Django code.
- When it comes to functionality, Django's built-in form capabilities are excellent, but they lack a polished design.
- Fortunately, django crispy forms is a Python tool that automatically styles Django forms CSS using built-in template packs.
- You can use this package to customise the rendering behaviour of your Django forms CSS without having to add any custom CSS declarations.

How to use Django crispy forms

1. To use this package, first install django crispy forms via pip and then add it to your Django project settings.

   ```
   pip install django-crispy-forms
   ```

2. Add **crispy_forms** to the list of installed apps in the settings.py file.

```
INSTALLED_APPS = [
             'main.apps.MainConfig',
            'django.contrib.admin',
            'django.contrib.auth',
            'django.contrib.contenttypes',
            'django.contrib.sessions',
            'django.contrib.messages',
            'django.contrib.staticfiles',
            'crispy_forms',
]
```

3. Add the **crispy_template_pack** to the settings.py file.
   CRISPY_TEMPLATE_PACK = 'uni_form'

   After that, put the crispy template pack underneath the list of installed apps. If you're not using a CSS framework, use the default crispy_template_pack ='uni form'.

   If the CSS framework you're using hasn't been established yet, the crispy forms package also includes the possibility to construct Django crispy forms custom template packs.

4. Using crispy forms in Django inside the projectload in the Django **crispy_forms_tags** in the HTML template.

```
#contact.html

{% load crispy_forms_tags %}
```

   Then, at the beginning of contact.html (or the template file of your choice), add load crispy forms tags. You may use this crispy form tags code to call the crisp form filters in the form below.

5. add the **|crispy or |as_crispy_field filter** to the Django form variable.
   - **Use the crispy filter in a Django form**
   - **Use the as_crispy_field filter in a Django form**

Use the crispy filter in a Django form

- To your Django form variable, add the |crispy filter.
  {{form|crispy}}

```
<!--Contact form-->
{% load crispy_forms_tags %}

<div style="margin:80px">

        <h1>Contact</h1>
        <form method="POST">
        {% csrf_token %}
       {{form|crispy}}
     </form>
  </div>
```

Use the as_crispy_field filter in a Django form

- If you want to change the order of Django form fields, call each one separately and then add |as crispy field to form.name of field.
  {{form.first_name|as_crispy_field}}

```
{% load crispy_forms_tags %}
        <!--Contact form-->
        <div style="margin:80px">
                <h1>Contact</h1>
                <h4>Contact us directly if you have any questions</h4>
```

```
                    <form method="POST">
                              {% csrf_token %}
                 {{form.first_name|as_crispy_field}}
                 {{form.last_name|as_crispy_field}}
                 {{form.email_address|as_crispy_field}}
                 {{form.comment|as_crispy_field}}
            </form>
          </div>
```

MEDIA FILES:

We've all utilised social media networks such as Instagram and Facebook. The one thing they all have in common is that users can contribute photographs and videos to them. Our files are uploaded to a server, where they are subsequently viewed by others. That is how social media works. So, how Django upload a file?

Working Django File Upload

Uploading files is the same as any other user input. For example, when you edit your profile image on Facebook. You choose a file from your device, and it is subsequently uploaded to Facebook's servers. The file is saved on the server and served whenever someone visits your profile. The file can be compared to static information on your website.

In Django, there are a variety of techniques to file uploading and handling. In Django, we'll use models to implement file upload. In this example, we'll use model forms.

Steps:

1. First create the application inside the project and after make a directory named static and also one more make a directory named media at the project level directory.

2. In the project's Settings.py file, set the static and media directory settings.
```
STATIC_URL = '/static/'
STATICFILES_DIRS=[STATIC_DIRS,]
MEDIA_ROOT=os.path.join(BASE_DIR,'media')
MEDIA_URL= '/media/'
```
   - By default, MEDIA ROOT is an empty string.
   - It accepts a directory's absolute file path. We passed in the file path of the media directory and utilised the join procedures. This directory will hold all of the files

that a Django user uploads. Although the path can be specified as a string in MEDIA ROOT.

- When you upload the file in the production environment, this method will cause issues. Use dynamic URLs that are relative to the project directory whenever possible.
- MEDIA URL behaves similarly to STATIC URL. Instead of providing files with an absolute path, Django uses this URL.
- When Django serves an uploaded file media, the URL is automatically updated.
- Remember that the values of MEDIA ROOT and MEDIA URL can be completely different. It is not necessary for the directory names to match. MEDIA URL can be given any name you choose.

3. Open the models.py in application and create model class regarding your requirements and when any media file upload to django. We want to two fields django have proper model field to handle uploaded files: FileField and ImageField.

- The files uploaded to FileField or ImageField are not stored in the database but in the file system.
- FileField and ImageField are created as astring filed in the database(usually VARCHAR), containing the reference to the actual file.
- If you delete a model instance containing FileField or ImageField, Django will not delete the physical file, but only reference to the file

4. Create the forms.py in application and ModelForm class regarding your requirements and when any media file get to user and upload to django.

- The basic concept of model forms is that they derive their structure from models. we don't have to define a form and connect it with models. We can do that directly via model forms.

- Models forms are a subclass of forms. When we make a form using **ModelForm** class, we use **META** class. In META, there are two attributes required, they are.

- **model:** It takes in the name of the model. We have to import the User_Profile model for this. This attribute defines the model to render a form. This form object is then used to save the user data to be stored directly in a database.

- **fields:** It is a list of strings. This list will take the name of the fields in the model. This field defines input fields for a form. We specify the fields from models in this list. All the fields mentioned in this list will be used to create a form.

5. Open the views.py in application and create a view regarding the upload the file.

- We have imported the models and forms in this view file. Then we have defined a list **IMAGE_FILE_TYPES**. We use it to check the type of file.
- Then our function definition starts.
- In this function, we have defined a form object. It will render a form. Then we are using form validation to clean our form data. This function returns True or False.

If the form has valid data, we can store the data locally. Then we are extracting the file type uploaded by the user.

- **request.FILES dictionary**
- When any file is uploaded in Django it becomes a part of the request object. More specifically, it can be accessed with the **request.FILES object**. It is a dictionary and has a key for each file uploaded. You can access data from the uploaded file via this dictionary.
- Although, we have an image file and we have not used any API to access its data.

6. Create Templates
- Now, again make a templates folder. The directory pattern will be **templates/loadread.html**. Remember to make two directories. First is templates and then the loadread directory inside it.
- Now, make a new file in **templates/read.html.**
- This template is used to display the image we just uploaded. We are using the **{{ user_pr.display_picture.url }}** in image src.

7. Configure URLs parent urls and child urls .

- We have made all the files. Open your main urls.py file. In this file.

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('user/',include('user.urls')),
]
if settings.DEBUG:
    urlpatterns += static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

- Concentrate on the if section. We are using these settings only when the project is in debug mode. Then we have added these patterns to the **urlpatterns variable**. These patterns are referred when a media file is requested. These settings are important when serving the file uploaded by the user.

8. After completed above steps than open cmd and execute below commands
   a. Python manage.py makemigrations
   b. Python manage.py migrate
   c. Python manage.py runserver

9. After run the server open chrome (any browser) and Search in URL bar.
   127.0.0.1:8000/upload

10. Test the application
note: You can upload image or files with the same name as we expect users to upload files with the same name. Django automatically generates a new name for the files uploaded. As you can see all the files have some code in their name. that is automatically generated by Django.

Example: **Upload the images into the django project.**

settings.py

```
BASE_DIR = Path(__file__).resolve().parent.parent
TEMPLATES_DIRS=os.path.join(BASE_DIR,"templates")
STATIC_DIRS=os.path.join(BASE_DIR,"static")

STATIC_URL = '/static/'
STATICFILES_DIRS=[STATIC_DIRS,]

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

Models.py

```
from django.db import models

# Create your models here.
class User(models.Model):
    fname = models.CharField(max_length=30)
    lname = models.CharField(max_length=30)
    email = models.EmailField(max_length=30)
    pic =models.FileField()
    def __str__(self):
        return self.fname
```

forms.py

```
from user.models import User
from django import forms
class User_pro(forms.ModelForm):
    class Meta:
        model = User
        fields = ['fname','lname','email','pic']
```

views.py

```
from django.shortcuts import render
from django.http import HttpResponse
from user.forms import User_pro
from user.models import User
from django.core.files.storage import FileSystemStorage
# Create your views here.
def create_user(request):
    up=""
    image_type=['png','jpg','jpeg']
    #image_type=['pdf','doc','docx','text']
    form=User_pro()
    if request.method=='POST'and request.FILES:
        form = User_pro(request.POST,request.FILES)
        if form.is_valid():
            up = form.save(commit=False)
            up.pic = request.FILES['pic']
            print(up.pic)
            file_type = up.pic.url.split('.')[-1]
            file_type = file_type.lower()
            if file_type not in image_type:
                return HttpResponse("error")
            up.save()
    return render(request,"loaduser.html",{"form":form,"fu":up})

def read(request):
    p1=User.objects.all()
    print(p1)
    return render(request,"read.html",{"fu":p1})
```

loaduser.html

```
{%load static%}
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>loaduser</title>
</head>
<body>
   <form method="POST" enctype="multipart/form-data">{%csrf_token%}
      <table>
         {{form.as_table}}
      </table>
      <input type="submit" name="submit" value="submit">
   </form>
   <h1>{{fu.fname}}</h1> <h1>{{fu.pic.url}}</h1>
   <img src="{{fu.pic.url}}" alt="image" height="100" width="100">
</body>
</html>
```

Read.html

```
{%load static%}
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>loaduser</title>
</head>
<body>
   <table border="2">
   {%if fu%}
   <th>FIRSTNAME</th>
 <th>LASTNAME</th>  <th>EMAIL</th> <th>PROFILE PIC</th>
   {%for i in fu%}
   <tr>
   <td>{{i.fname}}</td>  <td>{{i.lname}}</td> <td>{{i.email}}</td>
   <td><img src="{{i.pic.url}}" alt="image" width="100" height="75"></td>
</tr>
   {%endfor%}
   {%endif%}

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

User.urls(child urls)

```
from django.urls import path
from user import views
from django.conf import settings
from django.conf.urls.static import static
urlpatterns=[
    path("",views.create_user,name="createuser"),
    path("read/",views.read,name="read")
]
if settings.DEBUG:
    urlpatterns += static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

project.urls(parent urls)

```
from django.contrib import admin
from django.urls import path,include
from django.conf.urls.static import static
from django.conf import settings
from myapp import views
import user
urlpatterns = [
    path('admin/', admin.site.urls),
    path('user/',include('user.urls')),
]
if settings.DEBUG:
    urlpatterns += static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

ModelForm:

1. If you’re building a database-driven app, chances are you’ll have forms that map closely to Django models
2. In this case, it would be redundant to define the field types in your form, because you’ve already defined the fields in your model.
3. so in order to avoid such difficulties django has provided one helper class ModelForm which is responsible for creating forms based on models
4. Each and every form class must be inherited from ModelForm class of forms module

Advantages of ModelForms:
----
1. Mapping of model fields with form fields can be done easily
2. Reduces Code redundancy
3. Saving of te collected data from front-end can be done easily just by using Save() method
4. Customization of Model fields can be done easily

Syntax for Creating ModelForms:

1. Model forms are created inside the forms.py file only
   from django import forms
2. So first import models from app and forms module from django

```
class class_name(forms.ModelForm):
        class Meta():
                attributes=values
```

* Outer class is responsible for Defining a Model Form
* Inner Meta class is responsible for defining the information about the Model

Attributes of Meta class:

model:

- It is used for representing ModelName from which we need to create form

  syntax: **model = Modelname**

fields:

- It is used to specify which all the Model attributes are to be represented as form input input elements.
- Fields values can be represented in list or tuple

  syntax: fields = value

  fields='__all__' ------>takes all attributes of model

  fields=['attriute1','attriute2'......'attriuten']

exclude:

- It is used to specify which all the Model attributes are not to be represented as form input input elements.
- fields values can be represented in list or tuple

  syntax: **exclude==['attriute1','attriute2'......'attriuten']**

Customization of ModelForm attributes:

- Field types of Model are very sensitive while representing them as form field
- So we can convert CharField of Model to TextArea of form as shown below

  widgets={'fieldname':forms.Textarea()}

  By using the TextArea class forms module we can add TextArea for charfield
- help_text and labels for the ModelForms can be added by using gettext_lazy class of django.utils.translation

  from django.utils.traslation import gettext_lazy

  help_texts={"fieldname": gettext_lazy('message')}

  labels={"fieldname": gettext_lazy('message')}

Example:

Models.py

```
from django.db import models

# Create your models here.
class RegisterModel(models.Model):
    username=models.CharField(max_length=50)
    password=models.CharField(max_length=50)
    email=models.EmailField(max_length=50)
    phone=models.BigIntegerField()
    choice=[('MALE','Male'),("FEMALE",'Female')]
    gender=models.CharField(max_length=50,choices=choice)
```

forms.py

```
from django import forms
from myappdemo.models import RegisterModel
from django.utils.translation import gettext_lazy
class RegisterForm(forms.ModelForm):
    class Meta:
        model = RegisterModel
        fields=["username","password","phone"]
        #fields='__all__'
        #exclude=["username","password","phone"]

        #customize attributes
        widgets={'password':forms.PasswordInput,'phone':forms.TextInput,'username':forms.TextInput}
        help_texts={'username':gettext_lazy('plz enter the username'),'password':gettext_lazy('plz enter th
e password')}
        labels={'username':gettext_lazy('USERNAME'),'password':gettext_lazy('PASSWORD')}
```

views.py

```
from django.shortcuts import render
from myappdemo.forms import RegisterForm
# Create your views here.
def registermodelform(request):
    if request.method=='POST':
        form=RegisterForm(request.POST)
        if form.is_valid():
            form.save()
    form=RegisterForm
    return render(request,"user.html",{'form':form})
```

urls.py

```
from django.contrib import admin
from django.urls import path
from myappdemo import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('',views.registermodelform,name='register'),
]
```

How to apply bootstrap classes into model form :

Example:

Settings.py

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'myapp','myappdemo','dcookie','userregister',
    'crispy_forms',
]
CRISPY_TEMPLATE_PACK='bootstrap4'

STATIC_ROOT=os.path.join(BASE_DIR,"static")
STATIC_URL = '/static/'
MEDIA_ROOT=os.path.join(BASE_DIR,"media")
MEDIA_URL = '/media/'
```

Models.py

```
from django.db import models

# Create your models here.
class RegisterModel(models.Model):
    username=models.CharField(max_length=50)
    password=models.CharField(max_length=50)
    email=models.EmailField(max_length=50)
    phone=models.BigIntegerField()
    choice=[('MALE','Male'),("FEMALE",'Female')]
    gender=models.CharField(max_length=50,choices=choice)
```

forms.py

```
from django import forms
from userregister.models import UserRegister
from django.utils.translation import gettext_lazy
from django.core.files.storage import FileSystemStorage
class userregister(forms.ModelForm):
    class Meta:
        model =  UserRegister
        fields = ['username','password','repassword',"email","phone","address",'gender','imgs']
        widgets={'username':forms.TextInput(attrs={'class':"form-
control",'id':"floatingInput",'placeholder':"USERNAME"}),
                'password':forms.PasswordInput(attrs={'class':"form-
control",'id':"floatingInput",'placeholder':"password"}),
                'repassword':forms.PasswordInput(attrs={'class':"form-
control",'id':"floatingInput",'placeholder':"RE-PASSWORD"}),
                'email':forms.TextInput(attrs={'class':"form-
control",'id':"floatingInput",'placeholder':"name@example.com"}),
                'phone':forms.TextInput(attrs={'class':"form-
control",'id':"floatingInput",'placeholder':"PHONE"}),
                'address':forms.Textarea(attrs={'rows':2,'cols':10,'class':"form-
control",'id':"floatingInput",'placeholder':"ADDRESS"}),
                }
        labels={'username':gettext_lazy("USERANAME"),'password':gettext_lazy("PASSWORD"),
              'repassword':gettext_lazy("RE-PASSWORD"),'email':gettext_lazy("EMAIL"),
              'phone':gettext_lazy("phone"),'address':gettext_lazy("ADDRESS"),
              'gender':gettext_lazy("GENDER"),'imgs':gettext_lazy("PROFILE IMAGES")}


class UserLogin(forms.ModelForm):
    class Meta:
        model = UserRegister
        fields= ["username","password"]
        widgets={'username':forms.TextInput(attrs={'class':"form-
control",'id':"floatingInput",'placeholder':"name@example.com"}),
                 'password':forms.PasswordInput(attrs={'class':"form-
control",'id':"floatingInput",'placeholder':"password"})}
        labels={'username':gettext_lazy("USERANAME"),'password':gettext_lazy("PASSWPRD")}
```

base.html

```
{% load static %}
{% load crispy_forms_tags %}

<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <meta name="description" content="">
    <meta name="author" content="Mark Otto, Jacob Thornton, and Bootstrap contributors">
    <meta name="generator" content="Hugo 0.83.1">
    {%block title%}
    <title>BASE Template</title>
    {%endblock%}
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css" rel="stylesheet"
 integrity="sha384-
EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTwFspd3yD65VohhpuuCOmLASjC" crossorigin=
"anonymous">
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/js/bootstrap.bundle.min.js" integrity="
sha384-
MrcW6ZMFYlzcLA8Nl+NtUVF0sA7MsXsP1UyJoMp4YLEuNSfAP+JcXn/tWtIaxVXM" crossorigin
="anonymous"></script>

    <!-- Bootstrap core CSS -->

    <style>
      .bd-placeholder-img {
        font-size: 1.125rem;
        text-anchor: middle;
        -webkit-user-select: none;
        -moz-user-select: none;
        user-select: none;
      }

      @media (min-width: 768px) {
        .bd-placeholder-img-lg {
          font-size: 3.5rem;
        }
    }
   </style>


   <!-- Custom styles for this template -->
   <link href="{% static 'css/signin.css' %}" rel="stylesheet" type="text/css">

  </head>
  <body class="text-center">

   {%block content%}

   {%endblock%}

  </body>
</html>
```

Register.html

```
{% extends 'base.html'%}
{% load crispy_forms_tags %}
{% load static %}

   {%block title%}
   <title>Register page</title>
   {%endblock%}
   {%block content%}
   <main class="form-signin">
       <form method="POST" enctype="multipart/form-data">
         {% csrf_token %}
         <img class="mb-
4" src="{%static 'imgs/a.png'%}" alt="" width="72" height="57">
         <h1 class="h3 mb-3 fw-normal">Please sign in</h1>
         <div class="form-floating">{{form.username|as_crispy_field}}</div>
           <div class="form-floating">{{form.password|as_crispy_field}}</div>
         <div class="form-floating">{{form.repassword|as_crispy_field}}</div>
         <div class="form-floating">{{form.email|as_crispy_field}}  </div>
         <div class="form-floating">{{form.phone|as_crispy_field}}  </div>
         <div class="form-floating">{{form.address|as_crispy_field}}  </div>
         <div class="form-floating">  {{form.gender|as_crispy_field}} </div>
           <div class="form-floating"> {{form.imgs|as_crispy_field}}    </div>
<br><br>
         <button class="w-100 btn btn-lg btn-
primary" type="submit">Sign up</button>
         <p class="mt-5 mb-3 text-muted">&copy; 2017–2021</p>
       </form>
     </main>
   {%endblock%}
```

Login.html

```
{% extends 'base.html'%}
{% load crispy_forms_tags %}
{% load static %}
  {%block title%}
  <title>Login page</title>
  {%endblock%}
 <body class="text-center">
  {%block content%}
<main class="form-signin">
 <form method="POST">
  {% csrf_token %}
  <img class="mb-4" src="{% static 'imgs/a.png' %}" alt="" width="72" height="57">
  <h1 class="h3 mb-3 fw-normal">Please sign in</h1>

  <div class="form-floating">{{form.username|as_crispy_field}}</div>

  <div class="form-floating">{{form.password|as_crispy_field}}</div>

  <div class="checkbox mb-3">
   <label><input type="checkbox" value="remember-me"> Remember me</label>
  </div>
  <button class="w-100 btn btn-lg btn-primary" type="submit">Sign in</button>
  <p class="mt-5 mb-3 text-muted">&copy; 2017–2021</p>
 </form>
</main>

{%endblock%}

 </body>
</html>
```

Views.py

```
from django.shortcuts import render
from userregister.forms import *

# Create your views here.

def register(request):
    if request.method == "POST":
        form=userregister(request.POST,request.FILES)
        if form.is_valid():
            form.save()
    form = userregister()
    return render(request,"register.html",{'form':form})

def login(request):
    form = UserLogin()
    return render(request,"login.html",{'form':form})
```

urls.py

```
from django.urls import path
from userregister import views
from mform import settings
from django.conf.urls.static import static

urlpatterns=[
    path('login/',views.login,name='login'),
    path('register/',views.register,name='register'),
]

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

Built in login and logout authenticated in django:

1) create a django project and application
2) create a templates directory in project level.

    Register the templates inside the settings.py
3) create a directory called registration inside the template directory.
4) Create a login.html inside the registration directory.

```
 <h1>this login page</h1>
   <form method="POST">
       {% csrf_token %}
       {{form.as_p}}
       <input type="submit", value="sumbit">
</form>
```

5) Open the parent urls.py file in project level.

    Import the module called include

    from django.urls import path,include

    create the path for accounts/

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

6) Open the settings.py file and create configuration for database.(mysql)

```
import pymysql
pymysql.version_info=(1,4,2,"final",0)
pymysql.install_as_MySQLdb()
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'pro31',
        'USER':'root',
        'PASSWORD':'root',
        'HOST':'localhost',
        'PORT':'3306',
    }
}
```

7) Create a statement called LOGIN_REDIRECT_URL and LOGOUT_REDIRECT_URL.

```
LOGIN_REDIRECT_URL='home'
LOGOUT_REDIRECT_URL='home'
```

8) Open command prompt and user below commands.

    Python manage.py makemigrations

    Python manage.py migrate

9) Create superuser in django administration

    Python manage.py createsuperuser

    Give the username and password and renter password.

10) Open the 127.0.0.1:8000/admin then login to super user id then after Create normal user inside the django administration.
11) Open the tab and search below statement

    127.0.0.1:8000/accounts/login

12) Enter the username and password press submit button then
13) If user is authenticated open the home page.

Built in register form and login form and logout form.

User registration:

Django comes with a built-in user registration form. We just need to configure it to our needs (i.e. collect an email address and phone number upon registration):

- Django includes a pre-built registration form called UserCreationForm, which connects to the User model.
- The UserCreationForm, on the other hand, merely requires a username and password (password1 is the initial password and password2 is the password confirmation). Create a new file called forms.py in the app directory to customise the pre-built form.
- In the same directory as models.py and views.py, a new file is produced.
- Then, in a new class called NewUserForm, call UserCreationForm and add an email field and phone number field . To the user, save the email.
- Fill up the UserCreationForm with as many fields as you need.
- For the registration form, design an HTML template. This template will be known as register.html.
- Create a view regarding register
- Login with django.contrib.auth and import NewUserForm from forms.py.
- Then create a register request views function. Within the function, there are two if/else condition. The first determines whether or not the form is being submitted, while the second determines whether or not the form is valid.
- If both of these conditions are met, the form data is saved under a user's account, the user is logged in, and the user is sent to the homepage with a success message.
- If the form is not valid, an error message will be displayed. However, if the request isn't a POST in the first place, which means the first if statement failed, render the empty form in the register template.
- add a register path to the URLs.py .

- You can now go to the register URL, http://127.0.0.1:8000/register, in your browser window after the register function is complete.
- Display the success message "Registration successful" in same page. If everything is done successfully.

User login :

- You may have noticed that when a user registered an account, the views function we built instantly logged them in.
- We do, however, want the user to be able to log in without restriction. As a result, we'll need a login template, a URL, and a views feature.
- Go to views.py and add authenticate to the list of imports from **django.contrib.auth.forms**, then **import AuthenticationForm, from django.contrib.auth.forms** at the top.
- The Django form for logging in a user is called AuthenticationForm.
- Add an if/else statement to your login code that uses the Django authenticate function ().
- This function verifies user credentials (username and password) and returns the appropriate User object from the backend.
- The function will utilise **Django login()** to log in to the authenticated user if the credentials were authenticated by the backend. Otherwise, if the user is not authenticated, it displays a message to the user explaining that the username or password entered is invalid.
- If the form is not valid, the second else statement returns a similar error message.
- If the request is not a POST, the final else statement returns the login HTML template's blank form
- Now go to **http://127.0.0.1:8000/login** and log in to your user account.
- If you input the proper username and password, you will receive a success message and be logged in.

User Logout :

- User logout is the last thing we need to take care of. The logout link will be placed in the hyperlink, but it will only be visible to verified users (i.e. they are logged in).
- The current logged-in user is stored in the Django template variable user, and their rights are available in the template context.
- All we have to do now is add an if/else statement to show the logout link and the user's username if the user is authenticated, else show a login link.
- Finally, in the views file, add a logout function. When the logout URL is requested, the logout request method utilises the Django function logout() to log the user out of their account and send them to the homepage.

Project>registerlogin>Forms.py

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

class NewUser(UserCreationForm):
    email =forms.EmailField()
    phone =forms.IntegerField(widget=forms.TextInput)
    class meta:
        model = User
        fields={"username","email","phone","password1","password2"}
    def save(self, commit=True):
        user = super(NewUser,self).save(commit=False)
        user.email=self.cleaned_data['email']
        user.phone=self.cleaned_data['phone']
        if commit:
            user.save()
        return user
```

register.py

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>register</title>
</head>
<body>
    <h1>this register page</h1>
    {%for msg in messages%}
    <div>
        <strong>{{msg}}</strong>
        </div>
    {%endfor%}
    <form method="POST">{% csrf_token %}
        {{register_form.as_p}}
        <input type="submit", value="register">
    </form>
    <p>if you already have a account,<a href="/login">login</a></p>
</body>
</html>
```

Login.py

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>login</title>
</head>
<body>
    <h1>this login page</h1>
    {%for msg in messages%}
    <div>
        <strong>{{msg}}</strong>
        </div>
    {%endfor%}
    <form method="POST">
        {% csrf_token %}
        {{login_form.as_p}}
        <input type="submit", value="sumbit">
    </form>
</body>
```

```
</html>
```

Home.py

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>home page</title>
</head>
<body>

    <h1>this is home page</h1>
    {%for msg in messages%}
    <div>
        <strong>{{msg}}</strong>
        </div>
    {%endfor%}
    {%if user.is_authenticated %}
    <h3>haii   {{user.username}}</h3>
    <h4><a href="{% url 'logout' %}">LOGOUT</a></h4>
    {%else%}
    <p>you are not logged in</p>
    <h4><a href="{% url 'login' %}">LOGin</a></h4>
    {%endif%}
</body>
</html>
```

Views.py

```
from django.shortcuts import render,reverse,redirect
from django.http import HttpResponse,HttpResponseRedirect
from registerlogin.forms import NewUser
from django.contrib.auth import authenticate, login, logout
from django.contrib import messages
from django.contrib.auth.forms import AuthenticationForm
# Create your views here.
def register_req(request):
    if request.method=='POST':
        print("haii")
        form= NewUser(request.POST)
        if form.is_valid():
            print(form.cleaned_data["username"])
            user = form.save()
            login(request, user)
            messages.success(request,"registration successfully.")
        else:
            messages.error(request,"unsuccessfully registration ")
    form = NewUser()
    return render(request,"register.html",{"register_form":form})

def login_request(request):
    if request.method=="POST":
        form= AuthenticationForm(request,request.POST)
        if form.is_valid():
            username=form.cleaned_data['username']
            password=form.cleaned_data['password']
            print(username,password)
            user= authenticate(username=username,password=password)
            if user is not None:
                login(request,user)
                messages.info(request,f"login by mr {username}loggd in.")
                return HttpResponseRedirect(reverse('home'))
                #return redirect('home')
            else:
                messages.error(request,"invalid username and password")
        else:
            messages.error(request,"invalid username and password")
    form= AuthenticationForm
    return render(request,"login.html",{"login_form":form})

def home_req(request):
    return render(request,"home.html")

def logout_req(request):
    logout(request)
    messages.info(request,"you have successfully logged out")
    return redirect('home')
```

urls.py

```
from django.contrib import admin
from django.urls import path
from registerlogin import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path("register/",views.register_req,name="register"),
    path("login/",views.login_request,name="login"),
    path("home/",views.home_req,name="home"),
    ]
```

Function based views using curd operations (online shopping product):

1. **Create django project.**
2. **Create application(register the application name)**
3. **Create templates folder(register the templates directory)**
4. **Create static folder(register the static root and create static url)**
5. **Create media folder(register the media root and create media url)**
6. **Create and register the db configuration steps.**

Settings.py

```
import pymysql
pymysql.version_info=(1,4,2,"final",0)
pymysql.install_as_MySQLdb()
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'pro32',
        'USER':'root',
        'PASSWORD':'root',
        'HOST':'localhost',
        'PORT':'3306',
    }}
STATIC_ROOT=os.path.join(BASE_DIR,"static")
STATIC_URL = '/static/'
STATICFILESDIRS=[]

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

Models.py

```
from django.db import models

# Create your models here.
class Productdb(models.Model):
    pname=models.CharField(max_length=30)
    pdesc=models.CharField(max_length=200)
    pprice=models.FloatField()
    pdiscount=models.FloatField()
    pimg=models.ImageField(upload_to='product/%y/%m/%d')
```

Forms.py

```
from django import forms
from product.models import Productdb

class Productform(forms.ModelForm):
    class Meta:
        model=Productdb
        fields='__all__'
```

views.py

```
from django.shortcuts import render
from product.forms import Productform
from product.models import Productdb
from django.core.files.storage import FileSystemStorage
from django.http import HttpResponse,HttpResponseRedirect
from django.contrib import messages

#creating the product
def createproduct(request):
    if request.method=='POST':
        form = Productform(request.POST,request.FILES)
        print(form)
        if form.is_valid():
            form.save()
            messages.success(request,"your product is created")
    form = Productform()
    return render(request,"createpro.html",{'form':form})

#read the products
def readproduct(request):
    var=Productdb.objects.all()
    return render(request,"readpro.html",{'form':var})

#modify the product
def modifyproduct(request,id):
    #get the value inside db
    var=Productdb.objects.get(id=id)
    #post the values inside the db
    if request.method=='POST':
        form = Productform(request.POST,request.FILES,instance=var)
        if form.is_valid():
            form.save()
            messages.success(request,"modify the product information")
    return render(request,"modifypro.html",{'form':var})

#delete the product
def deleteproduct(request,id):
    #get the value inside db
    var=Productdb.objects.get(id=id)
    var.delete()
    var1=Productdb.objects.all()
    return HttpResponseRedirect('/readp/')
```

Createpro.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>create product</title>
</head>
<body>
    <div class=container align="center">
        {%for msg in messages%}
        <h3>{{msg}}</h3>
        {%endfor%}
        </div>
    <div class=container align="center">
        <form method="POST" enctype="multipart/form-data">
            {% csrf_token %}
            {{form.as_p}}
            <input type="submit", value="createproduct">
            </form>
            <br><br>
            <a href='/readp/'>display the products</a>
        </div>
</body>
</html>
```

Readpro.html

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

```
    <title>Readproducts</title>
</head>
<body>
    <div class=container align="center">
    {%for msg in messages%}
    <h3>{{msg}}</h3>
    {%endfor%}
    </div>

    <div class=container align="center">
    <table border="2">
  <th>productid</th>    <th>productname</th>     <th>productdesc</th>
 <th>productprice</th> <th>productdiscount</th> <th>productimage</th>
        <th>modifyproduct</th> <th>deleteproduct</th>

 {%for i in form%}
   <tr><td>{{i.id}}</td>  <td>{{i.pname}}</td> <td>{{i.pdesc}}</td>
<td>{{i.pprice}}</td>  <td>{{i.pdiscount}}</td>
        <td> <img src="{{i.pimg.url}}" alt="no img found" height="100" width="150"></td>
        <td> <a href='/modifyp/{{i.id}}'>modify</a></td>
        <td> <a href='/deletep/{{i.id}}'>delete</a></td>
        </tr>
        {%endfor%}
        </table>
        <br><br>
        <a href='/createp/'>create the products</a>
    </div>
</body>
</html>
```

Modifypro.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Modifyproduct</title>
</head>
<body>
    <div class=container align="center">
        {%for msg in messages%}
        <h3>{{msg}}</h3>
        {%endfor%}
        </div>
        <div class=container align="center">
        <form method="POST" enctype="multipart/form-data">
            {% csrf_token %}
            {%if form%}
            <table>
<tr><th><label for="id_pname">Pname:</label></th>
<td><input type="text" name="pname" value="{{form.pname}}" maxlength="30" required id="id_pn
ame"></td></tr>

<tr><th><label for="id_pdesc">Pdesc:</label></th>
<td><input type="text" name="pdesc" value="{{form.pdesc}}" maxlength="200" required id="id_pde
sc"></td></tr>
```

```
<tr><th><label for="id_pprice">Pprice:</label></th>
<td><input type="number" name="pprice" value="{{form.pprice}}" step="any" required id="id_ppric
e"></td></tr>

<tr><th><label for="id_pdiscount">Pdiscount:</label></th>
<td><input type="number" name="pdiscount" value="{{form.pdiscount}}" step="any" required id="id
_pdiscount"></td></tr>

<tr><th><label for="id_pimage">Pimage:</label></th>
<td><img src="{{form.pimg.url}}" alt="no img found" height="100" width="150"></td></tr>

<tr><th><label for="id_pimg">Pimg:</label></th><td><input type="file" name="pimg" accept="ima
ge/*" id="id_pimg"></td></tr>
    </table>
    {%endif%}
    <input type="submit", value="modifyproduct">
    </form>
    <br><br>
    <a href='/readp/'>display the products</a>
    </div>
</body>
</html>
```

Urls.py

```
from django.contrib import admin
from django.urls import path
from product import views
from onlineshop import settings
from django.conf.urls.static import static
my_app="product"

urlpatterns = [
    path('admin/', admin.site.urls),
    path('createp/',views.createproduct,name="createp"),
    path('readp/',views.readproduct,name="readp"),
    path('modifyp/<id>',views.modifyproduct,name="modifyp"),
    path('deletep/<id>',views.deleteproduct,name="deletep"),
]

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

Views

There are 2 types of views

1) Function Based Views
2) Class Based Views

- Till now we made a function as a responsible for views. But now new concept of this, we are going to make class as a responsible for views.
- All this class should be derived & inherited from any of the view class of django then we write a built-in method to override the method based on the requirements,

Important points regarding views:
1. A view is a callable which takes a request and returns a response.
2. This can be more than just a function, and Django provides an example of some classes which can be used as views

Class-based views:

1. Class-based views provide an alternative way to implement views as Python objects instead of functions.
2. CBV do not replace function-based views, but they have certain differences and advantages when compared to function-based views

The differences are:
1. Organization of code related to specific HTTP methods (GET, POST, etc.) can be addressed by separate methods instead of conditional branching.
2. Object oriented techniques such as mixins (multiple inheritance) can be used to factor code into reusable components.

Key points we have to follow when we r writing CBV:
---
1. Each and Every CBV must be inherited from any one of the View classes of Django
2. Url of CBV is created by using below syntax

```
path('suffix/',views.classname.as_view(),name='name of the url')
```

Some Of the View Classes of django along with their module name:
----

view Class_Name	**location of the view class**
1. View	django.views.generic
2. TemplateView	django.views.generic
3. ListView	django.views.generic
4. DetailView	django.views.generic
5. FormView	django.views.generic
6. CreateView	django.views.generic
7. UpdateView	django.views.generic
8. DeleteView	django.views.generic

Class based views:

- Class Based Views introduced in Django 1.3 Version to implement Generic Views.

- When compared with Function Based views, class Based views are very easy to use.
- Hence Class Based Views are most frequently used views in real time.
- Internally Class Based Views will be converted into Function Based Views. Hence Class Based Views are simply acts as wrappers to the Function based views to hide complexity.
- Function Based views are more powerful when compared with Class Based Views.
- **Explain the Scenario where we should use Function based Views only and we cannot use Class based Views?**
- For simple operations like listing of all records or display details of a particular record then we should go for Class Based Views.
- For complex operations like handling multiple forms simultaneously then we should go for Function Based Views.
- Create a class based view to display "helloworld " on screen.

HelloWorld Application By using ClassBasedViews
views.py

```
from django.views.generic import View
 from django.http import HttpResponse
 class HelloWorldView(View)
          def get(self,request):
          return HttpResponse('This is from ClassBasedView')
```

urls.py

```
from testapp import views
urlpatterns = [
path('', views.HelloWorldView.as_view(),name="hello"),
 ]
```

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 to provide response to the GET request. Similarly for other HTTP Methods also like POST,HEAD etc
3) When you doing url mapping at the place where we are going to give the **views.function_name** your should specify **views.classname.as_view().**

Example for function based view Vs class based view

```
from django.shortcuts import render
from django.http import HttpResponse
from django.views import View
from sampleview.models import funmodel,clsmodel,Funclsmodel
from sampleview.forms import FunclsFORM
# Create your views here.
```

```
def  funview(request):
          return render(request, "sampleview.html")

class clsview(View):
   def get(self,request):
      return render(request, "sampleview.html")
---------------------------------------------------------------------------------------------------------------
def  funview(request):
         return HttpResponse("<h1>haii welcome to function based view</h1>")

class clsview(View):
   def get(self,request):
      return HttpResponse("<h1>haii welcome to class based view</h1>")


---------------------------------------------------------------------------------------------------------------
def  funview(request):
   data="haii welcome to function based view"
   return render(request, "sampleview.html",{"data":data})

class clsview(View):
   data="haii welcome to class based view"
   def get(self,request):
      return render(request, "sampleview.html",{"data":self.data})


---------------------------------------------------------------------------------------------------------------
def  funview(request,id):
   return render(request, "sampleview.html",{"data":id})

class clsview(View):
   data="haii welcome to class based view"
   def get(self,request,id):
      return render(request, "sampleview.html",{"data":id})
---------------------------------------------------------------------------------------------------------------
def  funview(request,id):
   var=funmodel.objects.filter(id=id)
   return render(request, "sampleview.html",{"data":var})

class clsview(View):
   def get(self,request,id):
      var=clsmodel.objects.filter(id=id)
      return render(request, "sampleview.html",{"data":var})
---------------------------------------------------------------------------------------------------------------


def  funview(request,id):
   var=funmodel.objects.create(funtitle=id)
   var.save()
   return render(request, "sampleview.html",{"data":id})

class clsview(View):
   def get(self,request,id):
        var=clsmodel.objects.create(clstitle=id)
        var.save()
        return render(request, "sampleview.html",{"data":id})
-----------------------------------------------------------------------------------------------------------

def  funview(request,id):
    var=funmodel.objects.create(funtitle=id)
    var.save()
    p1=funmodel.objects.all()
    return render(request, "sampleview.html",{"data":p1})

class clsview(View):
    def get(self,request,id):
        var=clsmodel.objects.create(clstitle=id)
        var.save()
        p1= clsmodel.objects.all()
        return render(request, "sampleview.html",{'data':p1})
-----------------------------------------------------------------------------------------------------------
def  funview(request):
    if request.method == "POST":
        form= FunclsFORM(request.POST)
        if form.is_valid():
            print(form.cleaned_data['username'])
            print(form.cleaned_data['password'])
            return HttpResponse("welcome to function home page")

    form= FunclsFORM
    return render(request,"funcls.html",{'form':form})


class clsview(View):
    def get(self,request):
        form= FunclsFORM
        return render(request,"funcls.html",{'form':form})

    def post(self,request):
        if request.method == "POST":
            form= FunclsFORM(request.POST)
            if form.is_valid():
                print(form.cleaned_data['username'])
                print(form.cleaned_data['password'])
                return HttpResponse("welcome to class home page")
-----------------------------------------------------------------------------------------------------------
def  funview(request):
    if request.method == "POST":
        form= FunclsFORM(request.POST)
        if form.is_valid():
            user= form.cleaned_data['username']
            password= form.cleaned_data['password']
            print(user, password)
            p1=Funclsmodel.objects.create(username=user, password=password)
            p1.save()
            return HttpResponse("welcome to function home page")

    form= FunclsFORM
    return render(request,"funcls.html",{'form':form})



class clsview(View):
    def get(self,request):
        form= FunclsFORM
        return render(request,"funcls.html",{'form':form})

    def post(self,request):
        if request.method == "POST":
            form= FunclsFORM(request.POST)
            if form.is_valid():
                user= form.cleaned_data['username']
                password= form.cleaned_data['password']
                print(user, password)
                p1=Funclsmodel.objects.create(username=user, password=password)
                p1.save()
                return HttpResponse("welcome to class home page")
------------------------------------------------------------------------------------------------------------------
def  funview(request):
    if request.method == "POST":
        form= FunclsFORM(request.POST)
        if form.is_valid():
            form.save()
            return HttpResponse("welcome to function home page")

    form= FunclsFORM
    return render(request,"funcls.html",{'form':form})



class clsview(View):
    def get(self,request):
        form= FunclsFORM
        return render(request,"funcls.html",{'form':form})

    def post(self,request):
        if request.method == "POST":
```

```
        form= FunclsFORM(request.POST)
        if form.is_valid():
            form.save()
            return HttpResponse("welcome to class home page")
```

TemplateView:

- The Django generic TemplateView view class enables developers to quickly create views that display simple templates without reinventing the wheel.
- Instead of extending View, override the get method and then process the template and return an HttpResponse object using a render function.
- Simply extend TemplateView when we are dealing with Html files

first we have to import View class by below syntax:

```
from django.views.generic import TemplateView
```

Rendering a Normal html file with TemplateView view class:

```
class Class_name(TemplateView):
        template_name='name_of_the template'
```

Using TemplateView in urls.py:

1. you can use TemplateView directly in your URL. This provides you with a quicker way to render a template
2. in this case there is no need of creating any class inside views directly we can write in Urls in this type we have to import TemplateView into urls

syntax of url in Urls.py file:

```
urlpatterns=[
        path('suffix/',TemplateView.as_view(),name="mapping name"]
        ]
```

Django Template Context with TemplateView:

- If you are using TemplateView, you need to use the get_context_data method to provide any context data variables to your template.
- After that we have to re-define the get_context_data method and provide an implementation which simply gets a context dict object from the parent class (in this example it's TemplateView) then augments it by passing the message data.

Example:

```
class CBV_contextdata(TemplateView):
        template_name='CBV_context.html'
        def get_context_data(self,**kwargs):
                context=super().get_context_data(**kwargs)
                #context['data1']='hai hello how r u'
                #context['data2']='this second data'
                context['form']=Student()
                return context

        def post(self,request):
                form_data=Student(request.POST)
                if form_data.is_valid():
                return HttpResponse(str(form_data.cleaned_data))
```

Note:
1. TemplateView shouldn’t be used when your page has forms and does creation
   or update of objects.
2. In such cases FormView, CreateView or UpdateView is a better option.

Example:

- If ever wanted to display the template as a view is that case we specifically go for a TemplateView class.
- Inheritance from **django.views.generic** where we create a class which is inherited from template view and we override a variable called Template_name where we specify the name of the Templates.

View.py

```
from django.views.generic import TemplateView
class Demotemplateview(TemplateView):
        template_name=”sample.html”
```

- to pass the context data in template view class we need to override a method get_content_data as object method inside TemplateView class.
- How to send Context Paramters:

```
Class TemplateCBV(TemplateView):
    template_name = 'home.html'
    def get_context_data(self,**kwargs):
        context = super().get_context_data(**kwargs)
        context['name'] = 'venu'
        context['phone'] = 9966206473
        return context
```

- In template file we can access context parameters as follows(inside the html file)

```
{{name}} {{phone}}
```

Example function based views Vs Class based view:

```
def templater(request):
   return render(request,"index.html")
```

```
class Tempview(TemplateView):
    template_name = 'index.html'
    ------------------------------------------------------------------------------------------------------
def templater(request):
    return render(request,"index.html",{'name':'venugopal','age':25,'address':'andhra'})

class Tempview(TemplateView):
    template_name = 'index.html'
    def get_context_data(self,**kwargs):
        context=super().get_context_data(**kwargs)
        context['name']='sai'
        context['age']=20
        context['address']='chennai'
        return context
   ------------------------------------------------------------------------------------------------------


def templater(request):
    form =  Bookdetails()
    return render(request,"index.html",{'form':form})

class Tempview(TemplateView):
    template_name = 'index.html'
    form = Bookdetails()
    def get_context_data(self,**kwargs):
        context=super().get_context_data(**kwargs)
        context['form']=self.form
        return context
     ------------------------------------------------------------------------------------------------------


def templater(request):
    if request.method=="POST":
        form =  Bookdetails(request.POST)
        if form.is_valid():
            title=request.POST['title']
            author=request.POST['author']
            pages=request.POST['pages']
            price=request.POST['price']
            p1=Book.objects.create(title=title,author=author,pages=pages,price=price)
            p1.save()
            return HttpResponse("insert the record")
    form =  Bookdetails()
    return render(request,"index.html",{'form':form})

class Tempview(TemplateView):
    template_name = 'index.html'
    form = Bookdetails()
    def get_context_data(self,**kwargs):
        context=super().get_context_data(**kwargs)
        context['form']=self.form
        return context
    def post(self,request):
        if request.method=="POST":
            form =  Bookdetails(request.POST)
            if form.is_valid():
                title=request.POST['title']
                author=request.POST['author']
                pages=request.POST['pages']
                price=request.POST['price']
                p1=Book.objects.create(title=title,author=author,pages=pages,price=price)
                p1.save()
                return HttpResponse("insert the record")
```

models.py file

```
from django.db import models
class Book(models.Model):
    title=models.CharField(max_length=50)
    author=models.CharField(max_length=50)
    pages=models.IntegerField()
    price=models.IntegerField()
    def __str__(self):
        return self.name
```

forms.py

```
from django import forms
class Bookdetails(forms.Form):
    title=forms.CharField()
    author=forms.CharField()
    pages=forms.IntegerField()
    price=forms.IntegerField()
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1>welcome to index page</h1>
    <form method="POST">
        {% csrf_token %}
        {{form}}
        <input type="submit" value="login">
    </form>
</body>
</html>
```

Urls.py

```
from django.urls import path
from django.conf.urls import url
from cbv import views
urlpatterns=[
    #path('funindex/',views.templater,name="funindex"),
    #path('clsindex/',views.Tempview.as_view(),name="clsindex"),
    path('funlist/',views.funlist,name="funlist"),
    path('dislist/',views.displayview.as_view(),name="dislist"),
    path('disview/<pk>/',views.displaylist.as_view(),name="disview"),
]
```

FromView:

1. FormView should be used when you need a form on the page and want to perform certain action when a valid form is submitted.
2. We use form_class attribute to specify form class.
3. we use form_valid method inorder to validate and perform some opearations on submitted data

First we have to import View class by below syntax:

```
from django.views.generic import FormView
```

Creating a CBV with FormView view class

```
class Form(FormView):
        form_class=forms.Student        # specifying the Form class you want to use
        template_name='templateview.html'    #sepcify name of template

        def form_valid(self,form):
                data=form.cleaned_data
                return HttpResponse(str(data))
```

Creating a CBV with FormView view class and saving data into database:

```
class FormModel(FormView):
        form_class=forms.TopicForm         # specifying the Form class you want to use
        template_name='templateview.html'   #sepcify name of template

            def form_valid(self,form):
                form.save()
                return HttpResponse('Form is sumitted successfully')
```

Generic display views:

Django has Two generic class-based views which are designed to display data

1. ListView
2. DetailedView

ListView:

- Displaying a db data in the form of a list is done by using ListView.
- We can use ListView class to list out all records from database table (model). It is alternative to ModelClassName.objects.all()
- If we don't specify 'context_object_name' then by default it takes "modelname_list".
- Django will identify template automatically and we are not required to configure anywhere.
- But Django will always search for template file with the name **modelclassname_list.html** like book_list.html
- Django will always search for template file in the following location. projectname/appname/templates/appname/
- bydefault django will provide context object to the template file with the name: modelclassname_list Eg: book_list.

How to provide our own Context Object Name:

- By default the ListView is going to pass context object name is **modelclassname_list** as context object.
- If ever we wanted to any changes in the name in that case we should create an attribute with name **context_object_name** =”**contextname**” which you need.
  Example:

```
BookListView(ListView):
        context_object_name = 'books'
        model = Book
```

How to configure our own Template File at Project Level:

- By default, the listView is going to call the template on under the template which is there in address of “**appname/modelname_list.html**”.
  Example: templates/appname/modelname_list.html
  (templates and appname are directory.)

- If ever we wanted to any changes in the template name in that case we should create a one more argument that is equal to **“app_name/template.html”** which you need.
- Of course this approach is not recommended class

```
BookListView(ListView):
        context_object_name = 'books'
        model = Book
        template_name = 'testapp/durga.html'
```

- Note: Even if we are not specifying template_name variable, still django can recognize project level template file. But name should be modelclassname_list.html.

attributes of list view:

model	------->to specify model name(by default it fetch all data)
queryset	----->it is used to specify filtering of model data
context_object_name	----->it specifies context name
template_name	----->it is used specify the Html file file to render
ordering	-----> It is used order the data based on columns

Example:ListView

Models.py

```
from django.db import models
class Book(models.Model):
      title=models.CharField(max_length=300)
      author=models.CharField(max_length=30)
      pages=models.IntegerField()
      price=models.FloatField()
```

views.py

```
from testapp.models import Book
from django.views.generic import ListView
class BookListView(ListView):
        model=Book
```

templates/testapp/book_list.html

```
<!DOCTYPE html>
<html lang="en">
<head>
   <title>Document</title>
</head>
<body>
   <h1>this book list display page</h1>
```

```
{% for book in book_list%}
<h1> Title:{{book.title}}</h1>
<h1> Author:{{book.author}}</h1>
 <h1> Pages:{{book.pages}}</h1>
 <h1> Price:{{book.price}}</h1>
{%endfor%}
</body>
</html>
```

Urls.py

```
from django.urls import path

from testapp import views

urlpatterns=[

    path('funindex/',views. BookListView.as_view(), name="funindex"),
```

2. DetailView:

- DetailView should be used when you want to represent detail of a single model instance.

important points to remember:

- First of all, when it comes to web development you really want to avoid hard coding paths in your templates.
- The reason for this is that paths will be different for each and every instances of the model
- so it becomes very difficult to find out and change each and every url path manually
- So we have to define function which is responsible for returning the URL paths based on selected model instance
- This can be done by using get_absolute_url method

get_absolute_url():

1. get_absolute_url() method is used to tell Django how to create the canonical URL for an model object by providing model instance reference. A canonical URL is "the official" url to a certain page.
2. get_absolute_url method must be defined inside the model class as shown below along with reverse function of url

models.py

```
from django.db import models
from django.urls import reverse

class School(models.Model):
            name=models.CharField(max_length=100)
            principal=models.CharField(max_length=100)
```

```
        location=models.CharField(max_length=100)

        def __str__(self):
                return self.name

        def get_absolute_url(self):
                return reverse("detail",kwargs={"pk": self.pk})

class Student(models.Model):
        name=models.CharField(max_length=100)
        age=models.PositiveIntegerField()
        school=models.ForeignKey(School,on_delete=models.CASCADE,related_na
me="students")

        def __str__(self):
                return self.name
```

School_detail.html

```
{% extends "myapp/base.html" %}

{% block body_block %}
<div class="container">
<div class="jumbotron">
   <h1>School Details Are : </h1>
   <table  border="2pt">
      <tr>
      <th>Name</th>
      <th>Principal</th>
      <th>Location</th>
      </tr>
      <tr>
         <td>{{school.name}}</td>
         <td>{{school.principal}}</td>
         <td>{{school.location}}</td>
      </tr>
   </table>
</div>
<div class="jumbotron">
   <h1>Students Details are : </h1>
   {% for student in school.students.all %}
   <h3>Student Name : {{student.name}}  Age : {{student.age}}</h3>
   {% endfor %}
</div>
<a href="{% url 'myapp:update' pk=school.pk %}" class="btn btn-warning">Update</a>
<a href="{% url 'myapp:delete' pk=school.pk %}" class="btn btn-warning">Delete</a>
</div>
```

```
{% endblock %}
```

Using regular expressions in defining urls:
1. For creating cananical urls based on selected model instance we cannot give
   exact values instead we have to define a pattern
2. that can be done by using
    Python regular expressions, the syntax for named regular expression groups is

(?P<name>pattern)

```
where ?P   ------>is to capture the contents of <>
 name   ---->it is the name by using which we can access content
 pattern---->it is some pattern to match.
```

3. In-order to create urls with regular expressions we have to use
    re_path instead of path function

example:

```
from django.urls import re_path
 urlpatterns=[
                  re_path('(?P<pk>\d+)/',views.classname.as_view(),name='name'),
                 ]
```

Editing Views:
Django has Three generic class-based views which are designed to Edit the data
1. CreateView
2. UpdateView
3. DeleteView

CreateView:
1. CreateView should be used when you need a form on the page and need to do a db insertion on submission of a valid form.
2. A view that displays a form for creating an object, redisplaying the form with validation errors (if there are any) and saving the object.
3. The CreateView page displayed to a GET request uses a template_name_suffix of '_form'
4. so create a html file with modelname_form.html so that CreateView will automatically render that file without even mentioning template_name

Example:
Views.py

```
class SchoolCreateView(CreateView):
    model=School
    #fields='__all__'     (fields functionality is same as in Meta class)
    fields=('name','principal','location')
```

school_form.html

```
{% extends "myapp/base.html" %}

{% block body_block %}
    <h1>Creaate School By Filling Form : </h1>

  <form method="POST">
      {% csrf_token %}
      {{form.as_p}}
      <input type="submit" value="Create" class="btn btn-primary">
  </form>
{% endblock %}
```

UpdateView:

1. A view that displays a form for editing an existing object, redisplaying the form with validation errors (if there are any) and saving changes
to the object.
2. This uses a form automatically generated from the object's model class (unless a form class is manually specified).
3. The UpdateView page displayed to a GET request uses a template_name_suffix of '_form'

views.py:

```
class SchoolUpdateView(UpdateView):
    model=School
    fields=('name','principal','location')
```

Using same school_form.html file with some changes:

```
{% extends "myapp/base.html" %}

{% block body_block %}
{% if form.instance.pk %}
  <h1>Update The Details</h1>
{% else %}
    <h1>Creaate School By Filling Form : </h1>
{% endif %}

  <form method="POST">
      {% csrf_token %}
      {{form.as_p}}
      {% if form.instance.pk %}
      <input type="submit" value="Update" class="btn btn-warning">
      {% else %}
    <input type="submit" value="Create" class="btn btn-primary">
    {% endif %}

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

urls.py:

```
re_path('^update/(?P<pk>\d+)/',views.SchoolUpdateView.as_view(),name="update"),
```

DeleteView:

1. A view that displays a confirmation page and deletes an existing object.
2. The given object will only be deleted if the request method is POST.
3. If this view is fetched via GET, it will display a confirmation page that should contain a form that POSTs to the same URL.
4. The DeleteView page displayed to a GET request uses a template_name_suffix of '_confirm_delete'.

views.py:

```
from django.urls import reverse_lazy

class SchoolDeleteView(DeleteView):
    model=School
    context_object_name="school"
    success_url=reverse_lazy('myapp:list')
```

school_confirm_delete.html content:

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

{% block body_block %}
<h1>Are Your Sure In Deleting {{school}} ?</h1>
  <form method="POST">
    {% csrf_token %}
    <input type="submit" value="Delete" class="btn btn-security">
    <a href="{% url 'list' %}" class="btn btn-primary">Cancel</a>

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

urls.py:

```
re_path('^delete/(?P<pk>\d+)/',views.SchoolDeleteView.as_view(),name="delete"),
```

Class based views in curd(Create, Update, Read, Delete) operations.

models.py

```
from django.db import models
from django.urls import reverse

# Create your models here.
class School(models.Model):
    name=models.CharField(max_length=30)
    principal=models.CharField(max_length=30)
    location=models.CharField(max_length=30)
    def __str__(self):
        return self.name
    def get_absolute_url(self):
        return reverse('curd:details', kwargs={"pk":self.pk})
```

templates/curd/school_form.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1> create a school details </h1>
 <form method="POST">
    {% csrf_token %}
    {{form.as_p}}
    <input type="submit" value="create">
    </form>
</body>
</html>
```

templates/curd/school_list.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <div>
        <ol>
        {% if school_list %}
            {%for school in school_list%}
           <li> <h1><a href="/curd/details/{{school.id}}/">{{school}}</h1></li>
            {%endfor%}
        {%else%}
        <h1>no school found</h1>
        {%endif%}
            </ol>
    </div>
</body>
</html>
```

templates/curd/school_detail.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1>school details</h1>
    <h2>{{school.name}}</h2>
    <h2>{{school.principal}}</h2>
    <h2>{{school.location}}</h2>

   <h1> <a href="/curd/update/{{school.pk}}">update</a></h1>
   <h1> <a href="/curd/delete/{{school.pk}}">delete</a></h1>
</body>
</html>
```

templates/curd/school_confirm_delete.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Document</title>
</head>
<body>
    <h1>are you sure delete {{school}}</h1>
    <form method="POST">
        {% csrf_token %}
        <input type="submit" value="delete">
        <a href="/curd/list"> displaylist</a>
        </form>
</body>
</html>
```

Views.py

```
from django.shortcuts import render
from django.views import View
from django.views.generic import CreateView,ListView,DetailView,UpdateView,DeleteView
from curd.models import School,Student
from django.urls import reverse_lazy
# Create your views here.

class HomeView(View):
    def get(self, request):
        return render(request,'myapp/base.html')

class Schoollistview(ListView):
    model=School
    template_name='curd/school_list.html'

class Schooldetailsview(DetailView):
    model=School
    context_objects_name="schools"
    template_name='curd/school_detail.html'

class Schoolcreateview(CreateView):
    model = School
    fields='__all__'

class Schoolupdateview(UpdateView):
    model = School
    fields='__all__'

class SchoolDeleteview(DeleteView):
    model = School
    success_url=reverse_lazy('curd:list')
```

urls.py

```
from django.urls import path
from django.contrib import admin
from curd import views
app_name="curd"
urlpatterns=[
    path('admin/', admin.site.urls),
    path('home/',views.HomeView.as_view(),name="home"),
    path('create/',views.Schoolcreateview.as_view(),name="create"),
    path('list/',views.Schoollistview.as_view(),name="list"),
    path('details/<pk>/',views.Schooldetailsview.as_view(),name="details"),
    path('update/<pk>/',views.Schoolupdateview.as_view(),name="update"),
    path('delete/<pk>/',views.SchoolDeleteview.as_view(),name="delete"),
]
```

Session Management:

- The client and server can communicate through HTTP, which is a common language.
- HTTP's main drawback is that it is a stateless protocol. i.e., it is unable to remember client information for future queries.
- Every request made to the server is considered a new request. As a result, a method on the server side is required to remember clients.
- As a result, a method on the server side is required to remember client information across subsequent queries. This technique is nothing more than a session management system.

  The mechanisms for managing sessions are listed below.

  1) The cookies
  2) API for Sessions
  3) Rewrite URLs
  4) hidden form fields

Cookies :

- A cookie is a small piece of information which is stored in the client browser. It is used to store user's data in a file permanently (or for the specified time).
- Cookie has its expiry date and time and removes automatically when gets expire. Django provides built-in methods to set and fetch cookie.
- A cookie is a small piece of information which is stored in the client browser. It is used to store user's data in a file permanently (or for the specified time).
- Cookie has its expiry date and time and removes automatically when gets expire. Django provides built-in methods to set and fetch cookie.
- In views.py, two functions setcookie() and getcookie() are used to set and get cookie respectively

Views.py

```
from django.shortcuts import render
from django.http import HttpResponse
def setcookie(request):
        response = HttpResponse("Cookie Set")
        response.set_cookie('python', 'python.org')
        return response

def getcookie(request):
        tutorial = request.COOKIES['python']
        return HttpResponse("python tutorials @: "+ tutorial)
```

- And URLs specified to access these functions

urls.py

```
from django.urls import path
from .views import *
urlpatterns = [
        path('scookie', setcookie),
        path('gcookie', getcookie)
]
```

- Start Server After starting the server,
- set cookie by using localhost:8000/scookie URL.
- It shows output to the browser
- And get a cookie by using localhost:8000/gcookie URL.
- It shows the set cookie to the browser.

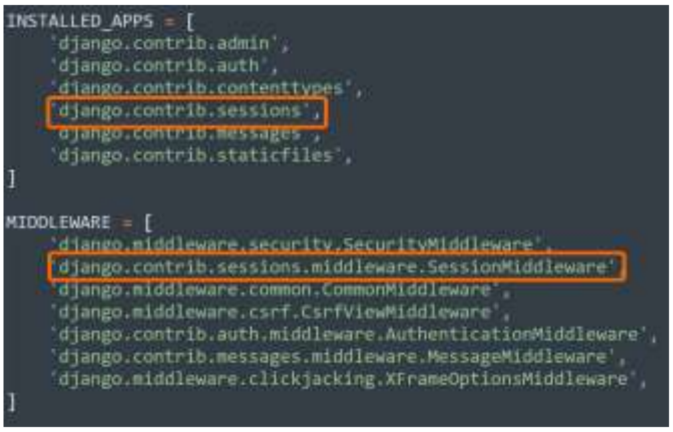
Session API:

- The session is a two-way, semi-permanent communication between the server and the browser.
- Let’s understand this technical definition in detail.
- Here semi means that session will exist until the user logs out or closes the browser. The two-way communication means that every time the browser/client makes a request, the server receives the request and cookies containing specific parameters and a unique Session ID which the server generates to identify the user.

- The Session ID doesn’t change for a particular session, but the website generates it every time a new session starts.
- When a session ends, Important Session Cookies containing these Session IDs are usually deleted. However, this will have no effect on cookies that have a set expiration date.
- Creating and creating sessions in a secure manner can be a difficult task.
- Now we'll have a look at Django's version of the same.

Sessions Django

- Django recognises the importance of website sessions and provides middleware and an inbuilt app to help you manage them.
- This will assist you in quickly generating session Ids hassle.



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

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',
]
```

- django.contrib.sessions is an application which works on middleware.SessionMiddleware and is convenient to work.
- The middleware.SessionMiddleware is responsible for generating your unique Session IDs.
- You will also require django.contrib.sessions application, if you want to store your sessions on the database.
- When we migrate the application, we can see the django_session table in the database.
- The django.contrib.sessions application is present in the list of INSTALLED_APPS in settings.py file.

Sessions expire when the user closes the browser:

- To work on this task, we need to add one line our ‘settings.py’ i.e. SESSION_EXPIRE_AT_BROWSER_CLOSE By default, the value of SESSION_EXPIRE_AT_BROWSER_CLOSE is False.
- So, we need to add this line into ‘setting.py’ file and set it to ‘True’ .
- Close the session when user closes the browser
- SESSION_EXPIRE_AT_BROWSER_CLOSE = True

Sessions expire after a period of inactivity:

- To work on this task, we need to add one line our ‘settings.py’ i.e. SESSION_COOKIE_AGE
- By default, the value of SESSION_COOKIE_AGE is 1209600.

- So, we need to add this line into ‘setting.py’ file and set it to ‘5 * 60’(5 minutes).

```
# Sessions expires after a period of inactivity
SESSION_COOKIE_AGE = 5 * 60
```