Django

27 October 2023 06:01 PM

Software:

→ Software is nothing but set of instructions or collection of programs which are used for either performing specific task or for operating a system.

Program:

→ Program is a set of instructions.

Instructions:

→ Set of Rules & Regulations that will make Computer to do a Task

Classification of Software's:

→ Software's are classified into 2 types

1. System Software
2. Application Software

System Software:

→ Software which is responsible operating and maintaining a system is known as Software System.

or

→ Software which is responsible for creating a platform where we can run the other software's.

Ex:

- Operating Systems like windows, Linux, macOS etc…

Application Software's:

→ Software's which are responsible for performing a specific task

Classification of Application Software:

→ Application Software's are classified into 3 types

1. Stand-alone Software
2. Web application software
3. Client-server application software

Stand-alone Software's:

→ Software which is not dependent on either Network connection or

Web Application Software's:

→ Software's which accessed through Web browsers

Ex:
Google chrome, Facebook, etc…..

Client-server Software's:

→ Client-server software is a distributed application structure which are used for dividing the tasks b/w Service providers(servers) and service requesters(Clients)

Ex:
Email, World Wide Web etc…..

Software Architecture:

→ Software Architecture is a Design Pattern in which we will create a software.

Classification of Software Architecture:

1. One-tier Architecture
2. Two-tier Architecture
3. N-tier Architecture

One-tier Architecture:

→ The Architecture in which there will be only one end or one phase. That type of Architecture is known as One-tier Architecture.
→ In this type of Architecture we will use only Front end technologies like HTML,CSS and JS for designing the application.
→ Modifications in the application can be done only by HTML page.
→ We can create only Static application by using One-tier

Ex:
Small School Websites

Two-tier Architecture:

→ In this type of Architecture we will be having two ends
1. Front End
2. Back End

→ We will use Front end Technologies for displaying content and we will use Python language in order to make some changes with the backend

→ Semi-Dynamic Web pages are created by using Two-tier Architecture.

Ex:
Medium scale industries

N-tier Architecture:

→ In this type we will be having 3 ends or phases.
1. Front End
2. Back End
3. Database Layer

Front End:

→ It is mainly used for displaying the content

Back End:

→ This end is used for receiving the data from DB and sending the data to front end

Database:

→ This layer is used for storing the received data

→ We can create Dynamic web applications by using N-tier Architecture.

Ex:
Pubg, Instagram etc.

Frame-work:

→ The main goal of the framework is to allow developers to focus on the components of the application that are new, instead of spending time on already developed components.

→ Simply we can say that framework is used for providing a prototype.

→ Frame-work is Software which is designed for supporting the development of the application

Web-Frame-work:

→ Which are responsible for developing Web applications

Classification of frameworks:

→ Frame-works are classified into Two types
1. Major Frame-work
2. Micro Frame-work

Micro framework:

→ By using Micro frame-works we can create application which is suitable for small scale industries like schools and colleges

drawbacks of micro-frame work:

→ Application cannot be accessed by many people at same time

Example of Applications Built by Micro Frameworks:
- School and College websites

Major frame-work:

→ By using Major frame work we can create an application which is suitable for large scale industries

→ Many people can access the application at same time

Example Applications Built by Major Frameworks:
- YouTube, pubg, fb etc.....

- Frame-works of different programming/scripting languages:

JAVA:
SPRING HYBERNATE STRUCTS
C#:
.NET
PHP:
LARAVEL PHALCON

PYTHON:

Major frame-works:

Django pyramid

Micro frame-works:

Flask

Some of the apps designed by using python:

1. Netflix
2. YouTube
3. Instagram
4. Dropox
5. Uber
6. Spotify
7. Reddit
8. Pinterest etc.....

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 to describe how the data and code segregation should be done while developing applications

→ 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 programs. this is used for displaying the content of database end to the user(FE)

Control:

→ Django frame work itself takes care of this controlling part

MVT:

MVC+Templates

Templates: It is used for dealing with HTML files

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

1. layout(html.css.js)
2. logic(python scripting 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 files as a response to the request sent by the client(user)

Installation of Django:

→ Go to command prompt and do the below operation

→ In order to make use of Django framework first we have to install Django package by using command shown below

pip install django ----> it will install latest version of django

pip install django==2.0

→ Make sure that your system is having internet connection while installing Django

VIRTUAL ENVIRONMENT:

→ Virtual environment is used for dividing a system into number of isolated brand new systems

→ Purpose of creating isolated parts is to avoid affecting of downloaded software versions to the entire system

Process to create Virtual Environment:

→ First we have to install virtualenv package

pip install virtualenv

→ Create a virtual environment by using below syntax

virtualenv name_of_the_virtualenvironment

→ Immediately after creating virtualenv one brand new system will be created

→ Brand new system which is created will be having the following files

1. Lib
2. Scripts
3. Environment configurations

→ We have to enter into that created virtual environment

cd virtual_env_name

→ We have to activate virtual environment

cd scripts

activate

→ Installing Django inside virtualenv(Because django software version should not be affecting entire system)

→ To turn off your activated virtual environment after use we have to run the following command

deactivate

Extensions that we have to install in VS CODE:

1. Click on extensions button
   i. Search for python
      Install python
   ii. Search for jinja
      Install jinja

pip freeze:

→ Pip freeze is responsible for viewing software's that we have installed along with the versions

Creation of Django Project:

→ Project is a container which is consisting of many number of features(applications)

Procedure to create a project:

→ Syntax for creating a project in django

django-admin startproject project_name

→ Immediately after executing above command one container will be created under project name

→ Inside this container we will be having two files

1. Project_name(folder)
   - One file is a folder which is having name same as your project-name
   - It is responsible for integrating the applications
2. Manage.py
   - It is managerial file which is responsible for controlling and triggering of the server(triggering is nothing but starting and stopping of server)

Information of files present inside the project folder:

1. __init__.py:

→ It is a blank python file which is used for making or converting folder with collection of modules into a package

2. asgi.py:

→ asgi stands for Asynchronous server gateway interface

→ asgi is used for creating standard interface between framework, project and web servers

Interface:

→ It is a place or platform by using which people or servers interact with each other

3. Settings.py:

→ It is a file where we will be making all the changes with respect to your project, application, database, static Files and templates etc...

4. urls.py:

→ urls.py file is responsible for performing the process of url mapping

url mapping:

- it is phenomenon of connecting or mapping the url with their respective function or class

5. wsgi.py:(web server gateway interface)

→ wsgi is used for hosting your application or software into the web.

Application:

→ application is nothing but the feature of the project

→ in single project we can have multiple applications

creation of application:

syntax for creating an application

python manage.py startapp app_name

→ information of the files present inside the application:

1. init

→ it is a blank file which is responsible representing a folder as package

2. apps.py:

→ it is a file where we will be doing all the application related configurations

3. models.py:

→ it is file where we create our models(tables)
models.py: creating tables

4. admin.py:

→ it is a file which is responsible for registering of created models
admin.py: registering of created tables

5. migrations:

→ it is used for storing the information related to models(tables) of the database

6. views.py:

→ it is a file where we write views(functions or classes) which are responsible for responding for received request.

7. tests.py:

→ it is file where we write our test cases which responsible for testing whether created functions or classes are working fine or not

Diagrammatic Representation of files of Project and application

syntax for running the server:
python manage.py runserver

1. immediately after running the server django creates one default url if there are no errors.
2. now copy the url and search the same in the browser
3. after execution we can stop the server by using ctrl+c

default database of django:
1. django will have sqlite3 database as default database
2. sqlite3 is a light weight database
3. immediately after running the server db.sqlite3 file will be activated inside the project folder

- Url>Uniform Resource Locator

information about the parts of url:
http://127.0.0.1:8000/suffix
url is a combination 4 important things

protocol://BaseAddress:port_number/suffix

protocol:

→ it is used specifying the standard rules that we have to follow for data processing and communication.

<u>http:</u>
1. it is unsecured and opensource protocol
2. We cannot process large amounts of data
3. Used only while developing

<u>https:</u>
1. it is secured and payed protocol
2. We can process large amounts of data
3. Used while developing and after deploying

<u>baseurl:</u>
1. it is logical address of the server
2. as u r running the server in your system django itself creates one virtual address that is 127.0.0.1

<u>portnumber:</u>
1. it is address of the channel through which the controller enter into the server
2. default django port_number(channel address) is 8000

<u>suffix:</u>

→ suffix is used for navigating to a particular page from a multiple pages

→ there r 2 types of suffixes:

1. primary suffix
2. secondary suffix

Extensions to be installed in VS code:

1. Python
2. Jinja

Code .

→ it is command that is used for carrying the current paths content

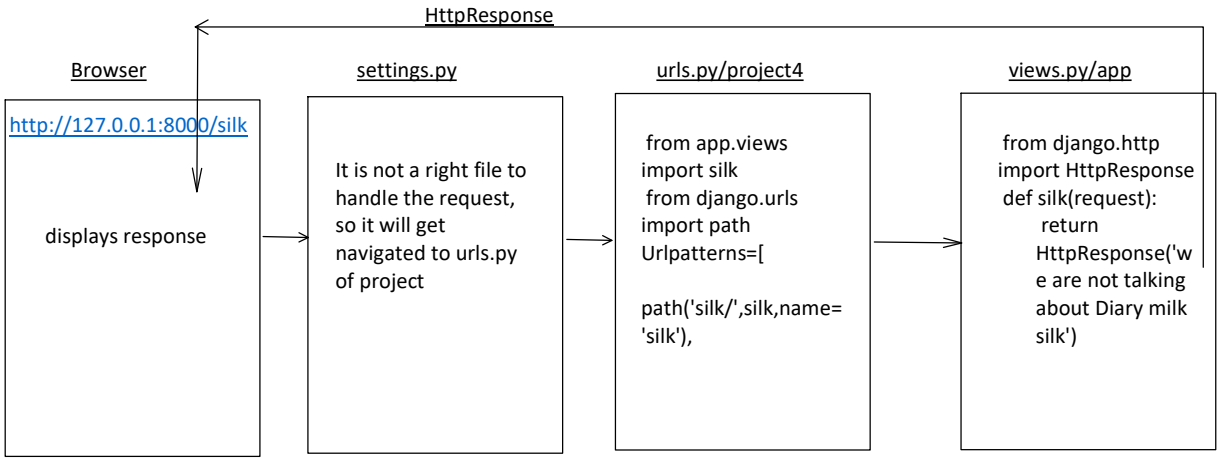
Procedure to connect app to project and create urls, views:

1. Syntax for creating an app
   python manage.py startapp app_name
2. Procedure to register app with project
   - Go to settings.py of project
   - Scroll down until you find INSTALLED_APPS VARIABLE
   - Now register your app by specifying the name as a element of the list.
3. Procedure to create views
   - There are 2 types of views
     1. Function-based view
     2. Class-based view
   - Go to views.py file of application
   - Create a function which is accepting request as first argument
   - In order to respond back we need HttpResponse
   - So import HttpResponse from django.http

   Sample example of view:

   ```
   def function_name(request):
       return HttpResponse('data that we have to respond')
   ```
4. Procedure to create urls:
   - Go to urls.py file of project
   - We can create urls by using path function
   - Syntax of path function:
     path('suffix/',address of the function, name='name_of_the_url')
   - Inside the url patterns list only we have to create urls

   Diagrammatic representation of flow of execution of URL:



Version Control Systems:

→ These are used for dealing with the different versions of the software development Process
→ Classification of Version Control System:
→ Generally Version Control systems are classified into Two Types

1. Centralized Version Control System
2. Distributed Version Control System(Mercurial,Bazaar, And GIT)

Git:

→ it is a Distributed version control system which responsible for storing and managing code we r writing

github:

→ git hub is a cloud based open-source software which used for hosting and managing the git repositories
→ Developers use this git-hub for code transmission purpose

git-repository:

→ it is folder that we r creating in memory which is provided by github.com

Procedure to upload projects in github:

1. go to the browser and search for below url https://git.scm.com/downloads
2. click on download 2.29.2 version
3. search for the url below https://github.com/
4. signup into github by providing your details
5. after signing up sign in into the git hub through browser
6. click on +(plus) symbol and click on new repository
7. give a name for the repository and click on create repository
8. go to the GIT command prompt by carrying path of the file which u need to upload in git

run the following commands:

1. git init --> initializes a local git-memory inside our local system
2. git add. --> it is used to connect the folders and files which r present in the current cmd path into git memory
3. git commit -m "first commit" --> by using above command we can save the connected files in our local git memory
4. git branch -m main --> IT IS USED FOR CREATING ONE BRANCH WITH NAME main

if u want to change the name use below syntax
git branch -m branchname

5. git remote add origin https://github.com/harshadvali11/first_project.git this above command is used for creating a repository(folder) Inside github.com
6. git push -u origin master

this command is used for pushing the saved connected files of git memory of local system into the github.com

Template Directory:

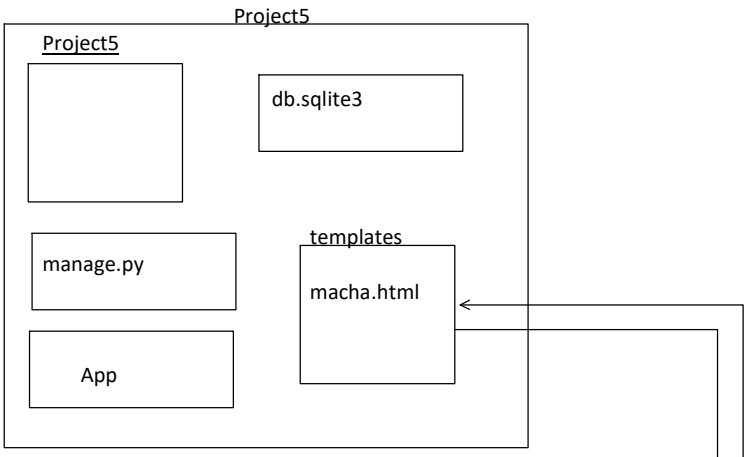
→ Templates directory is specifically used to store only html files
→ There are 2 types of templates directory

1. Generic template directory
2. Specific template directory

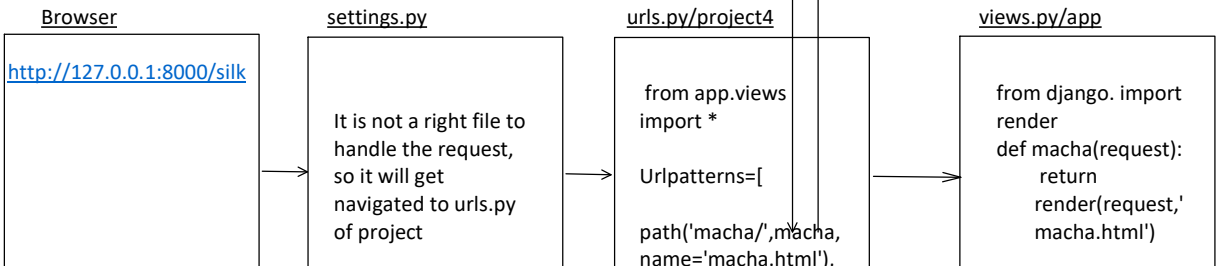
1. Generic template directory:

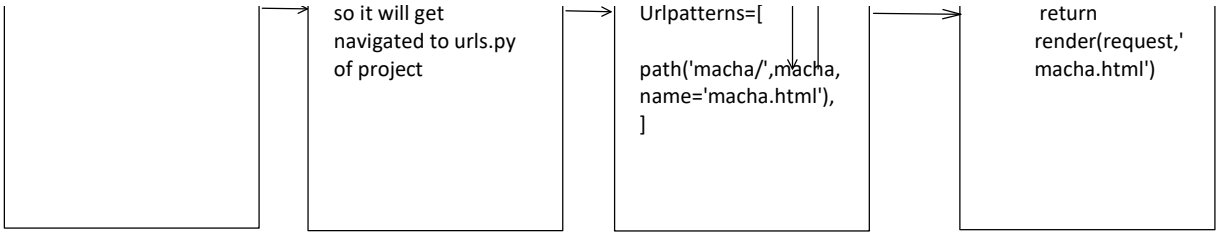
→ If we create template directory inside main project that type of template is known as Generic template directory.

Diagrammatic representation of Generic template directory:



Flow of execution of URL in case of rendering HTML page:





procedure to create and register templates directory:
create a project
create an application and register in settings.py
create a templates directory by command mkdir templates
procedure to register templates:
go to settings.py
create TEMPLATE_DIR Variable and store the path of templates as value
There are three methods how we can add path of a templates folder.

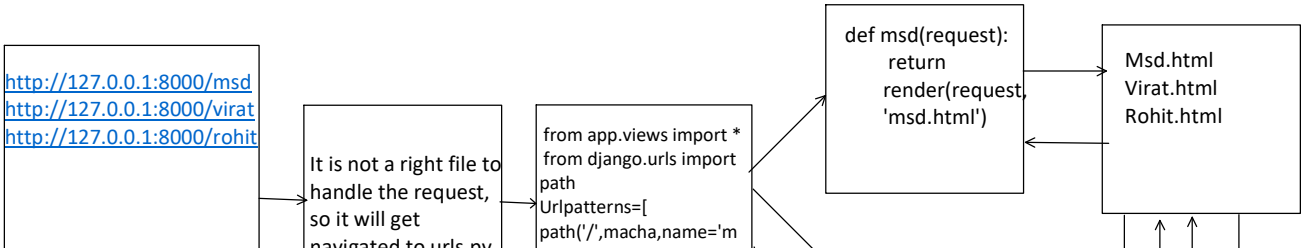
Method 1:
create TEMPLATE_DIR variable
copy the path of templates folder
paste it as a string value for TEMPLATE_DIR
change all the single slashes to double slashes
register TEMPLATE_DIR in DIRS key of TEMPLATES
Drawbacks of this method:
this method will not work if we r storing the templates directory in different path other than we specified.

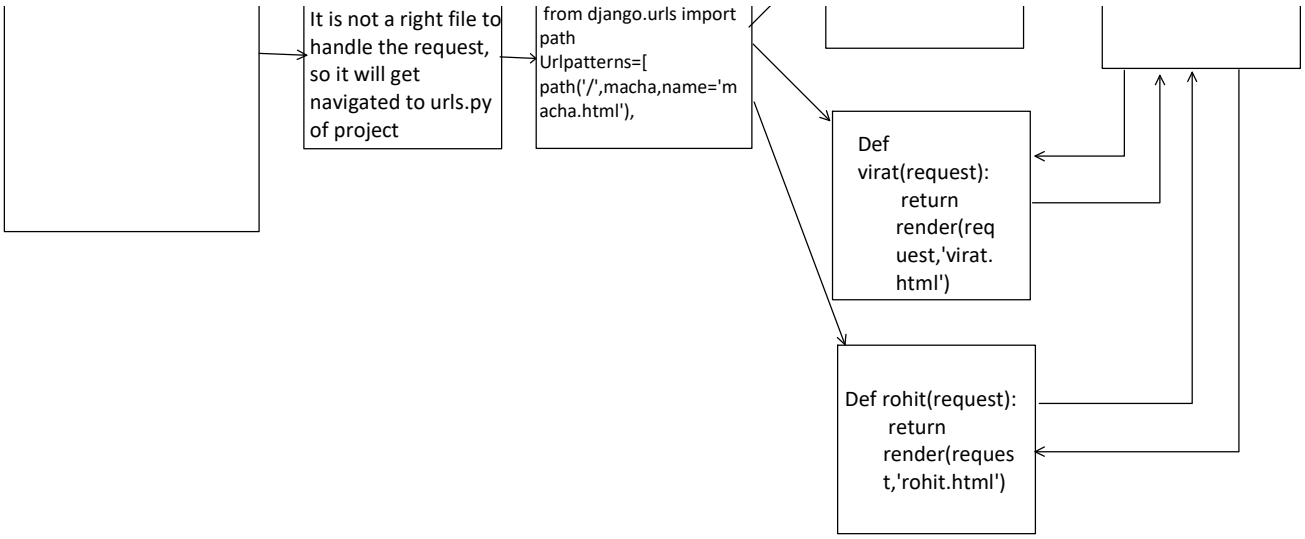
Method_2:
Concatenation of path of BASE_DIR with templates
1.as BASE_DIR gives the path till the main outer project independent of path where we save it.
2.so we can represent path as shown below
TEMPLATE_DIR = BASE_DIR/ 'templates'
register TEMPLATE_DIR in DIRS key of TEMPLATES
Drawbacks of this method:
This representation of path will not work in all type of operating systems because, we use forward slashes for representing path in windows and backward slash in Linux.
So, we go for representation of path by using OS module.

Method_3:
Using OS module
1. This is the universal way of representing path
2. In real time we use this method only
Import os TEMPLATE_DIR= os.path.join(BASE_DIR, 'templates') register TEMPLATE_DIR in DIRS key of TEMPLATES
creating html files:
select and right click on the templates folder
click on new file
and give the name of the html file with extension(ex:h1.html)
and write the content by using html tags
this is how views.py looks like while we render files:

```
from django.shortcuts import render
def function(request):
    return render(request, 'htmlfilename.html')
```

Creation of Multiple Applications:
→ In this scenario we will create multiple applications in a single project.
Flow of execution of URL in case of Multiple applications





Disadvantages of Generic URL mapping in case of multiple applications:

→ We cannot define a function with same name in more than one application
→ Code complexity will be more
→ Difficult to debug the URL mapping
→ While performing URL mapping time complexity will be more
→ In order to avoid above disadvantages we will create specific URLs in case of multiple applications

Specific urls.py:

→ Specific urls is nothing but creating an individual urls.py inside the application
→ To create urls.py file right click on app and choose new file and save as urls.py

Content we need to write in specific url file:

1. Import path
2. Import views of specific application
3. Urlpatterns list to be created
4. Specify value for app_name

Url representation of specific urls:

Syntax:

http://127.0.0.1:8000/primary_suffix/secondary_suffix

→ Primary_suffix is used to navigate the control from project urls to app urls
→ Secondary_suffix is used to navigate from app urls to views of application

Ex:

http://127.0.0.1:8000/app1/read

URL: Uniform Resource Locator

→ We use include function to connect specific urls.py with generic urls.py

```
import include from django.urls
    from django.urls import include
```

syntax of include:

```
    path('suffix/',include('app_name.urls'))
```

Sample content of urls.py looks like below:

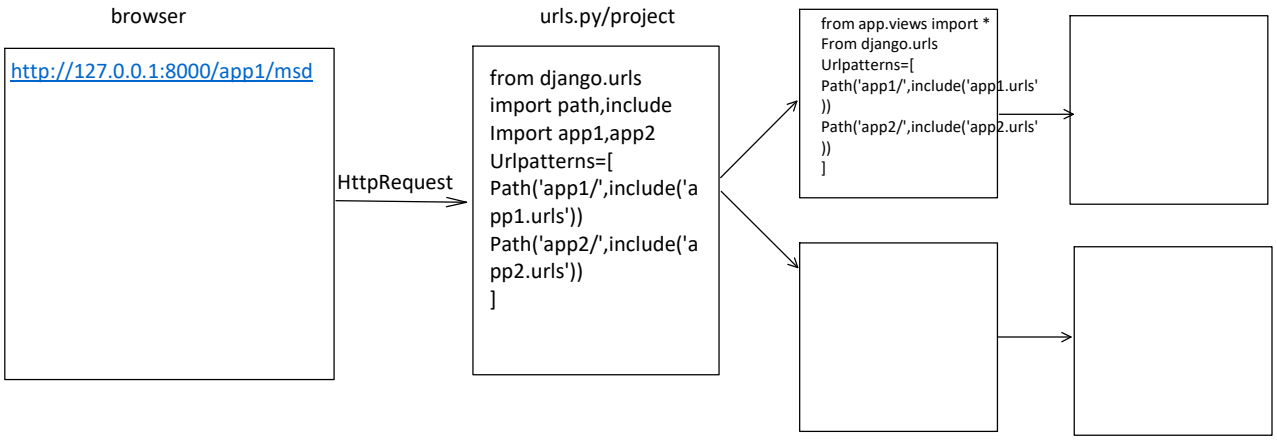
Urls.py of project:

```
from django.contrib import admin
from django.urls import path,include
import app
urlpatterns =[
    path('admin/',admin.site.urls),
    path('app/',include('app.urls')),
   ]
```

Urls.py of app:

```
from django.urls import path
from app import views
app_name='app'
```

```
Urlpatterns = [
      path('function1/',views.function1,name='function1'),
     ]
```



jinja tagging:

→ jinja tagging is a modern day tagging(template) language for python developers

or

jinja tagging is an advance level writing tags in html files
jinja tags are also known as expression tags or template tags

Classification of jinja tags:

→ Based on the operation we r doing with jinja tags these r classified into 2 types

1. tags used for printing the data which is sent from BE to FE
2. tags that are used for performing some operations

1. Tags used for printing the data:

→ By using this type of tags we can render some content from BE and represent same rendered data in the webpage

Representation of jinja tags used for printing data:

```
{{data}}
```

Procedure:

1. Go to html file represent jinja tag as shown below
   Syntax: {{variable_name}} or {{key_name}}
2. Go to views.py create a dictionary inside the function and pass it as value for the context attribute
   Syntax:

```
def fun_name(request):
    dict={'variable_name':'value of variable'}
    return render (request,'html file name',context=dict)
```

3. Usage of context attribute is optional
   Ex:

```
dict={'variable_name':'value of variable'}
def fun_name(request):
    return render(request,'html file name',dict)
```

Jinja tags that are used for performing some operations:

→ Whenever we want to perform some operations like inserting image, creating hyperlinks we go for this type tags

Representation jinja tags for operation:

```
{% expression %}
```

Operations that can be done by jinja tag are as follows:

1. Url navigation
2. If conditional statements
3. Looping statements
4. Insertion of images
5. Template inheritance

If - conditional statements by jinja tags:

Syntax for simple if condition:

```
{% if condition %}
    Statements
{% endif %}
```

Syntax for simple if - else condition:

```
{% if condition %}
    Statements of if
{% else %}
    Statements of else
{% endif %}
```

Syntax for simple if - elif - else condition:

```
{% if condition %}
    Statements of if
{% elif condition %}
    Statements of elif
{% else %}
    Statements of else
{% endif %}
```

Syntax for nested - if condition:

```
{% if condition %}
    {% if condition %}
        Statements of if block
    {% else %}
        Statements of else block
    {% endif %}
{% else %}
    {% if condition %}
        Statements of if block
    {% else %}
        Statements of else block
    {% endif %}
{% endif %}
```

Looping Statements:

→ We cannot perform while looping with jinja tags
→ We cannot perform for loop with range, range & CDT

Syntax for for loop with CDT(DERIVED DATATYPES):

```
{% for var in CDT %}
    STATEMENTS OF FOR LOOP
{% endfor %}
```

Url navigation:

→ It is a process navigating from one url to another url
→ We can achieve url navigation by using Anchor tag

Representation of href value for generic url file:

```
<a href='{% url "name_of_url" %}'> content </a>
```

Representation of href value for specific url file:

```
<a href='{% url "app_name_value:name_of_url" %}> content </a>
```

Sample body content of html url navigation:

```
<body>
    <h1> This is h1.html file </h1>
    <h1> Welcome Mr./Ms.{{name}}</h1>

    <a href="{% url 'h2' %}"> click on this to go to h2.html file</a><br>

    <a> href="{% url 'raina:h3' %}"> click on this to go to h3.html</a>
```

Creation of static folder:

→ In order to store images, js properties and css properties we use static folder
→ As content inside this will not change so we call this folder as static

We can create a static folder in 2 ways:

1. Generic static folder
2. Specific static folder

1. Generic static folder:

→ In this static folder all the images, js, css properties of all applications will be stored

Steps to create and register static folder:

- create a project
- cd project
- mkdir static
- cd static
- mkdir images.
- mkdir css
- mkdir js
- cd..

Registering generic static folder:

→ Go to settings.py
→ Create variable STATIC_DIR and store the path of static as a value
Ex: STATIC_DIR=os.path.join(BASE_DIR,'static')
→ Create a list-variable with a name STATICFILES_DIRS and do as shown below
Ex: STATICFILES_DIRS=[STATIC_DIR]

STEPS TO CHECK WHETHER STATIC FOLDER IS CONNECTED OR NOT

python manage.py runserver
http://127.0.0.1:8000/static/images/imagename.extension

→ We have to run the above url
→ If it is displaying the image then static folder is connected successfully

Insertion of images into webpages:

→ We use img html tag for inserting images

Syntax for inserting image in django is

```
<img src='{% static "images/image_name.extension" %}'>
```

→ We will get an error bcoz we have not loaded static files into the html file
→ So in order to avoid this, load the static file into html by using below command

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

Specific Static folders:

→ In any project css and js properties will be same for all the applications
→ So we will have css, js properties inside projects static folder
→ We will have images inside the app static folder

Steps to create and register specific static folder:

- Create a project, app
- Create a static folder inside inner project to store css, js
- Create another static folder inside an app to store images

Go to settings.py file

1. Register app
2. Register templates
3. Create a variable with a name STATIC_DIR_PROJECTNAME to store the path of project static folder
   Ex: STATIC_DIR_PROJECT16=os.path.join(os.path.join(BASE_DIR,'project16'),'static')
4. Create a variable with a name STATIC_DIR_APP to store the path of app static folder
   Ex: STATIC_DIR_APP=os.path.join(os.path.join(BASE_DIR,'app'),'static')
5. Create a list variable STATICFILES_DIRS to store the above variables
   Ex: STATICFILES_DIRS=[STATIC_DIR_PROJECT16,STATIC_DIR_APP]
6. In order to collect and store all the css, js and images from specific static folders we need to create one root static directory
   Ex: STATIC_ROOT=os.path.join(BASE_DIR,'static')
7. After registering we have to run below command to collect the static files
   python manage.py collectstatic

   Note:

→ In case of specific static directories we should not add static properties directly into root static folder

Bootstrap:

→ Bootstrap is a CSS framework
→ Bootstrap has some pre-defined classes with which we can add some CSS features
→ Bootstrap is a CSS framework in which the developers have given default properties for each and every tags
→ We can make a webpage more responsive by using bootstrap

Procedure to link bootstrap to HTML files:

In 2 ways we can connect Bootstrap to our HTML files

1. CDN method
2. Download method

2. CDN method:

→ CDN stands for CONTENT DELIVERY NETWORK METHOD

- Go to Bootstrap.com url
- Click on getstarted
- Copy the starter template in which they have connected Bootstrap css and js properties
- Paste the same in our respective HTML file

Disadvantages of CDN Method:

→ Network connection must be present
→ CDN cache will be cleared if we are not using particular content regularly
→ There may be a chance of blocking CDN when you block spam content

Download Bootstrap to our project:

- Go to Bootstrap.com
- Click on download, bootstrap download page will be opened
- There will be download button click on it
- Immediately one bootstrap properties ZIP file will be downloaded
- Copy the css properties of zip file and paste it in css folder of your project

Connecting css and js properties:

→ We use link tag to connect css properties
→ We use script tag to connect js properties

Ex:

```
<head>
<link rel="stylesheet" href={% static 'css/bootstrap-reboot min.css' %}>
</head>
<body>
    <script src="{% static 'js/bootstrap.bundle.js' %}"></script>
</body>
```

MDB:

→ MDB stands for Material Design Bootstrap
→ MDB has built in Bootstrap framework

Steps to download MDB4(jQuery):

- Go to browser and search for below url
  http://mdbootstrap.com/
- Click on getstarted
- Scroll down and click on download bootstrap4 with jQuery zip

→ After downloading open zip file and enter into css and copy and paste the following files into css folder of your project

- bootstrap.css
- Mdb.css

→ From js copy below mentioned files

- bootstrap.js
- mdb.js
- jquery.js
- popper.js

→ Steps to download MDB5(plain java script):

- Go to browser and search for below url
  http://mdbootstrap.com/
- Click on getstarted
- Scroll down and click on download bootstrap5 with plain java bootstrap5 script zip

→ After downloading open zip file and enter into css and copy paste the following file into css folder of your project

- mdb.min.css

→ From js copy below mentioned files

- mdb.min.js

Note point:

→ It is difficult to connect css and js properties to each and every html files. So in order to avoid that we have a concept called Template-Inheritance

Template-Inheritance:

→ It is a process of Inheriting the properties of one html file into another html file

Advantages:

→ It will reduce code redundancy and decreases code complexity

→ With in less time we can build our html content

We use extends jinja to inherit the properties

Syntax:

```
{% extends 'html_file_name.extension' %}
```

Ex:

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

Problem:

→ If we extend, then all the parent html content will be copied to child html along with the title

→ So in order to avoid that we use jinja block tags to stop inheriting all the properties

Jinja tags syntax for blocking title:

```
<head>
    {% block title %}
        <title> parent.html </title>
    {% endblock %}
</head>
```

Jinja tag syntax for blocking body content:

```
<body>
    Content which you need to inherit
    {% block body_block %}
        Content which you need to be inherited
    {% endblock %}
        Content which you need to inherit
</body>
```

Note:

→ We cannot have morethan one body_block in a single html file

→ It is mandatory to have empty body_block in case if you want to inherit all the properties into child and add new properties in child

n-tier architecture:

→ In this type of architecture we will be having more than 2 layers, i.e., our application will be having the following ends or phases

1. FRONT END
2. BACK END
3. DATABASE END

→ In development language we call the database end as Model Layer

MODELS:

→ Popularly we can store the data in 2 formats

1. Tables
2. Document format

→ In our Django frame work we will store the data in the form of Tables

→ As it is relational DBMS we have to use languages like MySQL, PostgreSQL, SQL etc. to manipulate the data

→ As it is difficult to learn SQL and MySQL or PostgreSQL, Django has come up with the concept of ORM to manipulate the data

ORM(OBJECT RELATIONAL MODELLING):

→ It is a concept which is responsible for writing Database Queries in the form of Python OBJECT ORIENTED PROGRAMMING

1. We can use any type of databases
2. By default Django provided sqlite3 database
3. We can connect other databases instead of sqlite3 database just by pip installing it and making some changes in the database section of our settings.py

Creation of database tables:

→ In order to create a table by using ORM we have to follow below considerations

→ Considering

1. 1 Class in ORM ----------> 1 Table in Database
2. 1 Class Variable in ORM ----------> 1 Column in Database
3. 1 Object in ORM ----------> 1 Row of Data in Database

Steps to create Models:

- Go to models.py of app
- Each and every model class must be inherited from Model class as shown below
- Create a class which represent as table_Name with a syntax given below

```
Class table/model_name(models.Model)
    Column_Name = models.fieldname(arguments)
```

Above created

→ Class member is used to represent column_Name
→ Field Name is used to represent Data Type
→ Arguments will be acting as constraints

We have to create a method with a name __str__ which is used for returning a string value

Giving access of models to admin:

- Go to admin.py file of app
- Import all the models into admin.py
- Register the models with below syntax

```
admin.site.register(modelname)
```

Storing the models into our Database:

- Go to command prompt
- Python manage.py migrate
  - It is used to check what all the models are present
- Python manage.py makemigrations
  - It is used for converting object oriented classes we have created into database language
- Python manage.py migrate
  - It is used to create actual tables into the database after conversion by ORM parser

Creating Super user:

→ In order to create superuser, run the following command in cmd
- python manage.py createsuperuser
- Asks for username:
- Email:
- Password:
- Re-enter password:
- We have to confirm the details, so press 'y'

→ With this superuser will be created

→ Runserver and go to browser and search for below url
http://127.0.0.1:8000/admin/
→ Admin interface will be opened give username and password

	deptNo.	Dname
X111	10	SALES
X222	20	ACCOUNTS
X333	30	RESEARCH
X444	40	OPERATION

	empno	Ename	Deptno
Y111	1234	A	X111
Y222	1231	B	X222
Y333	1232	C	X111
Y444	1432	D	X222
Y555	1321	E	X333
Y666	1244	F	X111

Note:
In SQL:
→ In child table parent tables' primary key value will be stored
In Django:
→ In child table parent tables' object will be stored(not primary key value)

Operations we can do with the tables:

1. Insertion of data
2. Retrieving of the data
3. Deleting the data
4. Updating the data

◆ Insertion of data:

→ We can insert the data into our models in 3 ways

- By creating an object and passing the values as key, values
- By create method

1. Creating an object to insert data:

Syntax:

1. ObjectName = ClassName(columnName1=value, columnName2=value,.....,columnNameN=value)
2. In order to save the inserted data we have to make utilize of save() method

Changing cmd into interactive shell:

→ Run the below command to enter into interactive console

```
python manage.py shell
```

→ Run the below command to come out of interactive console

```
exit() or ctrl+z
```

2. Inserting data by using create method:

Syntax:

1. ObjectName=className.objects.create(ColumnName1=value,col2=value)
2. In order to save permanently use save method

```
objectname.save()
```

- When you use create method to insert data it will return a instance directly

3. Inserting data by using get_or_create method:

Syntax:

1. ObjectName=classname.objects.get_or_create(colname=value)[0]
2. In order to save use save method

```
objectname.save()
```

- When you use get_or_create method to insert the data it will return a tuple (instance,booleanflag)

Note:
get_or_create method is an universal method which we will be using to insert data in real time

Retrieving the data:

→ In many ways we can retrieve the data from tables by using ORM

1. Syntax for retrieving all the data:

```
modelname.objects.all()
```

Process to display the retrieved content:

- Create one html file
- Go to views import all the models
- Create a variable and store the values of Query set
- Ex: variablename=modelname.objects.all()
- Send the query set to the front end by using context
- Use jinja tags to display the context data

Syntax for retrieving single data or particular data:

→ There are 2 methods by using which you can retrieve particular data

1. get
2. filter

1. Get method:

1. Based on condition we can retrieve the data
2. It will return directly the object

Syntax:

```
modelname.objects.get(columnname=value)
```

Drawbacks:

1. Value must be a primary key
2. Get throws an error
   i. If None of the rows are satisfying the given condition
   ii. If morethan one row is satisfying the condition

2. Filter method:

→ It will return query set in the form of list

Syntax:
modelname.objects.filter(columnname=value)

Note:
→ It is an ideal method by using which we will retrieve the data from the models

Note:
→ By default data will be displayed in insertion order

Ordering of retrieved data:
Ascending order syntax:
Modelname.objects.order_by('columnname') #orders in Ascending order

Descending order syntax:
Modelname.objects.order_by(''-columnname") #orders in the descending order

Arranging in the an order based on the length:
First import Length function from django.db.models.functions import Length
from django.db.models.functions import Length

Syntax:
modelname.objects.order_by(Length('columnname')) #orders in the ascending order
gmodelname.objects.order_by(Length('columnname').desc()) #orders in the descending order

Exclude method:
By using exclude method we can remove the rows which are satisfying the given condition
Syntax:
Modelname.objects.exclude(columnname=value)

Field Lookups:
1. These are used for achieving where conditions more precisely
2. Field lookups starts with double underscore

Syntax:
Columnname__lookupname=value
3. This field lookups must be written inside the filter method

Method syntax representation:
modelname.objects.filter(columnname__lookupname=value)
different types of field lookups:
__gt ---> it is equivalent to greater(>) than symbol
__lt ---> it is equivalent to less(<) than symbol
__gte --> it is equivalent to the greater than or equal to (>=) symbol
__lte --> it is equivalent to less than or equal to(<=) symbol

Date format: 'yyyy-mm-dd'
Month lookup:
→ It is used for comparing only the months of a date
Syntax:
Columnname__month='value in number'

Year lookup:
→ It is used for comparing only the year of a date
Syntax:
yyyy----> 2000 ----> Columnname__year='value in number'

Day lookup:
→ It is used for comparing only the day of a date
Syntax:
dd ---> 20 --> columnname__day='value in number'

LIKE operator functionality can be achieved by using below commands:
startswith:
→ It is used for checking whether specified string is starting with specified value or not
Syntax:
modelname.objects.filter(columnname__startswith=value)

endswith:

→ It is used for checking whether specified string is ending with specified value or not

Syntax:

```
modelname.objects.filter(columnname__endswith=value)
```

In:

→ It is used for checking for multiple values of same column

Syntax:

```
modelname.objects.filter(columnname__in=(value1,value2,……..,valuen))
```

Contains:

→ It is used for searching specified characters are present inside the string or not

Syntax:

```
modelname.objects.filter(columnname__contains=value)
```

Regex:

→ It is used for comparing by using the patterns

Syntax:

```
modelname.objects.filter(columnname__regex=r'pattern')
```

Note: r---> represents Raw Pattern

Q Objects:

1. Q objects is mainly used for encapsulating Key word arguments
2. We use Q objects whenever we are dealing with complex conditions or multiple conditions
3. In order to utilize Q object we have to import it first
   from django.db.models import Q
4. We can combine more than one query set by using &, | operators
5. Then new query will be emerged by combining the represented queries

Syntax for using AND operator between 2 conditions:

```
modelname.objects.filter(Q(condition1) & Q(condition2))
```

Using of AND without Q object:

```
modelname.objects.filter(condition1,condition2)
```

Syntax for using OR operator:

```
modelname.objects.filter(Q(condition1) | Q(condition2))
```

Forms:

→ Forms are used whenever we want to take the data from the user through Front End

→ In case of django we can create forms in 3 ways

1. Normal HTML forms
2. Built-in django forms
3. Model forms

Normal HTML Forms:

→ In this type we will create text boxes by input and text area tags of HTML

Ex:

```
<form action='urlname/filename' method='GET/Post'>
    <input type= 'text'> name </input>
</form>
```

Action attribute:

→ By using this attribute we can carry the submitted to another url or HTML file

Syntax:

```
action = '{%url''urlname''%}'
```

Method:

→ It is used to specify how we have to carry the submitted from data

→ There are 2 ways in which we can carry the data:

1. GET method
2. Post method

GET method:

→ It is used when we are using input element in order search some data

→ Large amount data cannot be transferred

→ It is unsecure method

Post method:

→ By using this method we can carry the data in order to store into the DB large amount data can be transferred
→ It is secure method

CSRF - token:

→ It is used for sending the form data more securely.
Syntax:
{% csrf_token %}

Process happens after submitting the data:

1. By default same URL/function will be called
2. POST method will be activated once you submit the data
3. Submitted data will be collected under the dictionary
4. The name of the dictionary is request.POST

Sample of submitted data:

```
request.POST={'un':'Musician','pw':'suresh@'}
```

Procedure to access the form input elements data:

Sample html file:

```
<form method='POST'>
    <label for='username'>username</label>
    <input name='user' id='username'>

    <label for='age'>age</label>
    <input name='age' id='age'>
</form>
```

In 3 ways we can access the data from submitted Form:

1. By using keyname

```
request.POST[keyname]
```

2. Get_method

We use get method whenever we are accessing single data
Syntax:

```
request.POST.get('keyname')
```

3. getlist method:

- We use this method in order for multiple input
- Output format us a list

```
request.POST.getlist('keyname')
```

Inserting the values into the models by taking values through front-end

Views.py:

```
def create_web(request):
    topics=Topic.objects.all()
    if request.method=='POST':
        topic_name=request.POST.get('topic')
        name=request.POST.get('name')
        url=request.POST.get('url')
        t=Topic.objects.get_or_create(topic_name=topic_name)[0]
        t.save()
        w=Webpage.objects.get_or_create(topic_name=t,name=name,url=url)[0]
        w.save()
        #return HttpResponse('one record has been added')
        webpages=Webpage.objects.all()
        return render(request,'displaywebpage.html',context={'webpages':webpages})
    return render(request,'create_web.html',context={'topics':topics})
```

Create_web.html:

```
<form method="POST">
    {% csrf_token %}
     topic_name:

    {% if topics %}

    <select name="topic" id="topic">
```

```
        {% for topic in topics %}
            <option value="{{topic}}">{{topic.topic_name}}</option>
        {% endfor %}
    </select>

    {% endif %}
    Name:<input type="text" name="name" id="name"><br>
    Url:<input type="url" name="url" id="url"><br>
    <input type="submit" value="submit">
</form>
```

Getlist method:

1. It is used for accessing multiple input from the front-end
2. Output will be in the form of list

Syntax:

varname=request.POST.getlist(name_attribute_value)

Note:

By using select tag, checkbox we can access multiple data

Displaying multiple content:

In order to display multiple Query sets we have to concatenate them by using parallel pipe(|)

Syntax for creating empty Query set:

varname=modelname.objects.none()

Modifying the heading and title in admin page:

---

go to urls.py of project

1.changing the site heading value

admin.site.site_header='value'

2.changing the site title value

admin.site.site_title='value'

3.changing the sites index title

admin.site.index_title='value'

Django Built-in forms:

1.it is unique feature of django by using which we can create
html input elements by using Object oriented programming

2. here one class member will be mapped to one html input element

3.by using django forms we cannot achieve submit,action,method,form tag,csrf_token
operations

procedure to create django forms:

---

first create forms.py inside ur application

each and every class must be inherited from Form class we have to
import forms module from django package

```
from django import forms
```

sample class representation:

```
class Classname(forms.Form):
    attributename=forms.Fieldname(Arguements)
```

types of FormFields:

CharField:

1.it is used for defining a input element of text type
2.we can give alphanumeric values

IntegerField:

1.it is used for defining a input element of number type
2.we can give numeric values

UrlField():

1.it is used for defining a input element of url type
2.we can give only a perfect url

EmailField:

1.it is used for defining a input element of email type
2.we can give only a perfect email ids

ImageField:
---
1. by using this field we can insert images

FileField:
---
1. by using this we can insert files

DateField and DateTimeField:

1.DateField:yyyy-mm-dd
2. DateTimeField:yyyy-mm-dd hr:min

TimeField:
---
1. it is used for defining Time in 24 hr format

ChoiceField:
---
by default it will create a dropdown list,which enables u to select
only one option
syntax:
---
attributename=forms.ChioceField(choices= tuple or list of values)

example for choices:

Listval=[('Male','male'),('female','female')]

(firstvalue,secondvalue)

Firstvalue------->this value will be stored into ur database
Second value------> this is the value u need to display in the front end

MultipleChoiceField:
---
by default it will create a dropdownlist,which enables u to select
multiple options
syntax:
---
attributename=forms.MultipleChioceField(choices=list or tuple of values)

password:

1.we dont have separate field for password,so we will convert

CharField into password by using widget.
syntax:
---
password=forms.CharField(widget=forms.PasswordInput)

TextArea:
---
syntax:
---
textarea=forms.CharField(widget=forms.Textarea(attrs={'cols':3,'rows':3}))

Radio Button:
---
we achieve radio buttons by converting ChoiceField by using widget:

radio=forms.ChoiceField(widget=forms.RadioSelect)

checkbox:
---
syntax:
---
checkbox=forms.MultipleChoiceField(widget=forms.CheckboxSelectMultiple)

how to render form from backend:
---
as we r rendering normal content from backend to front end in the same way
we will render the forms too.

after creating class in forms.py of app

go to views import forms from app

create function and inside it create a
instance of class and pass it front End as Context as shown below

syntax:

```
def fun_name(request):
    instance=forms.Classname()
    d={'instance':instance}
    return render(request,'htmlfile',context=d)
```

We can display form instance in html in 4 ways
---

1. {{ form }} will render them as normal representation

2. {{ form.as_table }} will render them as table cells wrapped in <tr> tags

3. {{ form.as_p }} will render them wrapped in <p> tags

4. {{ form.as_ul }} will render them wrapped in <li> tags

Validating and Accessing of data:
---
1. form = NameForm(request.POST)

The above syntax is used for 'binding data to the form'
(it is now a bound form).
2. Form class instance is having a Method called is_valid()

3. is_valid() method is used for running validation routines for all its fields

4. When this method is called, if all fields contain valid data,
it will return True and
place the form s data in its cleaned_data attribute.

5. if data is not Valid Then it will return False
and same html page will be opened

Demonstration of Above Procedure:
---

forms.py content

```
from django import forms

gender=(('male','male'),('female','female'))

#('what u have to carry as an output to database','What to display in form')

course=(('python','python'),('django','django'))

class New_Form(forms.Form):
   name=forms.CharField(max_length=10,min_length=5,label='LAST',label_suffix='NAME',help_text='Enter Ur Name')
   number=forms.IntegerField(max_value=100,min_value=25,required=False)
   url=forms.URLField(max_length=30)
   email=forms.EmailField(min_length=5)
   comments=forms.CharField(max_length=1000,widget=forms.Textarea(attrs={'cols':5,'rows':5}))
   password=forms.CharField(max_length=15,widget=forms.PasswordInput)
   gender1=forms.ChoiceField(choices=gender)
   gender=forms.ChoiceField(choices=gender,widget=forms.RadioSelect)
   courses=forms.MultipleChoiceField(choices=course)
   check=forms.MultipleChoiceField(choices=course,widget=forms.CheckboxSelectMultiple)
   time=forms.TimeField()# 24 hr format
   date=forms.DateField()#yyyy-mm-dd
   datetime=forms.DateTimeField()#yyyy-mm-dd hr:min
```

Views.py file content

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

def New_Form(request):
   form=forms.New_Form()#sending form input elments without data
   if request.method=='POST':
      form_data=forms.New_Form(request.POST)
      if form_data.is_valid():
         #print(form_data.cleaned_data)
         #return HttpResponse('form submitted successfully')
         #name=form_data.cleaned_data['name']#directly by keyname
         #number=form_data.cleaned_data.get('number')#using get method
         #url=form_data.cleaned_data.get('url')
         #email=form_data.cleaned_data['email']
         #gender=form_data.cleaned_data['gender']
         #courses=form_data.cleaned_data['courses']
         #check=form_data.cleaned_data['check']
         time=form_data.cleaned_data['time']#24 hr format
         date=form_data.cleaned_data['date']
         datetime=form_data.cleaned_data['datetime']
         #d={'name':name,'number':number,'url':url,'email':email,'gender':gender,'courses':courses,'check':check}
         d1={'time':time,'date':date,'datetime':datetime}
         return render(request,'formdata.html',context=d1)

   return render(request,'New_Form.html',context={'form':form})
```

form.html content:

```
<body>
   <center>
      <h1>displying forms.py file content</h1>
      <form method="POST">
         {% csrf_token %}

            {{form.as_p}}
```

```
        <input type="submit" value="submit">
      </form>
  </center>
</body>
```

form_data.html

```
<body>
  <center>

      <h1>displaying form data</h1>
      <h1>{{name}}</h1>
      <h1>{{number}}</h1>
      <h1>{{url}}</h1>
      <h1>{{email}}</h1>
      <h1>{{gender}}</h1>
      <h1>{{courses}}</h1>
      <h1>{{check}}</h1>
      <h1>{{time}}</h1>
      <h1>{{date}}</h1>
      <h1>{{datetime}}</h1>
  </center>

</body>
```

field Attributes:
---

attriute_name	functionality
label	used to provide a label
label_suffix	used to provide additional suffix for label
help_text	used to provide hint about the value u have to enter inside text box
required	helps to convert a field as mandatory or not
error_messages	helps to return user defined value while rising errors
validators	helps to register the validation functions

Validators
---

1. validators are set of instructions that are used for checking wheher form is working fine or not and data is correct or not

Classification:
---

there are 2 types of validators

1.User Defined validators or Custom Validators
2.Built in validators

1. Custom Validators:
---

we create Custom Validators in 2 ways
1. By using normal function
2. By Form class object methods

1. By using normal function:
---

define a function which is accepting the a value which is responsible for raising the Validation error based on a condition

after defining function connect with field by using validators attribute

example:

```
def check_for_a(s):
   if s[0].lower()!='a':
      raise forms.ValidationError('name should start with a')


class ContactForm(forms.Form):
   name=forms.CharField(max_length=15,required=True,validators=[check_for_a])
```

2. By Form class object methods
---
there are 2 Form class object methods
1.clean_element method
    we use clean_element method when we have to validate only a single
    Field(Botcatcher)
2.clean method
    we use clean method when we have to validate more than one field

example:

```
class ContactForm(forms.Form):
   botcatcher=forms.CharField(max_length=20,widget=forms.HiddenInput,
                                    required=False)
   Email=forms.EmailField(max_length=50)
   Re_Enter_Email=forms.EmailField(max_length=50)

   def clean_botcatcher(self):
      bot=self.cleaned_data.get('botcatcher')
      if len(bot)>0:
         raise forms.ValidationError('data is not entered by human')

   def clean(self):
      e=self.cleaned_data.get('Email')
       r=self.cleaned_data.get('Re_Enter_Email')

      if e!=r:
         raise forms.ValidationError('emails not matched')
```

HiddenInput widget:
---
1. we use HiddenInput widget in order to check wheather data submitted is written
by Human or any other automated software
2. when we make a field as Hidden-input then that field will not be visible
  in front form so we can get the form data which is submitted by Human

Note:

It is advised to use clean_element method with only HiddenInput field(botcatcher)

Built_in validators:
---
in order to make use of this Built_in validators we have to import
valiadators class from django.core

    syntax: from django.core import valiadators

There are various kind of validators some of them are:

---
1. RegexValidator
2. MaxValueValidator
3. MinValueValidator
4. MaxLengthValidator
5. MinLengthValidator
6. validate_email
7. validate_url etc

syntax for representation of Built_in validators:

```
from django.core import validators

class class_name(forms.Form):
    attribute_name=forms.Field_name(constraints,validators=[validators.valida
                                                            torname()]
```

example:
---
forms.py content:

```
class ContactForm(forms.Form):
  phone=forms.CharField(max_length=10,min_length=10,validators=[validators.
                RegexValidator('[6-9]\d{9}')])
```

filters
---
filters are the functionalities which are used for performing some

operations on the data that we r sending through back-end to front-end.
2.we use (|) symbol to apply the filters

syntax:

```
{{variable|filter_name:arguements}}
```

Classification of filters:

Filters are classified into 2 types
1. Built_in filters
2. User Defined or Custom Filters

1. Built_in filters
these are created by django developers

Some of the Built_in filters are:
---
1. upper:
---
it is used to represent the data in UpperCase

example: {{variable|upper}}

---
2. lower:
---
it is used to represent the data in LowerCase

example: {{variable|lower}}

---
3. capfirst:
---
it is used to represent the first character of entire string on Uppercase
and remaining in the case how they r defined.

example: {{variable|capfirst}}

---

4. title:

---

it is used to represent the first character of each and every word
in Uppercase and remaining in lower case

example: {{variable|title}}

---

5. length:

---

it is used to represent the count of the number of characters in the
given string

example: {{variable|length}}

---

6. date:

---

it is used to represent the date in required format

example: {{variable|date:'dateformat'}}

Refer below link for date formats:

---

https://simpleisbetterthancomplex.com/references/2016/06/21/date-filter.html

7. truncatewords:

---

it is used to represent the specified number of words after that
(...) will be represented

example: {{variable|truncatewords:'nummber of words'}}

---

8. truncatechars:

---

it is used to represent the specified number of characters after that
(...) will be represented

example: {{variable|truncatechars:'nummber of characters'}}

---

9. slice:

---

1. it is used to represent the specified number of characters
2. functionality is same as trunccatechars but ... will not be represented

example: {{variable|slice:'nummber of characters'}}

---

10. pluralize:

---

1. it is used to pluralize the given word based on given value
2. other than 1 each and every value is considered as plural value only

example: {{variable|pluraq lize:'args'}}

---

if u r not passing any args By default pluralize add 's' to the word

1. if u want to add particular pluralization value :

example: {{variable|pluralize:'es'}}

---

2. if u want add either of the value based on count:

example: {{variable|pluralize:'arg1,arg2'}}

---

arg1---->it is displayed for singular values
arg2---->it is displayed for plural values

2. User Defined or Custom Filters
---
1. in_order to create Custom Filters we have to create a templatetags directory inside the application,and create __init__.py in order make it as a package
2. after that create a .py file in which we will write custom Filters

Diagramatic representation of Application:

```
application/
    __init__.py
    models.py
    templatetags/
        __init__.py
        usdfilter.py
    views.py
```

Content we have to write inside usedfilter.py file:
---
1. import template module from django
```
from django import template
```

2. Now create a template Library instance and store it in a register variable
```
register=template.Library()
```

3.Writing custom template filters
Custom filters are Python functions that take one or two arguments

arg1---->The value of the variable (input) not necessarily a string.
arg2---->The value of the argument this can have a default value
or be left out altogether.

example of custom filter with both value and arg:

```
def replace_char(value, arg):
  return value.replace(arg, 'value with we need to replace')
```

example of custom filter with only value:
---
most of the functions will not be taking arguements so

```
def swap(value): # Only one argument.
    return value.swapcase()
```

4. Registering custom filters
---
1. we have to register custom filters with django.template.Library.filter()
2. as we have already created library instance and stored in register variable this registering of custom filters can be done in 2 ways

1. normal way

The Library.filter() method takes two arguments:

arg1---->The name of the filter a string.
arg2---->The compilation function a Python function
(not the name of the function as a string).

example:

```
register.filter('rep',replace_char)
reister.filter('low',lower)
```

2. decorator method

```
@register.filter(name=nameof filter)
def functionname(value):
    return some value
```

5. How to load custom filters into html file

By using load jinja tag we can load custom filters

```
{% load udf_filter_file_name %}
```

Giving customize view to the admin:

1. In order to customize the admin interface we have Model admin class
2. In admin.py we have to create a class which is inherited from ModelAdmin
3. ModelAdmin class is present inside the admin module

Content of admin.py file:

```
from app.models import *
class WebpageAdminView(admin.ModelAdmin):
    #values for class members can be provided in the form of list or tuple
     list_display=['topic_name','name','url']
     list_editable=('name',)
     list_per_page=3
     list_display_links=['url']
     search_fields=['name','url']
     list_filter=['url','name','topic_name']

admin.site.register(Topic)
admin.site.register(Webpage,WebpageAdminView)
admin.site.register(AccessRecords)
```

Model Forms:-

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 inorder 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 the 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
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:
----
1. It is used to specify which all the Model attributes are to be represented as
   form input elements
2. fields values can be represented in list or tuple
      syntax: fields=value
      ----
      fields='_all_' ------>takes all columns of model
      fields=['attriute1','attriute2'.....'attriuten']
exclude:
----
1. It is used to specify which all the Model attributes are not to be represented
   as form input elements
2. exclude values can be represented in list or tuple
      syntax: exclude=['attriute1','attriute2'.....'attriuten']

Customization of ModelForm attributes:
----
1. Field types of Model are very sensitive while representing them as form field
2. so we can convert CharField of Model to TextArea of form as shown below

Textarea:

By using the TextArea class forms module we can add TextArea for charefield

3. help_texts,widgets and labels for the ModelForms can be added by using dictionary

Sample class representing above all properties:
----
forms.py content:

```
from django import forms
from app.models import *

class WebpageForm(forms.ModelForm):
  class Meta:
      model=Webpage
      fields='_all_'
      #fields=['topic_name','name']
      #exclude=['name']
      help_texts={'topic_name':'should not be integers','name':'only Alphabets'}
      labels={'topic_name':'TN','name':'N'}
      widgets={'url':forms.PasswordInput,'name':forms.Textarea}
```

User Registration:

1. Create a project (User Registration Form)
2. Create app
   Create templates
   Create media
   Go to vs code by code.

3. Go to settings
- Register templates
- For registering media we need to follow few steps:
  1. Create Url as:
        MEDIA_URL = 'media/'
  2. Create path as:
        MEDIA-ROOT =Os.path.join(BASE, DIR, 'media')
  3. Go to urls.py. MEDIAFILES_DIRS concatenate your MEDIA_URL and MEDIAFILES_DIRS by using static method and we also need to import settings into urls.py file as:
        from django.conf import settings
        from django.cong.urls.Static import Static
        urlpatterns = [----------

----------]+ Static (settings MEDIA_URL, document_root = settings MEDIA_ROOT)

4. Go to Models.py file in app
- Here we already have inbuilt user model so we just need to import user model as
  from django. contrib. auth. models import User
- Create Profile model with 3 columns
  username
  address
  profile pic
5. Register the models in admin.py.
  •Here first need to import models into admin.py as:
  from app.models import *
  • We only need to register Profile model, we don't need to register user model, because it is already an inbuilt model which will be registered by default
  admin site register (Profile)

6. Create forms: Go to forrns.py (create forms.py file in app folder)
   - Create UserForm and profile form.
   - While Creating Userform, we just need only usernami, email, password from many field.
   - Make the password invisible by using a widget.
     widgets = {'password': forms. PasswordInput}
   - While Creating ProfileForm, We just need to create columns address and profile pic. We DON'T need username. Because we are sending Userform Object and Profile form abject to a single view. So, already UFO will be there, so we don't want to use again.

7. Create views: Go to views.py file.
   - Create jet a function with name registration
   - Get Empty User form and profile form obj into views
   - Return this to 'registration.html' html page.
   - Once the data is submitted,
     1. Post method gets activated and gets all the files.
     2. collect all the data
     3. Perform Validation for both utd and pfd.
     4. Convert Non-Modifiable user form object into Modifiable form object by
        Ufd save (commit = False).

     This is done because we need to encrypt the password so we cannot do on direct form object
     Hence we have to convert it into modifiable.
     5. collect password & store it in one variable.
     6. Perform HASHING to password by using set_password method
        MUFDO set_password (pw)
     7. Save it
        MUFDO. Save()
     8. Convert Profile form object into modifiable because we only have 2 cols in forms byt 3 cols in models. So we need to add 1 col to it by
        MPFDO.Username = MUFDO
     9. Save it
        MPFDO.Save()
8. Go to urls.py:
   write Url Mapping for the view registration
9. Goto templates --> registration html
- Create User registration form.
- In form tag along with method also write enctype to store data into the database.
  <form method="POST" enctype="multipart/form-data's

Connecting MySql with django:

1. Download MySql installer by using below url
   https://dev.mysql.com/downloads/installer/
2. once u click on above url it will ask u to create a oracle account create it by giving some random details
3. after that log in into that and download with default settings
4. while installing it will install the following packages
   1. MySQL workbench
   2. MySQL commandline client
   3. MYSQL installer
   4. MySql notifier
   5. mysql-installer-web-community
5. After successfull installations search for mysql and click on mysql commandline client

6. it will ask u to enter password
    root (is the password)
7. syntax for creating database
    create database databasename;
8. In order to use that created database by using below command
    use databasename;
9. Syntax to see the tables of Database
    show tables;
10. syntax to retrieve the data
select * from Relationname;

Now we have to install Two python packages inorder to connect mysql to ur django project
    1. pip install pymysql
    2. pip install mysql-connector-python

Now Configuring with in the settings.py tfile of your project

```
import pymysql
pymysql.version_info=(1,4,6,"final",0)
pymysql.install_as_MySQLdb()

#now after this scroll down to database section
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'name of the database u have created',
        'USER':'root',
        'PASSWORD':'root',
        'HOST':'localhost',
        'PORT':3306,
      }
    }
```

Python postgresql

https://www.tutorialspoint.com/postgresql/postgresql_python.htm

PostgreSQL

1. PostgreSQL, also known as Postgres.
2. It is a free and open-source relational database management system
3. It has many features that safely store and scale the most complicated data workloads.

postgresql installation steps notesv

https://medium.com/@9cv9official/creating-a-django-web-application-with-a-postgresql-database-on-windowsc1eea38fe294

Installation steps

1. link for downloading postgresql
https://www.enterprisedb.com/downloads/postgres-postgresql-downloads
2. click on downloaded software
3. check the below mentioned softwares
    1. PostgreSQL server
        2. pgadmin 4
        3. Commandline tools and click on next
4. Now it will ask for password
    enter password
    re-enter-password and press next
5. leave the default port number as it is
6. With the above steps PostgreSQL is installed successfully

psycopg2:

1. in order to utilize postegresql we have to install below mentioned
    python package
    pip install psycopg2
2. psycopg2 is a database driver
3. psycopg2 sends SQL statements to the database.

Now Configuring postgresql with in the settings.py file of ur project

1. Go to settings.py file
2. now after this scroll down to database section

Syntax:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'DataBasename',
        'USER' : 'username',
        'PASSWORD' : 'usernamepassword',
        'HOST' : 'localhost' ,
        'PORT' : 5432,
    }
 }
```

example:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'harshad',
        'USER' : 'postgres',
        'PASSWORD' : 'nikky786',
        'HOST' : 'localhost' ,
        'PORT' : 5432,
        }
    }
```

postgradesql installation steps notes

https://medium.com/@9cv9official/creating-a-django-web-application-with-a-postgresql-database-on-windowsc1eea38fe294

syntaxes/symbols usage

1. create database databasename;	creates a database
2. \c databasename;	access to Database
3. \d,\dt	displays the database tables relations
4. \l	displays the databases of user
5. \l+	displays the databases of user with sizes
6. \?	displays each and every symbols along with usage
7. q	helps end or come back
8. \du	displays the list of users
9. \du+	displays the list of users along with description

Syntaxes:

1. For creating a database
   create database databasename;

Procedure to access The data of Models:

1. first create a database
   create database databasename;
2. For accessing the created database
   \c databasename;
3. after accessing the data base, we cannot access the table details directly
   by using the Model_names
4. So in-order to access the tables we need to to go with relations
5. syntax for accessing the list relations of the database
   \dt;
6. Example:

```
        List of relations
Schema | Name                | Type | Owner
public | app_access_records  | table | postgres
public | app_topic | table | postgres
public | app_webpage | table | postgres
public | auth_group | table | postgres
public | auth_group_permissions | table | postgres
public | auth_permission | table | postgres
public | auth_user | table | postgres
public | auth_user_groups | table | postgres
public | auth_user_user_permissions | table | postgres
public | django_admin_log | table | postgres
public | django_content_type | table | postgres
public | django_migrations | table | postgres
public | django_session | table | postgres
(13 rows)
```

7. we have to access the Table information by using the Relation names
8. Syntax for accessing the info of models
    select * from relation_name;

    example:
        select * from app_topic;
    procedure to create user by using pg admin:
    https://www.guru99.com/postgresql-create-alter-add-user.html#1

    Installation steps
    1. link for downloading postgradesql
     https://www.enterprisedb.com/downloads/postgres-postgresql-downloads
    2. click on downloaded software
    3. check the below mentioned softwares
        1. PostgreSQL server
        2. pgadmin 4
        3. Commandline tools and click on next
        4. Now it will ask for password
            enter password
            re-enter-password and press next
        5. leave the default port number as it is
        6. With the above steps PostgreSQL is installed successfully

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

View Class:

---

first we have import View class by below syntax
    from django.views.generic import View

Responding some data with View class:

---

views.py file:

```
from django.shortcuts import render
from django.http import HttpResponse
# Create your views here.
#returning string by using FBV

from django.views.generic import View,TemplateView
from app.forms import *

def FBV(request):
    return HttpResponse('<h1>This is Function Based View</h1>')


#returning string by using CBV

class CBV(View):
    def get(self,request):
        return HttpResponse('<h1>This is CBV</h1>')


#returning A HTML page by using FBV

def FBV_template(request):
    return render(request,'FBV_template.html')

#returning A HTML page by using CBV

class CBV_template(View):
    def get(self,request):
        return render(request,'CBV_template.html')



# validating Form by using FBV

def FBV_form(request):
    form=Student()
    if request.method=="POST":
        form_data=Student(request.POST)
        if form_data.is_valid():
            return HttpResponse(str(form_data.cleaned_data))
    return render(request,'FBV_form.html',context={'form':form})


# Validating Form By using CBV

class CBV_form(View):
    def get(self,request):
        form=Student()
        return render(request,'CBV_form.html',context={'form':form})

    def post(self,request):
        form_data=Student(request.POST)
        if form_data.is_valid():
            return HttpResponse(str(form_data.cleaned_data))

# Dont use View Class whenever u r dealing with Templates and Forms

class CBV_TemplateView(TemplateView):
    template_name='CBV_template.html'
```

Note:

It is advised not to use View Class when we r dealing with Html files and
Forms instead use TemplateView and FormView to reduce Code complexity

TemplateView:
---

1. The Django generic TemplateView view class enables developers to quickly
   create views that display simple templates without reinventing the wheel

2. Instead of extending View, override the get method and then process the
   template and return an HttpResponse object using a render function.
3. Simpy extend TemplateView when we r 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
3. in this type we have to import TemplateView into urls

 syntax of url in Urls.py file:

```
from django.views.generic import TemplateView

urlpatterns=[
    path('suffix/',TemplateView.as_view(template_name='name_of_the
                                                    _template'),name='name'),
    ]
```

Django Template Context with TemplateView:
---
1. If u r using TemplateView, you need to use the get_context_data method to
   provide any context data variables to your template.
2. 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.

FormView:
----
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.StudentForm        # 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. DetailView

ListView:
----

1. ListView should be used when you want to present a list of objects in a
   html page.
2. ListView can achieve everything which TemplateView can do with less code.

first we have to import View class by below syntax:
----
    from django.views.generic import ListView

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:

```
class CBV_listview(ListView):
   model=School
   #queryset=School.objects.filter(name='QSPIDERS')
   template_name='list.html'
   context_object_name='
   ordering = ['name']
```

School_list.html file content:

```
{% extends "myapp/base.html" %}

{% block body_block %}
<div class="jumbotron">
   <h1>Schools Are : </h1>
   {% if schools %}
   <ol>
      {% for school in schools%}
      <li><a href="{{school.id}}">{{school}}</a></h1></li>
      {% endfor %}
   </ol>

   {% else %}
     <h1>No School Found</h1>
   {% endif %}

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

2. DetailView:

1. DetailView should be used when you want to represent detail of a single model instance.

important points to remember:

1. First of all, when it comes to web development you really want to avoid hard coding paths in your templates.
2. The reason for this is that paths will be different for each and every instances of the model
3. so it becomes very difficult to find out and change each and every url path manually
4. So we have to define function which is responsible for returning the URL paths based on selected model instance
5. 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.
   1. 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 content:

```
from django.db import models
from django.urls import reverse

# Create your models here.

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_name="students")

    def __str__(self):
        return self.name
```

School_detail.html content:

```
{% 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:

```
class SchoolCreateView(CreateView):
   model=School
   #fields='__all__'    (fields functionality is same as in Meta class)
   fields=('name','principal','location')
```

school_form.html content:

```
{% 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('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"),
```

Django Model Fields

The fields defined inside the Model class are the columns name of the mapped table. The fields name should not be python reserve words like clean, save or delete etc.

Django provides various built-in fields types.

Field Name			Class	Particular
AutoField	AutoField(**options)	It An IntegerField that automatically increments.		
BigAutoField	BigAutoField(**options)	It is a 64-bit integer, much like an AutoField except that it is guaranteed to fit numbers from 1 to 9223372036854775807.		
BigIntegerField	BigIntegerField(**options)	It is a 64-bit integer, much like an IntegerField except that it is guaranteed to fit numbers from -9223372036854775808 to 9223372036854775807.		
BinaryField	BinaryField(**options)	A field to store raw binary data.		
BooleanField	BooleanField(**options)	A true/false field. The default form widget for this field is a CheckboxInput.		
CharField	CharField(max_length,**options)	is to represent the varchar max_lenght is mandatory along with that we can give any arguments that are there in the options		
DateField	DateField(auto_now=False, auto_now_add=False, **options)	It is a date, represented in Python by a datetime.date instance.		
DateTimeField	DateTimeField(auto_now=False, auto_now_add=False, **options)	It is used for date and time, represented in Python by a datetime.datetime instance.		
DecimalField	DecimalField(max_digits=None, decimal_places=None, **options)	It is a fixed-precision decimal number, represented in Python by a Decimal instance.		
DurationField	DurationField(**options)	A field for storing periods of time.		
EmailField	EmailField(max_length=254, **options)	It is a CharField that checks that the value is a valid email address.		
FileField	FileField(upload_to=None, max_length=100, **options)	It is a file-upload field.		
FloatField	FloatField(**options)	It is a floating-point number represented in Python by a float instance.		
ImageField	ImageField(upload_to=None, height_field=None, width_field=None, max_length=100, **options)	It inherits all attributes and methods from FileField, but also validates that the uploaded object is a valid image.		
IntegerField	IntegerField(**options)	It is an integer field. Values from -2147483648 to 2147483647 are safe in all databases supported by Django.		
NullBooleanField	NullBooleanField(**options)	Like a BooleanField, but allows NULL as one of the options.		
PositiveIntegerField	PositiveIntegerField(**options)	Like an IntegerField, but must be either positive or zero (0). Values from 0 to 2147483647 are safe in all databases supported by Django.		
SmallIntegerField	SmallIntegerField(**options)	It is like an IntegerField, but only allows values under a certain (database-dependent) point.		
TextField	TextField(**options)	A large text field. The default form widget for this field is a Textarea.		
TimeField	TimeField(auto_now=False, auto_now_add=False, **options)	A time, represented in Python by a datetime.time instance.		

Django Model Fields Example

```
1. first_name = models.CharField(max_length=50) # for creating varchar column
2. release_date = models.DateField()            # for creating date column
3. num_stars = models.IntegerField()            # for creating integer column
```

Field Options

Each field requires some arguments that are used to set column attributes. For example, CharField requires mac_length to specify varchar database.

Common arguments available to all field types. All are optional.

Field Options	Particulars
Null	Django will store empty values as NULL in the database.
Blank	It is used to allowed field to be blank.
Choices	An iterable (e.g., a list or tuple) of 2-tuples to use as choices for this field.

Default	The default value for the field. This can be a value or a callable object.
help_text	Extra "help" text to be displayed with the form widget. It's useful for documentation even if your field isn't used on a form.
primary_key	This field is the primary key for the model.
Unique	This field must be unique throughout the table.

Steps:

1. Create project
2. Create app
3. Create templates folder
4. Create static folder in inner project
   - create css,js folders inside static
5. Create static folder in app
   - Create images folder inside static
6. Create variable for path of project static folder
7. Create variable for path of app static folder
8. Register the paths of STATIC_PROJECT_DIR & STATIC_DIR_APP in list variable
9. Create a variable for STATIC_ROOT
10. Run python manage.py collectstatic for collecting all static files