Django Index

→To create a project in Django cmd:"Django-admin startproject "projectname"

→To create a application in Django cmd:"py manage.py startapp "appname"

→To runserver cmd: "py manage.py runserver"

→ To create Template folder inside projectlevel.

TEMPLATE_DIR=BASE_DIR'templates'

→ To create Static folder inside projectlevel.

STACTIC_DIR=BASE_DIR'static'

STATICFILES_DIRS=[BASE_DIR]

→To test a project cmd:"py test.py"

CHAPTER-1

Introduction to Web development and Django

Web Application:

-->The application which will provide services over the web are called as web applications.

Ex:

gmail,facebook...........

-->Every web application contains 2-components.

1).Front End

2).Back End

1).Front End:

-->It represents what end user is seeing on the website.

-->We can develop front-end component by using HTML,CSS,JS,jquery & bootstrap.

-->Jquery and bootstrap are advanced front-end technologies, which are developed using HTML,CSS and JS only.

HTML:

-->Hyper Text Markup Language

-->Every web application should contain html. i.e HTML is the mandatory technology for web development. It represent structure of web page.

CSS:Cascade Style Sheets

-->It is an optional technology, still web application contains CSS.

-->The main objective of CSS is to add styles to the HTML pages like colors,fonts,borders etc....

Java Script:

-->It allows to add interactivity to the web application including programming logic.

-->The main objective of java script is to add functionality to the HTML pages. i.e to add dynamic nature to the HTML pages.

HTML-->Meant for static response

HTML + JS-->Meant for dynamic response

Ex-1:

To display "Welcome To Django Classes" response to the end user only HTML is enough, b'z it is a static response.

Ex-2:

To display current server date and time to the end user, only HTML is not enough we required to use some extra technology like JS, JSP, ASP, PHP etc as it is dynamic response.

Static Response Vs Dynamic Response:

-->If the response is not varied from time to time and person to person then it is considered as static response.

Ex:

Login page Gmail
Login page of HMS

-->If the response is varied from time to time and person to person then it is considered as dynamic response.

Ex:

Inbox page of gmail
Balance page of HDFC Bank

2).Back End:

-->It is the technology used to decide what to show to the end user on the Front-End.

-->i.e back end is responsible to generate required response to the end user, which is displayed by the Front-End.

-->Back-End have 3-components.

1).The language like java,python etc....
2).The framework like Django, Flask, Pyramid etc....
3).The Database like SQLite,Oracle,MySQL etc.....

-->For the Backend language python is the best choice b'z of the reasons are:

Simple and easy to learn, libraries, concise code.

-->For the framework Django is the best choice b'z it is fast, secure and scalable. Django is the most popular web application framework for python.

-->Django provides inbuilt database which is nothing but SQLite, which is the best choice for database.

Django:

-->Django is a web application framework.

-->Django is used to develop web applications.

-->Django is completely written in python programming language.

-->It is a high level, opensource python based web framework.
-->DRY(Don't Repeat Yourself).
-->It provides RAD(Rapid Application Development) facilities.
-->It is based on MVT(Model View Template) design pattern.
Diagram
-->Django was created in 2003 as an project at Lowrence journal-world news paper for their web development.
-->The original authors of Django framework are:Adrian Holovaty, Simon Willison.
-->After testing this framework with heavy traffics, developers released for the public as opensource framework on july 21st 2005.
-->The django was named in the memory of Guitarist Django Reinhardt.
-->Official website:https://www.djangoproject.com/

Top 5 Features of Django Framework:

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

1).Fast:

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

2).Fully Loaded:

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

3).Security:

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

4).Scalability:

Some of the busiest sites on the planet use Djangos ability to quickly and flexibly scale to meet the heaviest traffic demands.

5).Versatile:

Companies, organizations and governments have used Django to build all sorts of things — from content management systems to social networks to scientific computing platforms.

CHAPTER-2

Django & Atom Installation and Development of First Web Application

<u>How to install Django:</u>
Python should be available in machine.

```
C:\Windows\System32>py --version
Python 3.10.5
```

-->Install Django by using pip

```
C:\Windows\System32>pip install django==4.1
```

To check the django version:

```
C:\Windows\System32>python -m django --version
4.0
```

IDE:
    ATOM/Pycharm/Vscode

<u>Django Project vs Django Application:</u>
-->A django project is a collection of applications and configurations which forms a full web application.
Ex:
    Banking Peoject

-->A django application is responsible to perform a particular task in our entire web application.
Ex:
    loan app
    registration app
    polling app

Project: Several Applications + Configuration Information

<u>Note:</u>
    1.The django applications can be plugged into other projects. i.e these are re-usable.(Pluggable Django Applications)
    2.Without existing Django project there is no chance of existing Django application. Before creating any application first we have to create project.

<u>How to create Django project?</u>

```
D:\>mkdir Django_19MAR_7PM
D:\>cd Django_19MAR_7PM
D:\Django_19MAR_7PM>django-admin startproject firstproject
                                   or
D:\Django_19MAR_7PM>py -m django startproject firstproject

D:\Django_19MAR_7PM>tree /f
D:.
└───firstproject
    │   manage.py
    │
    └───firstproject
            asgi.py
            settings.py
            urls.py
            wsgi.py
            __init__.py
```

1).__init__.py:

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

2). settings.py:

In this file we have to specify all our project settings and configurations like installed apps, middleware configuration, database configuration etc......

3).urls.py:

Here we have to store all our url-patterns of our project.

For every view(web page), we have to define separate url-pattern. End user can use url-patterns to access our web pages.

4).wsgi.py:

wsgi-->web server gateway interface

We can use this file while developing our application in production on online server.

5).asgi.py:

asgi--> Asynchronous server gateway interface

<u>6).manage.py:</u>

The most commonly used python script is manage.py

It is a commandline utility to interact with Django project in various ways like to run development server, run tests, create migrations etc.......

<u>How to run Django development server:</u>

-->We have to move to manage.py file location and we have to execute.

```
D:\Django_19MAR_7PM\firstproject>py manage.py runserver
```

Now server will be started.

How to send the first request:

-->Open browser and send request:

http://127.0.0.1:8000/

Ex:

```
a = [0,1,2,3]
for a[-1] in a:
	print(a[-1])
```

Mar 20

<u>Role of web server:</u>

-->Web server will provide environment to run our web application.

-->Web server is responsible to recieve the request and formard request to the corresponding web component based on url-pattern and to provide response to the end user.

-->Django framework is responsible to provide development server. Even Django framework provides one inbuilt database sqlite-3

<u>Note:</u>

Once we started server a special database related files will be generated in our project folder structure.

<u>Creation of first web application:</u>

-->Once we create Django project, we can create any number of applications in that project.

To create an application:

```
D:\Django_20MAR_7PM\firstproject>py manage.py startapp firstapp
```

The following folder structure got created under firstapp:

```
D:.
|   admin.py
|   apps.py
|   models.py
|   tests.py
|   views.py
|   __init__.py
|
└───migrations
      __init__.py
```

1.__init__.py:

It is a blank python script.

2.admin.py:

We can register our models in this file. Django will use these models with Django's admin interface.

3.apps.py:

In this file we have to specify application's specific configurations

4.models.py:

In this file we have to store application's data models.

5.tests.py:

In this file we have to specify test functions to test our code.

6.views.py:

In this file we have to save functions that handles requests and return required response.

7).Migrations folder:

This directory stores database specific information related to models.

Note:

The most commonly used files in every project are views.py and models.py

Activities required for application:

Activity-1: Add our application in settings.py, so that Django aware about our application.

- **settings.py**

```
INSTALLED_APPS = [
    'firstapp',
]
```

Activity-2:

Create a view function for our application in views.py

View is resposible to prepare required response to the end user. i.e view contains business logic.

There are 2-types of views

1.Function based views

2.Class based views

- **views.py**

```
from django.http import HttpResponse
def wish(request):
   s = '<h1>Hello students welcome to Mahesh Sir django classes</h1>'
   return HttpResponse(s)
```

Note:

1.Each view will be specified as one function in views.py

2.In the above example wish is the name of the function which is nothing but one view.

3.Each view should take atleast one arg(request)

4.Each views should return HttpResponse object with required response.

View can accept request as input and perform required operations and provide proper response to the end user.

Activity-3:

Define url-pattern for our view in urls.py

This url-pattern will be used by end user to send request for our view.

The 'urlpatterns' list routes URL's to views.

- **urls.py**

```
from firstapp import views
urlpatterns = [
   path('admin/', admin.site.urls),
   path('greet/',views.wish)
]
```

Whenevr end user sending the request with url-pattern:greet then wish() function will be executed and provide required response

Activity-4:

start server send the request.
http://127.0.0.1:8000/greet/

Mar 21

Step-1:Create project
Step-2:Create application
Step-3:Add app in settings.py
Step-4:Function in views.py
Step-5:url for function
Step-6:Start server send request

Http Request flow in Django Application:

-->Whenever end user sending the request (http://127.0.0.1:8000/greet/) first django development server will get that request.
-->From the request Django will identify url-pattern and by using urls.py, the corresponding view will be identified.
-->The request will be forwarded to the view. The corresponding function will be executed and provide required response to the end user.

Q.write django application to send request to display server time as response

Step-1: Create project

```
D:\Django_20MAR_7PM>django-admin startproject secondproject
```

Step-2: Create application

```
D:\Django_20MAR_7PM\secondproject>py manage.py startapp firstapp
```

Step-3: Add app in settings.py

```
INSTALLED_APPS = [
      'firstapp'
]
```

Step-4: Function in views.py

```
import  datetime
from django.http import  HttpResponse
def timeinfo(request):
   date = datetime.datetime.now()
   msg = '<h1>Hello Frien Good Evening!!!!!!</h1><hr>'
   msg += '<h2>Now the server time is:'+str(date)+'</h2>'
   return HttpResponse(msg)
```

Step-5:url for function

```
        path('time/', views.time_info),
```

Step-6: Start server send request

```
        http://127.0.0.1:8000/time/
```

Q.Single application with multiple views

```
D:\Django_20MAR_7PM>django-admin startproject sunnyjobsproject
D:\Django_20MAR_7PM>cd sunnyjobsproject
D:\Django_20MAR_7PM\sunnyjobsproject>py manage.py startapp jobsapp
```

-->Add app in settings.py

- **views.py**

```
from django.http import HttpResponse
def hyd_jobs_view(request):
   s = '<h1>Hyderabad Jobs Information</h1>'
   return HttpResponse(s)
def bng_jobs_view(request):
   s = '<h1>Bangalore Jobs Information</h1>'
   return HttpResponse(s)
def pune_jobs_view(request):
   s = '<h1>Pune Jobs Information</h1>'
   return HttpResponse(s)
def bihar_jobs_view(request):
   s = '<h1>Bihar Jobs Information</h1>'
   return HttpResponse(s)
```

- **urls.py**

```
path('hyd/', views.hyd_jobs_view),
path('pune/', views.pune_jobs_view),
path('bng/', views.bng_jobs_view),
path('bihar/', views.bihar_jobs_view),
```

Start server send request:

```
        http://127.0.0.1:8000/hyd/
        http://127.0.0.1:8000/pune/
        http://127.0.0.1:8000/bng/
        http://127.0.0.1:8000/bihar/
```

Q.Based on time good mng, good aft, good evening & good night

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

Ex:

```
n = 6
while n>0:
        print(n)
        n -= 2 if n%3==0 else 1
```

Mar 22

Q.Single project with multiple applications?

```
D:\Django_20MAR_7PM>django-admin startproject multiappProject
D:\Django_20MAR_7PM>cd multiappProject
D:\Django_20MAR_7PM\multiappProject>py manage.py startapp firstapp
D:\Django_20MAR_7PM\multiappProject>py manage.py startapp secondappp
```

-->Add apps in settings.py

- **FirstApp views.py**

```
from django.http import HttpResponse
def wish1(request):
   return HttpResponse('<h1>Hello This Is From First Application</h1>')
```

- **SecondApp:views.py**

```
from django.http import HttpResponse
def wish2(request):
   return HttpResponse('<h1>Hello This Is From Second Application</h1>')
```

- **urls.py**

1st way:

```
from firstapp import views as v1
from secondapp import views as v2
urlpatterns = [
   path('wish1/', v1.wish1),
   path('wish2/', v2.wish2),
]
```

2nd way:

```
from firstapp.views import wish1
from secondapp.views import wish2
urlpatterns = [
   path('wish1/', wish1),
   path('wish2/', wish2),
]
```

Defining URL patterns at Application level instead of Project level:

-->A Django project can contains multiple applications and each application contains multiple views. Defining url-pattern for all views of all applications inside urls.py file of project creates maintenance problem and reduces re-usability of applications.

-->We can solve this problem by defining url-pattern at application level instead of project level. For every application we have to create a separate urls.py file and we have to define all that application specific urls in that file.

-->We have to link this application level urls.py file to project level urls.py file by using include() function.

<u>Ex:</u>

```
D:\Django_20MAR_7PM>django-admin startproject baseproject
D:\Django_20MAR_7PM>cd baseproject
D:\Django_20MAR_7PM\baseproject>py manage.py startapp testapp
```

-->Add app in settings.py

- **views.py**

```
from django.http import HttpResponse
def first_view(request):
   return HttpResponse('<h1>First View Response</h1>')
def second_view(request):
   return HttpResponse('<h1>Second View Response</h1>')
def third_view(request):
   return HttpResponse('<h1>Third View Response</h1>')
def fourth_view(request):
   return HttpResponse('<h1>Fourth View Response</h1>')
def fifth_view(request):
   return HttpResponse('<h1>Fifth View Response</h1>')
```

-->Create a separate file urls.py file inside application

- **urls.py**

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

Include this application level urls.py file inside project level urls.py file

- **project level urls.py**

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

-->Start server send request:

```
            http://127.0.0.1:8000/testapp/first/
```

Note:

We can see re-usability of application in other projects just with only 2-lines addition.

1).settings.py--->Add application

2).urls.py-->just add:path('testapp/', include('testapp.urls')),

Advantages:

The main advantages of defining url-pattern at application level instead of project level are:

1.It promotes re-usability of django application across multiple projects.

2.Project level urls.py will be clean and more readable.

Mar 25

Ex:

```
D:\Django_20MAR_7PM>django-admin startproject applevelurlsproject
D:\Django_20MAR_7PM>cd applevelurlsproject
D:\Django_20MAR_7PM\applevelurlsproject>py manage.py startapp testapp
```

-->Add app in settings.py

- **views.py**

```
from django.http import HttpResponse
def exams_view(request):
   return HttpResponse('<h1>Exams View</h1>')
def attendance_view(request):
   return HttpResponse('<h1>Attendance View</h1>')
def fees_view(request):
   return HttpResponse('<h1>Fees View</h1>')
```

- **urls.py-->Application level**

```
from django.urls import path
from . import views
urlpatterns = [
   path('exams/', views.exams_view),
   path('attendance/', views.attendance_view),
   path('fees/', views.fees_view),
]
```

Create another project:
D:\Django_20MAR_7PM>django-admin startproject sunnyproject

-->Copy testapp from applevelproject and paste it in current project, then include app level urls in project level urls.

- **project level urls**

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

start server send request:

http://127.0.0.1:8000/testapp/exams/
http://127.0.0.1:8000/testapp/fees/
http://127.0.0.1:8000/testapp/attendance/

CHAPTER-3

Django Templates & Static Files

-->It is not recommended to write HTML code inside python script(views.py file) because:

1.It reduces readability because of python code mixed with HTML code.

2.No separation of roles. Python developers has to concentrate on both python code and HTML code.

3.It does not promotes re-usability of code.

-->We can overcome these problems by separating HTML code into a separate html file. This html file is nothing but template.
-->From the python file(views.py) we can use these templates based on our requirement.
-->We have to write templates at project level only, we can use these templates in multiple applications.

<u>Python stuff:</u>
pathlib-->module name
Path-->class name

pathlib module provides various classes representing file system paths based on different OS.

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

<u>Note:</u>
The main advantage of this aapproach is we are not required to hard code system specific paths(locations) in python script.

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

MVT design pattern:
M-->Model(Database)
V-->View(Business logic-->Python file)
T-->Template(Presentation Layer)

Steps to develop Template Based Application:

```
D:\Django_20MAR_7PM>django-admin startproject templateproject
D:\Django_20MAR_7PM>cd templateproject
D:\Django_20MAR_7PM\templateproject>py manage.py startapp testapp
```

-->Add app in settings.py

-->Create a 'templates' folder inside main project folder.
-->In that templates folder create a separate folder named with testapp to hold that particular application specific templates.
-->Add templates folder to settings.py file so that django can aware of our templtes.

```
TEMPLATES = [
                                        'DIRS':
[D:\Django_20MAR_7PM\templateproject\templates],
]
```

-->It is not recommended to hard code system specific location in settings.py file.
To overcome this problem, we can generate templates directory path programmatically as.

```
                  BASE_DIR = Path(__file__).resolve().parent.parent
                  TEMPLATE_DIR = BASE_DIR/'templates'
```

-->Specify TEMPLATE_DIR inside settings.py

```
                  'DIRS': [TEMPLATE_DIR],
```

-->Create html file inside templateproject/templates/testapp folder. This html file is nothing but Template.

- **wish.html**

```
<body>
   <h1>Welcome to Django Template Demo</h1>
   <h2>Second hero of Django in MVT:Templates</h2>
</body>
```

-->Define function based view inside views.py

- **views.py**

```
def wish(request):
   return render(request,'testapp/wish.html')
```

Define url-pattern:

- **urls.py**

```
path('test/',views.wish)
```

-->start server send request:http://127.0.0.1:8000/test/

Mar 26

Django Templates:

M	-->Model
V	-->View
T	-->Template

Template Tags:

-->From python views.py we can inject dynamic content to the template file by using template tags.
-->Template tags also known as Template variables.
-->Take special care about template tag syntax it is not python syntax and not HTML syntax it is a Django syntax.

Template tag syntax for inserting text data:{{insert_data}}

-->This template tag we have to place inside template file(i.e html file) and we have to provide insert_data value form the python views.py file.

Send date and time from views.py to template file

- **wish.html**

```
<body>
   <h1>Welcome to Django Template Demo</h1>
   <h2>Now server time is:{{insert_date}}</h2>
</body>
```

- **views.py**

```
import datetime
def wish(request):
   date = datetime.datetime.now()
   my_dict = {"insert_date":date}
   return render(request,'testapp/wish.html',context=my_dict)
```

Note:

The values to the template variables should be passed from the views in the form of dictionary as argument to context. context is an optional

```
return render(request,'testapp/wish.html',my_dict)
```

-->We can pass directly dict to render function:

```
return render(request,'testapp/wish.html',{"insert_date":date})
```

To display student info:

- **views.py**

```
name = 'Sunny'
rollno = 101
marks = 98
my_dict = {"insert_date":date,'name':name,'rollno':rollno,'marks':marks}
```

- **wish.html**

```
        <ol>
   <li>Name:{{name}}</li>
   <li>Roll No:{{rollno}}</li>
   <li>Marks:{{marks}}</li>
  </ol>
```

Application:

End User:

Hello Friend, The current server time is:date and time
Course Information:
  Name:Django
  Prerequisite:Python
  Current Batch Students are:

```
D:\Django_20MAR_7PM>django-admin startproject templateproject2
D:\Django_20MAR_7PM>cd templateproject2
D:\Django_20MAR_7PM\templateproject2>py manage.py startapp testapp
```

-->Add app in settings.py

- **views.py**

```
import datetime
def info_view(request):
   time = datetime.datetime.now()
   name = 'Django'
   prerequisite = 'Python'
   my_dict = {'time':time,'name':name,'prerequisite':prerequisite}
   return render(request,'testapp/results.html',my_dict)
```

-->Create templates folder in main project folder. Add TEMPLATE_DIR in settings.py.

- **results.html**

```
<body>
   <h1>Hello Friend, The server time is:{{time}}</h1>
   <h2>Course Name:{{name}}</h2>
   <h2>prerequisite:{{prerequisite}}</h2>
   <ul>
     <li>Good to get job very easily</li>
     <li>Learning is also very easy</li>
     <li>You can claim 3 to 4 years of exp</li>
     <li>It is very helpful for freshers</li>
   </ul>
   <table border="3">
     <thead>
       <th>S.No</th>
       <th>Name</th>
       <th>Feedback</th>
     </thead>
     <tr>
       <td>101</td>
       <td>Sunny</td>
       <td>Good</td>
     </tr>
     </table>
  </body>
```

<u>Astrology Project:</u>

```
D:\Django_20MAR_7PM>django-admin startproject astrologyproject
D:\Django_20MAR_7PM>cd astrologyproject
D:\Django_20MAR_7PM\astrologyproject>py manage.py startapp testapp
```

-->Add app in settings.py

- **views.py**

```
import datetime,random
def result_view(request):
   msg_list = [
   'The golden days a head',
   'Better to sleep more time even in class also',
   'Tomorrow will be the best day of your life',
   'Tomorrow is the perfect day to propose your GF',
   'Very soon you will get the job'
   ]
   names_list = ['sunny', 'kareena','samantha','samyuktha','radhika']
   time = datetime.datetime.now()
   h = int(time.strftime('%H'))
   if h < 12:
      s = 'Good Morning'
   elif h < 16:
      s = 'Good Afternoon'
   elif h < 21:
      s = 'Good Evening'
   else:
      s = 'Good Night'
   name = random.choice(names_list)
   msg = random.choice(msg_list)
   my_dict = {'time':time,'name':name,'msg':msg,'wish':s}
   return render(request,'testapp/astro.html',my_dict)
```

- **urls.py**

```
path('astr/', views.result_view),
```

- **<u>astro.html</u>**

```
<body>
   <h1>Hello Friend, <span>{{wish}}</span></h1>
   <h2>Now The Server Time Is:<span>{{time}}</span></h2>
   <h1>Astrology Info For Prasad:<span>{{msg}}</span></h1>
   <h2>Very soon, you are going to get marriage
with:<span>{{name}}</span></h2>
</body>
```

Mar 27

Working with static files

-->Up to this just we injected normal text data into templates by using template tags.

-->But sometimes our requirement is to insert static files like images, css files etc inside template file

Process to include stattic files inside template:

1).Create folder named with 'static' inside main project folder. It is exactly same as creating 'templates' folder.

2).In that 'static' folder create 'images' folder to place image files.

3).Add static directory path to settings.py, so that django can aware of our images.

Ex:

```
D:\Django_20MAR_7PM>django-admin startproject staticfilesproject
D:\Django_20MAR_7PM>cd staticfilesproject
D:\Django_20MAR_7PM\staticfilesproject>py manage.py startapp testapp
```

-->Add app in settings.py

- **settings.py**

```
STATIC_DIR = BASE_DIR/'static'

STATIC_URL = 'static/'
STATICFILES_DIRS = [STATIC_DIR]
```

Make sure all the paths are correct or not

    http://127.0.0.1:8000/static/images/sunny1.jpg

Use template tag to insert an image.

    At the beginning of HTML just after <!DOCTYPE html>, we have to include the following template tag

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

-->Just we have to conveying to the Django to load all static files.

-->We have to include image file as:

```
<img src="{% static 'images/sunny1.jpg' %}" alt="">
```

- **views.py**

```
def result_view(request):
   subjects = {'s1':'Python','s2':'Django','s3':'RestAPI','s4':'MongoDB'}
   return render(request,'testapp/results.html',subjects)
```

- **results.html**

```
<body>
   <h1>Hello frinds, My brnad ambassodor for my classes</h1>
   <img src="{% static 'images/sunny1.jpg' %}" alt="">
   <h2>Courses by Mahesh Sir:</h2>
   <ul>
     <li>{{s1}}</li>
     <li>{{s2}}</li>
     <li>{{s3}}</li>
     <li>{{s4}}</li>
   </ul>
  </body>
```

- **urls.py**

```
path('results/', views.result_view),
```

<u>How to include css files:</u>

```
D:\Django_20MAR_7PM>django-admin startproject sunnynewproject
D:\Django_20MAR_7PM>cd sunnynewproject
D:\Django_20MAR_7PM\sunnynewproject>py manage.py startapp testapp
```

-->Add app in settings.py
-->Create a folder templates
-->Update this one in settings.py

```
TEMPLATE_DIR = BASE_DIR/'templates'
```

-->Create static folder:
Inside static folder create 'css' and 'images' fodlers

```
STATIC_DIR = BASE_DIR/'static'
STATICFILES_DIRS = [STATIC_DIR]
```

- **views.py**

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

- **index.html**

```
<!DOCTYPE html>
{% load static %}
   <link rel="stylesheet" href="{% static 'css/demo.css' %}">
  <body>
   <h1>Welcome To SUNNY NEWS Portal</h1>
   <img src="{% static 'images/news.jpg' %}" alt="">
   <ul>
     <li><a href="#">Movies Information</a></li>
     <li><a href="#">Sports Information</a></li>
     <li><a href="#">Politics Information</a></li>
   </ul>
```

- **demo.css**

```
body{
  background: yellow;
  color: red;
  text-align: center;
}
ul{
  font-size: 20px;
  text-align: left;
}
img{
  height: 250px;
  width: 300px;
  margin: 25px;
  border: 5px solid red;
}
```

- **views.py**

```
def movies_view(request):
   head_msg = 'Movies Information'
   sub_msg1 = 'OG is the upcoming movie'
   sub_msg2 = 'Devara will release on next month'
   sub_msg3 = 'Planning for aashiqui-3 with Mahesh sir and Sunny Leone'
   my_dict = {'head_msg':head_msg,'sub_msg1':sub_msg1,
'sub_msg2':sub_msg2,'sub_msg3':sub_msg3}
   return render(request,'testapp/news.html',my_dict)
```

- **news.html**

```
<body>
   <h1>{{head_msg}}</h1>
   <ul>
     <li>{{sub_msg1}}</li>
     <li>{{sub_msg2}}</li>
     <li>{{sub_msg3}}</li>
   </ul>
   <img src="{% static 'images/1.jpg' %}" alt="">
   <img src="{% static 'images/2.jpg' %}" alt="">
   <img src="{% static 'images/3.jpg' %}" alt="">
 </body>
```

- **urls.py**

```
path('', views.news_info),
path('movies/', views.movies_view),
```

- **index.html**

```
<li><a href="/movies">Movies Information</a></li>
```

Mar 28

- **views.py**

```
def sports_view(request):
   head_msg = 'Sports Information'
   sub_msg1 = 'Yesterday IPL match won by SRH'
   sub_msg2 = 'Today match b/w RR & DC'
   sub_msg3 = 'Who will win IPL cup???????????'
   type = 'sports'
   my_dict = {'head_msg':head_msg,'sub_msg1':sub_msg1,
'sub_msg2':sub_msg2,'sub_msg3':sub_msg3,'type':type}
   return render(request,'testapp/news.html',my_dict)

def politics_view(request):
   head_msg = 'Politics Information'
   sub_msg1 = 'Telangana CM was revanth reddy'
   sub_msg2 = 'India PM was Modi'
   sub_msg3 = 'Who is upcoming CM for AP????????????'
   type = 'politics'
   my_dict = {'head_msg':head_msg,'sub_msg1':sub_msg1,
'sub_msg2':sub_msg2,'sub_msg3':sub_msg3,'type':type}
   return render(request,'testapp/news.html',my_dict)
```

- **news.html**

```
<body>
   <h1>{{head_msg}}</h1>
   <ul>
     <li>{{sub_msg1}}</li>
     <li>{{sub_msg2}}</li>
     <li>{{sub_msg3}}</li>
   </ul>
   {% if type == 'movies' %}
   <img src="{% static 'images/1.jpg' %}" alt="">
   <img src="{% static 'images/2.jpg' %}" alt="">
   <img src="{% static 'images/3.jpg' %}" alt="">
   {% elif type == 'sports' %}
   <img src="{% static 'images/4.jpg' %}" alt="">
   <img src="{% static 'images/5.jpg' %}" alt="">
   <img src="{% static 'images/6.jpg' %}" alt="">
   {% elif type == 'politics' %}
   <img src="{% static 'images/7.jpg' %}" alt="">
   <img src="{% static 'images/8.jpg' %}" alt="">
   <img src="{% static 'images/9.jpg' %}" alt="">
   {% endif %}
 </body>
```

CHAPTER-4

Working with Models and Databases:

-->As part of web application development, compulsory we required to interact with database to store our data and to retrieve our stored data.
-->Django provied a in-built support for database operations. Django provides in-built database sqlite3.
-->For small to medium applications this database is more enough. Django can provide support for other DB also like Oracle,Mysql,MongoDb.........

Database Configurations:
-->If we want to use default DB(sqlite3) then we are not required to do any configuration.
-->The default sqlite3 configurations in settings.py file are declared as:
```
DATABASES = {
   'default': {
      'ENGINE': 'django.db.backends.sqlite3',
      'NAME': BASE_DIR / 'db.sqlite3',
   }
}
```

Database connection with Mysql:
```
DATABASES = {
   'default': {
      'ENGINE': 'django.db.backends.mysql',
      'NAME': 'djangodb',
                     'USER': 'root',
                     'PASSWORD': 'root',
                     'HOST': 'localhost',
                     'PORT': 3306,
   }
}
```

```
SQL> select * from global_name;
```

Database connection with Oracle:
```
DATABASES = {
   'default': {
        'ENGINE': 'django.db.backends.oracle',
        'NAME': 'ORCL',
                      'USER': 'scott',
                      'PASSWORD': 'tiger',
                      'HOST': 'localhost',
                      'PORT': 1521,
    }
}
```

-->If we dont want to sqlite3 database then we have to configure our own database with the following parameters.

1).ENGINE:Name of the 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 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 command in shell.

```
        D:\Django_20MAR_7PM\sunnynewproject>py manage.py shell
>>>from django.db import connection
>>>c = connection.cursor()
```

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

Model Class:
-->A model is a python class which contains database information.
-->It contains fields and behaviours 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 database field(Column name in table).
-->We have to write all model classes inside 'models.py' file.

App:

```
D:\Django_20MAR_7PM>django-admin startproject modelproject
D:\Django_20MAR_7PM>cd modelproject
D:\Django_20MAR_7PM\modelproject>py manage.py startapp testapp
```

-->Add app in settings.py

- **models.py**

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

Note:

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

table_name:appname_modelname
:testapp_employee

Fields:eno,ename,esal and eaddr and one extra field:id
Behaviours:eno is integer, ename is char.....
Model class:database table name + filed name + field behaviours

Converting Model class into database specific SQL code:

Once we write Model class, we have to migrate the corresponding SQL code, for this we have to use makemiggrations.

```
D:\Django_20MAR_7PM\modelproject>py manage.py makemigrations
```

How to see the corresponding SQL code of migrations:

```
D:\Django_20MAR_7PM\modelproject>py manage.py sqlmigrate testapp 0001

CREATE TABLE "testapp_employee" ("id" integer NOT NULL PRIMARY KEY
AUTOINCREMENT, "eno" integer NOT NULL, "ename" varchar(30) NOT NULL,
"esal" real NOT NULL, "eadddr" varchar(30) NOT NULL);
COMMIT;
```

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.

```
D:\Django_20MAR_7PM\modelproject>py manage.py migrate
```

What is 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.

How to check created table in Django admin interface:

We have to register model class in 'admin.py' file

- **admin.py**

```
from django.contrib import admin
from  testapp.models import  Employee
admin.site.register(Employee)
```

Creation of superuser to login to admin interface

We can create super user by using the command.

```
D:\Django_20MAR_7PM\modelproject>py manage.py createsuperuser
```

We can login to admin interface.

start server and login to admin interface by using

http://127.0.0.1:8000/admin/

Q.Difference between makemigrations and migrate?

'makemigrations' is responsible to generate SQL code for python model class where as 'migrate' is responsible to execute the SQL code so that tables will be created in the database.

Read data from the database and display for the end user?

1).Start project
2).Start app
3).Add app in settings.py
4).Add database configurations in settings.py
5).Test database connection
6).Create Model class

7).Makemigrations and Migrate
8).Register model in admin.py
9).Create super user.
10).Login to admin interface and check table created or not.
11).Template file and static file and corresponding configuration in settings.py
12).view function to communicate with database and to get data and send this data to template file which is responsible to display to end user.

Apr 01

pp:

```
D:\Django_20MAR_7PM>django-admin startproject modelproject2
D:\Django_20MAR_7PM>cd modelproject2
D:\Django_20MAR_7PM\modelproject2>py manage.py startapp testapp
```

-->Add app in settings.py

- **models.py**

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

-->Makemigrations and migrate
-->Register model class and ModelAdmin class in admin.py

- **admin.py**

```
from testapp.models import Employee
class EmployeeAdmin(admin.ModelAdmin):
   list_display = ['eno','ename','esal','eaddr']
admin.site.register(Employee,EmployeeAdmin)
```

-->create super user:

```
            py manage.py createsuperuser
```

-->Login to admin interface and add some data manually.
Create a function in views.py

- **views.py**

```
from testapp.models import Employee
def empdata_view(request):
   emp_list = Employee.objects.all()
```

```
    my_dict = {'emp_list':emp_list}
    return render(request,'testapp/emp.html',my_dict)
```

- **emp.html**

```
<link rel="stylesheet" href="{% static 'css/emp1.css' %}">
<body>
  <h1>Employee List</h1>
  {% if emp_list %}
  <table border="3">
   <thead>
    <th>ENO</th>
    <th>ENAME</th>
    <th>ESAL</th>
    <th>EADDR</th>
   </thead>
   {% for emp in emp_list %}
   <tr>
    <td>{{emp.eno}}</td>
    <td>{{emp.ename}}</td>
    <td>{{emp.esal}}</td>
    <td>{{emp.eaddr}}</td>
   </tr>
   {% endfor %}
  </table>
  {% else %}
  <p>No Records Found!!!!</p>
  {% endif %}
 </body>
```

- **urls.py**

```
path('emp/', views.empdata_view),
```

- **emp1.css**

```
body{
 background: yellow;
 color: red;
}
h1{
 text-align: center;
}
```

```
table{
  margin: auto;
}
```

1).End user sending request:
```
        http://127.0.0.1:8000/emp/
```

2).View function will get request.

3).view asking model to connect with database and provide required data.
```
                Employee.objects.all()
```

4).Model will communicate with database and provide required data to the view function.
```
                emp_list
```

5).View will send that data to template file
```
                return
render(request,'testapp/emp.html',{'emp_list':emp_list})
```

Questions related to models and templates?
1).How to configure database inside settings.py?
2).How to check the connections?
3).How to define model class inside models.py?
4).How to perform makemigrations?
5).How to perform migrate?
6).What is the diff between makemigrations and migrate?
7).What is the advantage of creating table by using migrate command instead of creating manually in the database?
8).How to add modle to admin interface inside admin.py?
9).To display total data, how to write model admin class inside admin.py?
10).How to create super user?
11).How to see generated sql code as result of makemigrations?

<u>How to generate fake data for application:</u>
```
>>>pip install faker
from faker import Faker
fake = Faker()
name = fake.name()
print(name)
```

```
fname = fake.first_name()
print(fname)
lname = fake.last_name()
print(lname)
date1 = fake.date()
print(date1)
number = fake.random_number(5)
print(number)
email1 = fake.email()
print(email1)
city = fake.city()
print(city)
print(fake.random_int(min=0,max=9999))
print(fake.random_element(elements=('sunny','katrina','kareena','deepika')))
```

Phone Number Generation:

```
from random import *
def phonenumbergen():
        d1 = randint(6,9)
        num = '' + str(d1)
        for i in range(9):
                num += str(randint(0,9))
        return int(num)
for i in range(10):
        print(phonenumbergen())
```

Apr 02

Insert data into modek class by using faker module

```
D:\Django_20MAR_7PM>django-admin startproject modelproject3
D:\Django_20MAR_7PM>cd modelproject3
D:\Django_20MAR_7PM\modelproject3>py manage.py startapp testapp
```

-->Add app in settings.py

-->Database configuration(Use Mysql)
-->Create a database in Mysql(studentdb_7pm)
-->Check database connections.

- **models.py**

```
class Student(models.Model):
   rollno = models.IntegerField()
```

```
    name = models.CharField(max_length=30)
    dob = models.DateField()
    marks = models.IntegerField()
    email = models.EmailField()
    phonenumber = models.BigIntegerField()
    address = models.TextField()
```

-->Makemigrations and migrate

- **admin.py**

```
from testapp.models import Student
class StudentAdmin(admin.ModelAdmin):
    list_display = ['rollno','name','dob','marks','email','phonenumber','address']
admin.site.register(Student,StudentAdmin)
```

-->Create super user, then for table in admin interface.
-->Create a file with the name populate.py under modelproject3 folder.

- **populate.py**

```
import os
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'modelproject3.settings')
import django
django.setup()

from testapp.models import Student
from faker import Faker
from random import *
def phonenumbergen():
        d1 = randint(6,9)
        num = '' + str(d1)
        for i in range(9):
                num += str(randint(0,9))
        return int(num)
def populate(n):
    for i in range(n):
        fake = Faker()
        frollno = fake.random_int(min=1,max=999)
        fname = fake.name()
        fdob = fake.date()
        fmarks = fake.random_int(min=1,max=100)
        femail = fake.email()
        fphonenumber = phonenumbergen()
        faddress = fake.address()

Student.objects.get_or_create(rollno=frollno,name=fname,dob=fdob,marks=f
marks,email=femail,phonenumber=fphonenumber,address=faddress)
n = int(input('Enter number of records:'))
populate(n)
print(f'{n} Records Inserted Successfully.....')
```

- **views.py**

```
from testapp.models import Student
def student_view(request):
   student_list = Student.objects.all()
   return render(request,'testapp/std.html',{'student_list':student_list})
```

- **urls.py**

```
path('std/',views.student_view),
```

- **std.html**

```
<body>
   <h1>Student Information</h1>
   {% if student_list %}
   {% for student in student_list%}
   <h2>{{student.name}} Information</h2>
   <ul>
     <li>Student Rollno:{{student.rollno}}</li>
     <li>Student BOD:{{student.dob}}</li>
     <li>Student Marks:{{student.marks}}</li>
     <li>Student Email:{{student.email}}</li>
     <li>Student Phone Number:{{student.phonenumber}}</li>
     <li>Student Address:{{student.address}}</li>
   </ul>
   <br>
   {% endfor %}
   {% else %}
   <p>No records found in the database</p>
   {% endif %}
  </body>
```

- **views.py**

```
student_list = Student.objects.filter(marks__lt=35)
student_list = Student.objects.filter(name__startswith='S')
student_list = Student.objects.all().order_by('marks')#Ascending order of
marks
student_list = Student.objects.all().order_by('-marks')#Descending order
```

Apr 03

<u>How to generate fake data:</u>
By using faker module and our own customized code by using random module.

<u>Another way to generate fake data:</u>
django-seed is a django based customized application to generate fake data for every model automatically.
Internally this application using faker module only.

<u>Steps to use django-seed:</u>
1).pip install django-seed
2).Add 'django_seed' app in our INSTALLED_APPS inside settings.py.
3).generate and send fake data to the models.

```
py manage.py seed testapp --number=10
```

<u>Generate fake data by using django-seed:</u>

```
D:\Django_20MAR_7PM>django-admin startproject modelproject4
D:\Django_20MAR_7PM>cd modelproject4
D:\Django_20MAR_7PM\modelproject4>py manage.py startapp testapp
```

-->Add 'testapp', 'django_seed' in settings.py

- **models.py**

```
class Student(models.Model):
   rollno = models.IntegerField()
   name = models.CharField(max_length=30)
   dob = models.DateField()
   marks = models.IntegerField()
   email = models.EmailField()
   phonenumber = models.CharField(max_length=30)
   address = models.TextField()
```

-->Database(Mysql)
-->Create a database in Mysql(seeddb_7pm)
-->Makemigrations and migrate

- **admin.py**

```
from testapp.models import Student
class StudentAdmin(admin.ModelAdmin):
   list_display = ['rollno','name','dob','marks','email','phonenumber','address']
admin.site.register(Student,StudentAdmin)
```

-->To insert data to the table.

```
py manage.py seed testapp --number=50
```

<u>To connect to Mysql</u>

- **settenigs.py**

```
DATABASES = {
   'default': {
      'ENGINE': 'django.db.backends.mysql',
      'NAME': 'seeddb_7pm',
      'USER':'root',
      'PASSWORD':'root',
      'HOST':'localhost',
      'PORT':3306
   }
}
```

<u>Project Name:SunnyJobs</u>

```
D:\Django_20MAR_7PM>django-admin startproject sunnyjobs
D:\Django_20MAR_7PM>cd sunnyjobs
D:\Django_20MAR_7PM\sunnyjobs>py manage.py startapp testapp
```

-->Add app in settings.py

- **models.py**

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

Mysql database:
jobsdb_7pm

-->Makemigrations and migrate.

- **admin.py**

```
from testapp.models import HydJobs
class HydJobsAdmin(admin.ModelAdmin):
  list_display =
['date','company','title','eligibility','address','email','phonenumber']
admin.site.register(HydJobs,HydJobsAdmin)
```

-->create super user.

- **views.py**

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

- **urls.py**

```
path('', views.homepage_view),
```

- **index.html**

```
<body>
  <div class="container">
   <div class="jumbotron">
    <h1>Welcome To SUNNYJOBS</h1>
    <p>Contenious Job Updates For Every Hour....</p>
    <a class="btn btn-primary btn-lg" href="#" >Hyderabad
Jobs</a>&nbsp&nbsp&nbsp&nbsp
    <a class="btn btn-primary btn-lg" href="#">Pune
Jobs</a>&nbsp&nbsp&nbsp&nbsp
    <a class="btn btn-primary btn-lg" href="#">Bangalore Jobs</a>
   </div>
  </div>
 </body>
```

- **jobs1.css**

```
.container{
 margin-top: 200px;
 text-align: center;
}
.container .jumbotron{
  background: red;
  color: white;
}
.jumbotron a{
  background: yellow;
  color: red;
  border: 2px solid green;
}
```

Apr 04

- **views.py**

```
from testapp.models import HydJobs
def hydjobs_view(request):
    jobs_list = HydJobs.objects.all()
    return render(request,'testapp/hydjobs.html',{'jobs_list':jobs_list})
```

- **hydjobs.html**

```
<body>
    <h1>Hyderabad Jobs Information</h1>
    {% if jobs_list %}
    <table border="3">
      <thead>
        <th>Date</th>
        <th>Company</th>
        <th>Title</th>
        <th>Eligibility</th>
        <th>Address</th>
        <th>Email</th>
        <th>Phone Number</th>
      </thead>
      {% for job in jobs_list%}
      <tr>
        <td>{{job.date}}</td>
        <td>{{job.company}}</td>
        <td>{{job.title}}</td>
        <td>{{job.eligibility}}</td>
        <td>{{job.address}}</td>
        <td>{{job.email}}</td>
      <td>{{job.phonenumber}}</td>
     </tr>
     {% endfor %}
    </table>
    {% else %}
    <p>No Jobs In Hyderabad......</p>
    {% endif %}
  </body>
```

- **populate.py**

```
import os
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'sunnyjobs.settings')
import django
django.setup()

from testapp.models import HydJobs
from faker import Faker
from random import *
fake = Faker()
def phonenumbergen():
        d1 = randint(6,9)
        num = '' + str(d1)
        for i in range(9):
                num += str(randint(0,9))
        return int(num)
def populate(n):
   for i in range(n):
      fdate = fake.date()
      fcompany = fake.company()
      ftitle = fake.random_element(elements=('Project Manager','Team
Lead','Software Engineer','Associate Engineer'))
      feligibility =
fake.random_element(elements=('B.Tech','M.Tech','MCA','Phd','Msc','Mahesh
Sir Student'))
      faddress = fake.address()
      femail = fake.email()
      fphonenumber = phonenumbergen()
      hyd_jobs_record = HydJobs.objects.get_or_create(
      date = fdate,
      company = fcompany,
      title = ftitle,
      eligibility = feligibility,
      address = faddress,
      email = femail,
      phonenumber = fphonenumber
      )
n = int(input('Enter number of records:'))
populate(n)
print(f'{n} Records inserted successfully.....')
```

CHAPTER-5

Working with Django Forms

-->The main purpose of the forms is to take user input.
Ex:
login form, registration form, enquiry form.......

-->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.
-->Here we have to use Django specific forms 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 widget/componets(like text area,email, pwd etc)
very quickly.
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 forms will be come very wasy.

model class--->Converted into database table.
form class-->Converted into html form

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
forms.py=====>views.py=====>Template file(HTML)

Step-2: Usage of forms.py inside views.py file
views.py file is responsible to send this form to the template file.

- **views.py**

```
from testapp.forms import StudentForm
def studentinput_view(request):
    form = StudentForm()
    return render(request,'testapp/input.html',{'form':form})
```

Step-3: Creation of html file to hold form
Inside template file we have to use template tag to inject form {{form}}

- **input.html**

```
<!DOCTYPE html>
{% load static %}
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>Student Form</title>
    <link rel="stylesheet" href="{% static 'css/form1.css' %}">
  </head>
  <body>
    <h1>Student Input Form</h1>
    <div class="container" align='center'>
      <form method="post">
        {{form.as_p}}
        <input type="submit" name="" value="Submit">
      </form>
    </div>
  </body>
</html>
```

- **form1.css**

```
h1{
  text-align: center;
}
body{
  background: yellow;
  color: red;
}
```

- **urls.py**

```
from testapp import views
urlpatterns = [
   path('admin/', admin.site.urls),
   path('std/', views.studentinput_view),
]
```

Apr 05

-->If we submit this form we will get 403 status code response

```
Forbidden (403)
CSRF verification failed. Request aborted.

Reason given for failure:
   CSRF token missing.
```

<u>CSRF(Cross Site Request Forgery) Token:</u>

-->Every form should satisfy CSRF verification, otherwise Django wont 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.

- **input.html**

```
<form method="post">
      {{form.as_p}}
      {% csrf_token %}
</form>
```

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

```
<input type="hidden" name="csrfmiddlewaretoken" value="nbmbmnbmb">
```

The value of this field is keep on changing from request to request. hence it is impossible to forgery of our request.

If we configures csrf_token in html form then only django will accept our form.

<u>How to process input data from the form inside views.py</u>

Inside views.py, we have to read data provided by end user and we have to use that data based on our requirement.

form = StudentForm()--->empty form object to display form to the end user.
form = StudentForm(request.POST)-->This form object contains end user provided data.

cleaned_data-->dictionary which contains end user provided data.
cleaned_data['name']-->The name entered by end user
cleaned_data['marks']-->The marks entered by end user

form.is_valid()-->To check whether validations are successful or not.

<u>Difference between GET & POST:</u>
There are multiple ways to send GET request:

1.Typing URL in the address bar and enter.
2.Clicking on hyperlinks.
3.Submitting the HTML form without method attribute.
4.Submitting the HTML form with method attribute of GET value.

There is only one way to send POST request.

1.Submitting the HTML form with method attribute of POST value.

- **views.py**

```
def studentinput_view(request):
  submitted = False
  sname = ''
  if request.method == 'POST':
    form = StudentForm(request.POST)
    if form.is_valid():
      print('Form validation success and print data')
      print('Name:',form.cleaned_data['name'])
      print('RollNo:',form.cleaned_data['rollno'])
      print('Marks:',form.cleaned_data['marks'])
      submitted = True
      sname = form.cleaned_data['name']
  form = StudentForm()
  return render(request,'testapp/input.html',
{'form':form,'submitted':submitted,'sname':sname})
```

- **input.html**

```
<body>
  <div class="container" align='center'>
```

```
    {% if submitted %}
    <h1>Student with the name:{{sname}} processed successfully</h1>
    <h2>Enter next student information</h2>
    {% else %}
    <h1>Student Input Form</h1>
    {% endif %}
    <form method="post">
     {{form.as_p}}
     {% csrf_token %}
     <input type="submit" name="" value="Submit">
    </form>
   </div>
  </body>
```

Apr 08

Feedback project:

```
D:\Django_20MAR_7PM>django-admin startproject feedbackproject
D:\Django_20MAR_7PM>cd feedbackproject
D:\Django_20MAR_7PM\feedbackproject>py manage.py startapp testapp
```

-->Add app in settings.py

- **forms.py**

```
from django import forms
class FeedBackForm(forms.Form):
   name = forms.CharField()
   rollno = forms.IntegerField()
   email = forms.EmailField()
   feedback = forms.CharField(widget=forms.Textarea)
```

- **views.py**

```
from testapp.forms import FeedBackForm
def feedback_view(request):
   submitted = False
   name = ''
   if request.method == 'POST':
      form = FeedBackForm(request.POST)
      if form.is_valid():
         print('Form validation success and printing feedback information')
         print('*'*55)
         print('Name:',form.cleaned_data['name'])
         print('RollNo:',form.cleaned_data['rollno'])
```

```
            print('Email:',form.cleaned_data['email'])
            print('Feedback:',form.cleaned_data['feedback'])
            submitted = True
            name = form.cleaned_data['name']
    form = FeedBackForm()
    return render(request,'testapp/feedback.html',

            {'form':form,'submitted':submitted,'name':name})
```

- **feedback.html**

```
<body>
    <div class="container" align='center'>
     {% if submitted %}
     <h1>Hello {{name}}, thanks for providing feedback, It is very helpful for us
to provide better service</h1>
     {% else %}
    <h1>Student Feedback Form</h1>
    <form method="post">
     {{form.as_p}}
     {% csrf_token %}
     <input type="submit" name="" value="Submit Feedback">
    </form>
    {% endif %}
    </div>
  </body>
```

- **feed1.css**

```
body{
  background: yellow;
  color:red;
}
```

- **urls.py**

```
path('feed/', views.feedback_view),
```

<u>Form Validations:</u>

-->We can implement validation logic by using 2-ways.

1.Explicitly by the programmer by using clean mmethods

2.By using django in-built validators.

-->Total validation logic should be written inside forms.py

Basic OOP knowledge:

```
class Parent:
        def __init__(self):
                self.x = 333
        def property(self):
                print('gold + land + cash')
class Child(Parent):
        def education(self):
                print('B-Tech qualification + Job')
c = Child()
c.education()
c.property()
print(c.x)
```

Ex:

```
class Form:
        def __init__(self):
                self.cleaned_data = {'name':'sunny'}

class FeedBackForm(forms.Form):
   name = forms.CharField()
   rollno = forms.IntegerField()
   email = forms.EmailField()
   feedback = forms.CharField(widget=forms.Textarea)

from = FeedBackForm()
form.cleaned_data['name']
```

Ex:

```
class Parent:
        def marry(self):
                print('Marry Appalamma')
class Child(Parent):
        def marry(self):
                print('Marry Katrina Kaif')
c = Child()
c.marry()
```

1.Explicitly by the programmer by using clean mmethods
Syntax for clean method:clean_fieldname(self)

-->In the Form class, for any field, if we define clean method, then at the time of submit this form, django will call this method automatically to perform validations. If clean method wont raise any error then only request will be processed.

- **forms.py**

```
        def clean_name(self):
    print('Validating name field')
    inputname = self.cleaned_data['name']
    if len(inputname) < 4:
       raise forms.ValidationError('The minimum number of characters in the
name field should be 4')
    return inputname
        def clean_rollno(self):
    print('Validating rollno field')
    inputrollno = self.cleaned_data['rollno']
    return inputrollno
  def clean_email(self):
    print('Validating email field')
    inputrollno = self.cleaned_data['email']
    return inputrollno
  def clean_feedback(self):
    print('Validating feedback field')
    inputrollno = self.cleaned_data['feedback']
    return inputrollno
```

Note:

Django will call these field level clean methods automatically and we are not required to call these methods explicitly. The names are fixed because these are understandable by Django.

2).By using django inbuilt validators:
Django provides several inbuilt validators to perform very common validations.
We can use directly and we are not responsible to implement those.
All inbuilt validators present in django.core module

- **forms.py**

```
from django.core import validators
class FeedBackForm(forms.Form):
   feedback = forms.CharField(widget=forms.Textarea,validators=
           [validators.MaxLengthValidator(40),validators.MinLengthValidator(10)
])
```

How to implement custom validators by using same validators parameter?
Ex: The name should starts with 's'

- **forms.py**

```
class FeedBackForm(forms.Form):
   def starts_with_s(value):
      print('starts_with_s function execution')
      if value[0].lower() != 's':
         raise forms.ValidationError('Name should be starts with s or S')

   name = forms.CharField(validators=[starts_with_s])
```

<u>mail should contains @gmail.com</u>

```
mail = 'mahesh@gmail.com'
print(mail[-10:])

def gmail_validator(value):
      print('Checking for gmail validation')
      if value[-10:] != '@gmail.com':
         raise forms.ValidationError('Mail extension should be gmail')

email = forms.EmailField(validators=[gmail_validator])
```

<u>Validation of total form directly by using single clean() method</u>
We are not required to write separate field level methods. Inside single clean method all validations we can perform.

```
def clean(self):
      print('Total form validation......')
      total_cleaned_data = super().clean()
      print('Validating Name')
      inputname = total_cleaned_data['name']
      if inputname[0].lower() != 's':
         raise forms.ValidationError('Name should be starts with s')
```

```
        print('Validating Rollno')
        inputrollno = total_cleaned_data['rollno']
        if inputrollno <= 0:
           raise forms.ValidationError('Rollno should be > 0')
        print('Validating email')
        inputemail = total_cleaned_data['email']
        if inputemail[-10:] != '@gmail.com':
           raise forms.ValidationError('Email extension should be gmail')
```

-->If we want to validate multiple field values together, then single clean() method is the best choice.

How to check original pwd and re-entered pwd are same or not:

```
class FeedBackForm(forms.Form):
   password = forms.CharField(label='Enter Password',
widget=forms.PasswordInput)
   rpassword =
forms.CharField(label='Password(Again)',widget=forms.PasswordInput)

def clean(self):
        total_cleaned_data = super().clean()
        pwd = total_cleaned_data['password']
        rpwd = total_cleaned_data['rpassword']
        if pwd != rpwd:
           raise forms.ValidationError('Both passwords must be same.....')
```

CHAPTER-6

Working with Model Forms:

Apr 12

Model Form-->Model based form

name = form.cleaned_data['name']
marks = form.cleaned_data['marks']

records = Student.get_or_create(name=name,marks=marks)

10-->fields are there
100 fields

Model based form
only one line
form.save()

-->Sometimes we can create form based on Model, such type of forms are called as Model Based Forms or Model Forms.

-->The main advantage of model forms is we can grab the end user input data and we can save that data very easily in the database.

```
        form.save()

        form.save(commit=True)
```

<u>How to develop model based forms:</u>

1).While developing form class, we have to inherit from forms.ModelForm class instead of forms.Form class

```
        class RegistrationForm(forms.Form)--->Normal Form
        class RegistrationForm(forms.ModelForm)-->Model Form
```

```
2).class RegistrationForm(forms.ModelForm):
                class Meta:
                        model = Student
                        fields = '__all__'
```

Case-1:Instead of all fields, if we required particular fields.

```
        class Meta:
            model = Student
            fields = ('field1','field2','field3')
```

Case-2:Instead of all fields, if we want to exclude certain fields.

```
        class Meta:
            model = Student
            exclude = ['field1','field2']
```

Ex:ModelFormProjct

```
D:\Django_20MAR_7PM>django-admin startproject ModelFormProject
D:\Django_20MAR_7PM>cd ModelFormProject
D:\Django_20MAR_7PM\ModelFormProject>py manage.py startapp testapp
```

-->Add app in settings.py

- **models.py**

```
class Student(models.Model):
   name = models.CharField(max_length=30)
   marks = models.IntegerField()
```

-->Makemigrations and migrate.

- **admin.py**

```
from testapp.models import Student
class StudentAdmin(admin.ModelAdmin):
   list_display = ['name','marks']
admin.site.register(Student,StudentAdmin)
```

- **forms.py**

```
from django import forms
from testapp.models import Student
class StudentForm(forms.ModelForm):
   name = forms.CharField()
   marks = forms.IntegerField()
   class Meta:
      model = Student
      field = '__all__'
```

- **views.py**

```
from testapp.forms import StudentForm
def student_view(request):
   if request.method == 'POST':
      form = StudentForm(request.POST)
      if form.is_valid():
         form.save(commit = True)
         print('Record inserted into DB successfully....')
   form = StudentForm()
   return render(request,'testapp/studentform.html',{'form':form})
```

- **studentform.html**

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

- **urls.py**

```
path('register/',views.student_view)
```

- **std1.css**

```
body{
  background: yellow;
  color:red;
}
h1{
  text-align: center;
}
```

Ex:MovieProject

```
D:\Django_20MAR_7PM>django-admin startproject MovieProject
D:\Django_20MAR_7PM>cd MovieProject
D:\Django_20MAR_7PM\MovieProject>py manage.py startapp testapp
```

-->Add app in settings.py

- **models.py**

```
class Movie(models.Model):
   rdate = models.DateField()
   moviename = models.CharField(max_length=20)
   hero = models.CharField(max_length=20)
   heroine = models.CharField(max_length=20)
   rating = models.FloatField()
```

-->makemigrations and migrate

- **admin.py**

```
from testapp.models import Movie
class MovieAdmin(admin.ModelAdmin):
   list_display = ['rdate','moviename','hero','heroine','rating']
admin.site.register(Movie,MovieAdmin)
```

- **views.py**

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

- **urls.py**

```
path('',views.index_view)
```

- **index.html**

```
<body>
   <div class="container" align='center'>
     <h1>Sunny Movies.......</h1>
     <p>Upto date movie information</p>
     <a href="#" class="btn btn-primary">ADD</a>
     <a href="#" class="btn btn-primary">List Movies</a>
   </div>
  </body>
```

Apr 13

<u>MovieProject:</u>

- **views.py**

```
from testapp.models import movie
def list_movies_view(request):
   movies_list = Movie.objects.all()
   return render(request,'testapp/listmovies.html',{'movies_list':movies_list})
```

- **listmovies.html**

```
<body>
   {% if movies_list %}
   <h1>Movie List</h1>
   <table>
     <thead>
       <th>Release Date</th>
       <th>Movie Name</th>
       <th>Hero</th>
       <th>Heroine</th>
       <th>Rating</th>
     </thead>
     {% for movie in movies_list %}
     <tr>
       <td>{{movie.rdate}}</td>
       <td>{{movie.moviename}}</td>
       <td>{{movie.hero}}</td>
       <td>{{movie.heroine}}</td>
       <td>{{movie.rating}}</td>
     </tr>
     {% endfor %}
   </table>
   {% else %}
   <h1>No Movies Found</h1>
   {% endif %}
  </body>
```

- **urls.py**

```
path('movielist/', views.list_movies_view),
```

- **index.html**

```
<a href="/movielist" class="btn btn-primary">List Movies</a>
```

- **forms.py**

```
from django import forms
from testapp.models import Movie
class MovieForm(forms.ModelForm):
   class Meta:
      model = Movie
      fields = '__all__'
```

- **views.py**

```
from testapp.forms import MovieForm
def add_movie_view(request):
   form = MovieForm()
   if request.method == 'POST':
      form = MovieForm(request.POST)
      if form.is_valid():
         form.save(commit=True)
      return index_view(request)
   return render(request,'testapp/addmovie.html',{'form':form})
```

- **addmovie.html**

```
<body>
   <div class="container" align='center'>
     <h1>Add Movie Information</h1>
     <form method="post">
      {{form.as_p}}
      {% csrf_token %}
      <input type="submit" class="btn btn-primary" name="" value="Add
Movie">
     </form>
   </div>
</body>
```

- **urls.py**

```
path('addmovie/', views.add_movie_view)
```

- **index.html**

```
<a href="/addmovie" class="btn btn-primary">ADD</a>
```

CHAPTER-7

Working With Advanced Templates:

1.Template Inheritance
2.Template Filters

1.Template Inheritance:
-->If multiple template files have same 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 template file, which is also known as base template. The remaining template files should required to extend base template so that the common code will be inherited automatically.
-->Inheriting common code from base template to remaining templates is nothing but template inheritance.

How to implement template inheritance

- **base.html**

```
<body>
                    common code required for every child template
                    {% block child_block %}
                              Anything out of this block available to child tag
                              In child template the specific code should be in this
block
                    {% endblock %}
</body>
```

- **child.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
          {% block child_block %}
                    Child specific code
          {% endblock %}
```

Apr 15

Ex:

```
D:\Django_20MAR_7PM>django-admin startproject advtemplateproject
D:\Django_20MAR_7PM>cd advtemplateproject
```

```
D:\Django_20MAR_7PM\advtemplateproject>py manage.py startapp testapp
```

-->Add app in settings.py

- **base.html**

```
<body>
  <nav class='navbar'>
    <a class='navbar' href="#">Mahesh News</a>
    <div class="container">
      <ul class='navbar-nav'>
        <li><a class='navbar' href="#">Movies</a></li>
        <li><a class='navbar' href="#">Sports</a></li>
        <li><a class='navbar' href="#">Politics</a></li>
      </ul>
    </div>
  </nav>
  {% block body_block %}
  {% endblock %}
 </body>
```

- **views.py**

```
def base_view(request):
   return render(request,'testapp/base.html')
```

- **urls.py**

```
path('', views.base_view)
```

- **movie.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% load static %}
{% block body_block %}
<h1>This is movies information</h1>
{% endblock %}
```

- **views.py**

```
def movie_view(request):
   return render(request,'testapp/movie.html')
```

- **urls.py**

```
path('movie/',views.movie_view)
```

- **sports.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% load static %}
{% block body_block %}
<h1>This is sports information</h1>
{% endblock %}
```

- **views.py**

```
def sports_view(request):
   return render(request,'testapp/sports.html')
```

- **urls.py**

```
path('sports/', views.sports_view),
```

- **politics.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% load static %}
{% block body_block %}
<h1>This is politics information</h1>
{% endblock %}
```

- **views.py**

```
def politics_view(request):
   return render(request,'testapp/politics.html')
```

- **urls.py**

```
path('politics/', views.politics_view),
```

Note:
1).In the parent template, we can define any number of blocks. But child template is responsible to implement these blocks.
2).It is not mandatory to implement every parent block in child. Based on our requirement, child template can decide which blocks are required to implement.

3).While implementing blocks in child template, it i snot required to follow order.

Advantages of Template Inheritance:
1).What ever code available in base template is bydefault available to child template and we are not required to write again. Hence it promotes code re-usability.
2).It reduces length of the code and improves readability.
3).It reduce development time.
4).It provides unique and same look and feel for total web application.

Template Filters:
-->In the template file, the injected data can be displayed by using template tags.

```
{{name}}
{{emp.eno}}
```

-->Before displaying to the end user, if we want to perform some modifications to the injected text, like add something or cut something, case conversions etc then we should go for filters.

Syn for Template Filter:

```
{{value | filtername}}
{{value | filtername:'argument'}}
```

-->Arguments to the filter are optional.

Ex:

```
D:\Django_20MAR_7PM>django-admin startproject filterproject
D:\Django_20MAR_7PM>cd filterproject
D:\Django_20MAR_7PM\filterproject>py manage.py startapp testapp
```

-->Add app in settings.py

- **models.py**

```
class FilterModel(models.Model):
   name = models.CharField(max_length=20)
   subject = models.CharField(max_length=20)
   dept = models.CharField(max_length=20)
   date = models.DateField()
```

-->makemigrations and migrate

- **admin.py**

```
from testapp.models import FilterModel
class FilterModelAdmin(admin.ModelAdmin):
   list_display = ['name','subject','dept','date']
admin.site.register(FilterModel,FilterModelAdmin)
```

- **views.py**

```
from testapp.models import FilterModel
def upper_data_view(request):
   records = FilterModel.objects.all()
   return render(request,'testapp/upperdata.html',{'records':records})
```

- **upperdata.html**

```
<body>
   {% for record in records%}
   <h1>{{record.name}} Information</h1>
   <ul>
     <li>Name:{{record.name | upper}}</li>
     <li>Subject:{{record.subject | lower}}</li>
     <li>Dept:{{record.dept | title}}</li>
     <li>Date:{{record.date | date:"d-m-Y"}}</li>
   </ul><hr>
   {% endfor %}
</body>
```

- **urls.py**

```
path('upperview/', views.upper_data_view)
```

date filters

```
<li>Date:{{record.date | date:"d-m-Y"}}</li>
<li>Date:{{record.date | date:"m-d-Y"}}</li>
<li>Date:{{record.date | date:"m/d/Y"}}</li>
<li>Date:{{record.date | timesince}}</li>
<li>Date:{{record.date | date:'d-b-Y'}}</li>
<li>Date:{{record.date | date:'l,F j, Y'}}</li>
```

<u>Template Filters:</u>
<u>How to create our own filters:</u>
-->Based on our requirement, we can create own filters, If predefined filters are not fullfill our requirement.

<u>Steps:</u>
1).Create a folder 'templatetags' inside our application.
2).Create a special file named with '__init__.py' inside this folder(templatetags), so that django will consider this folder as python package.
3).Create a python file inside this folder to define our filters cust_filters.py

- **cust_filters.py**

```
from django import template
register = template.Library()

def first_five_upper(value):
   result = value[:3].upper()
   return result

def first_n_upper(value,n):
   result = value[:n].upper()
   return result

register.filter('ffu',first_five_upper)
register.filter('fnu',first_n_upper)
```

- **html file**

```
<body>
   {% load cust_filters %}
            <li>Name:{{record.name | ffu}}</li>
            <li>Name:{{record.name | fnu:6}}</li>
```

CHAPTER-8

Session Management:

-->Client and server can communicate with some common language which is nothing but HTTP.

-->The basic limitation of HTTP is it is stateless protocol. i.e it is unable to remeber client information for future purpose across multiple requests. Every request to the server is treated as new request only.

-->To remember client information at server side, some special mechanism must be required which is nothing but session management mechanism.

Different session management mechanisms are:

1).Cookies
2).Session API

<u>Session management by using cookies:</u>
Coockies is a very small amount of information created by server and maintained by client. Cookies are key-value pairs.

To add cookies to the response.

```
response.set_cookie(cname,cvalue)
```

To get cookcies send by client:

```
request.COOKIES[cname]
request.COOKIES.get(cname)
request.COOKIES.get(cname,defaultvalue)
```

Ex:

```
>>> d={101:'sunny'}
>>> d[101] #'sunny'
>>> d[102] #KeyError: 102
>>> print(d.get(101)) #sunny
>>> print(d.get(102)) #None
>>> print(d.get(101,'pinny')) #sunny
>>> print(d.get(102,'pinny')) #pinny
```

Page Count Application

```
D:\Django_20MAR_7PM>django-admin startproject pagecountproject
D:\Django_20MAR_7PM>cd pagecountproject
D:\Django_20MAR_7PM\pagecountproject>py manage.py startapp testapp
D:\Django_20MAR_7PM\pagecountproject>py manage.py makemigrations
D:\Django_20MAR_7PM\pagecountproject>py manage.py migrate
```

- **views.py**

```
def page_count_view(request):
   print('Cookies from the client:',request.COOKIES)
   count = int(request.COOKIES.get('count',0))
   count += 1
   response = render(request,'testapp/counter.html',{'count':count})
   response.set_cookie('count',count)
   return response
```

- **counter.html**

```
<body>
   <h1>The Page Count:{{count}}></h1>
</body>
```

- **urls.py**

```
path('page/',views.page_count_view),
```

Apr 18

Client is sending the first request:

-->What information required for future purpose, sever will decide, with that information, server will create cookie and it should be handover to the client. Response will be send to the client.

-->Servere will add cookie to the response then automatically cookies will be reached to the client.

cookies--->per server and per browser
from chrome--->facebook, gmail

sessionproject2

```
D:\Django_20MAR_7PM>django-admin startproject sessionproject2
D:\Django_20MAR_7PM>cd sessionproject2
D:\Django_20MAR_7PM\sessionproject2>py manage.py startapp testapp
```

-->Add app in settings.py
-->makemigrations and migrate.

- **forms.py**

```
from django import forms
class LoginForm(forms.Form):
   name = forms.CharField()
```

- **views.py**

```
from testapp.forms import LoginForm
def home_view(request):
   form = LoginForm()
   return render(request,'testapp/home.html',{'form':form})
```

- **home.html**

```
<body>
   <h1>Welcome To Naresh IT</h1>
   <form action="">
     {{form.as_p}}
     {% csrf_token %}
     <input type="submit" name="" value="Enter Name">
   </form>
  </body>
```

- **urls.py**

```
path('home/',views.home_view)
```

- **views.py**

```
def date_time(request):
   name = request.GET['name']
   response = render(request,'testapp/datetime.html',{'name':name})
   response.set_cookie('name',name)
   return response
```

- **datetime.html**

```
<body>
   <h1>Hello {{name}}</h1>
   <a href="#">Click Here To Get The Date and Time</a>
</body>
```

- **urls.py**

```
path('second/',views.date_time)
```

- **home.html**

```
<form action="/second">
```

- **views.py**

```
import datetime
def result_view(request):
   name = request.COOKIES.get('name')
   date_time = datetime.datetime.now()
   return
render(request,'testapp/results.html',{'name':name,'date_time':date_time})
```

- **results.html**

```
<body>
   <h1>Hello {{name}}</h1>
   <h1>Current Date and Time:{{date_time}}</h1>
   <a href="/results">Click here to updated time</a>
</body>
```

- **urls.py**

```
path('results/', views.result_view),
```

- **datetime.html, results.html**

```
 <a href="/results">Click Here To Get The Date and Time</a>
 <a href="/results">Click here to updated time</a>
```

Flow:
http://127.0.0.1:8000/home/
-->home.html will be displayed.
-->If we submit then request will go to the url-pattern:/second which is associated with views.date_time
-->This view function sends datetime.html as response and in that response name cookie added for the future purpose.

Session Project-3:

```
D:\Django_20MAR_7PM>django-admin startproject sessionproject3
```

```
D:\Django_20MAR_7PM>cd sessionproject3
D:\Django_20MAR_7PM\sessionproject3>py manage.py startapp testapp
```

-->Add app in settings.py

-->Makemigrations and migrate

- **views.py**

```
def home_view(request):
   return render(request,'testapp/home.html')
```

- **home.html**

```
<body>
   <h1>Welcome To NAresh IT</h1>
   <form action="">
     Enter Name:<input type="text" name="name" value=""><br><br>
     <input type="submit" name="" value="Submit Name">
   </form>
 </body>
```

- **urls.py**

```
path('home/',views.home_view)
```

- **views.py**

```
def age_view(request):
   print(request.COOKIES)
   username = request.GET['name']
   response = render(request,'testapp/age.html',{'name':username})
   response.set_cookie('name',username)
   return response
```

- **age.html**

```
<body>
   <h1>Hello {{name}}</h1>
   Enter Age:<input type="text" name="" value=""><br><br>
   <input type="submit" name="" value="Submit Age">
 </body>
```

- **urls.py**

```
path('age/',views.age_view)
```

- **home.html**

```
<form action="/age">
```

Apr 19

- **views.py**

```
def gf_view(request):
   print(request.COOKIES)
   username = request.COOKIES['name']
   age = request.GET['age']
   response = render(request,'testapp/gf.html',{'name':username})
   response.set_cookie('age',age)
   return response
```

- **gf.html**

```
<body>
   <h1>Hello {{name}}</h1>
   <form action="">
      Enter GF Name:<input type="text" name="gf" value=""><br><br>
      <input type="submit" name="" value="Submit GF Name">
   </form>
</body>
```

- **urls.py**

```
path('gf/',views.gf_view)
```

- **views.py**

```
def result_view(request):
   print(request.COOKIES)
   username = request.COOKIES['name']
   age = request.COOKIES['age']
   gfname = request.GET['gf']
   return
render(request,'testapp/results.html',{'name':username,'age':age,'gf':gfname})
```

- **results.html**

```
<body>
   <h1>Hello {{name}}, thanks for providing information</h1>
   <h2>Please cross check your data and confirm</h2>
   <ul>
    <li>Name:{{name}}</li>
    <li>Age:{{age}}</li>
    <li>GF Name:{{gf}}</li>
  </ul>
</body>
```

- **urls.py**

```
path('results/',views.result_view)
```

<u>Temporary Vs Permanent cookies</u>

If we are not setting any max_age for the cookie, then cookies will be stored in borwsers cache. Once we close the browser, automatically the cookies will be expired.
Such type of cookies are called as temporary cookies.

We can create temporary cookies:

```
        response.set_cookie(name,value)
```

If we are setting max_age for the cookies, then cookies will be stored in local file system permanently. Once the specified max_age expired then onnly cookies will be expired---->Permanent cookies or persistent cookies.

We can create permanent cookies:

```
        response.set_cookie(name,value,seconds)
```

max_age:3-months==>3*30*24*60*60

<u>sessionproject-4</u>

```
D:\Django_20MAR_7PM>django-admin startproject sessionproject4
D:\Django_20MAR_7PM>cd sessionproject4
D:\Django_20MAR_7PM\sessionproject4>py manage.py startapp testapp
```

-->Add app in settings.py
-->Makemigrations and migrate

- **views.py**

```
def index_view(request):
  return render(request,'testapp/home.html')
```

- **home.html**

```
<body>
  <div class="container">
    <div class="container" align = 'center'>
```

```
        <h1>MAHESH ONLINE SHOPPING APP</h1>
        <a class="btn btn-danger" href="#">Add Item</a>
        <a class="btn btn-danger" href="#">Display Items</a>
      </div>
    </div>
</body>
```

- **urls.py**

```
path('',views.index_view)
```

- **forms.py**

```
from django import forms
class AddItemForm(forms.Form):
    itemname = forms.CharField()
    quantity = forms.IntegerField()
```

- **views.py**

```
from testapp.forms import AddItemForm
def additem_view(request):
    form = AddItemForm()
    response = render(request,'testapp/additem.html',{'form':form})
    if request.method == 'POST':
        form = AddItemForm(request.POST)
        if form.is_valid():
            name = form.cleaned_data['itemname']
            quantity = form.cleaned_data['quantity']
            response.set_cookie(name,quantity)
    return response
```

- **additem.html**

```
<body>
    <div class="container" align='center'>
      <h1>Add Item Form</h1>
      <form method="post">
        {{form.as_p}}
        {% csrf_token %}
        <input type="submit" name="" value="Add Item">
      </form>
    </div>
</body>
```

- **urls.py**

```
path('additem/', views.additem_view),
```

Apr 20

- **views.py**

```
def display_items_view(request):
  return render(request,'testapp/displayitems.html')
```

- **displayitems.html**

```
<body>
  <h1>Shopping Cart Items</h1>
  {% if request.COOKIES %}
  <table border="3">
    <thead>
      <th>Item Name</th>
      <th>Quantity</th>
    </thead>
    {% for key,value in request.COOKIES.items %}
    {% if key != 'csrftoken'%}
    <tr>
      <td>{{key}}</td>
      <td>{{value}}</td>
    </tr>
    {% endif %}
    {% endfor %}
  </table>
  {% else %}
  <p>No Items in the shopping cart</p>
  {% endif %}
 </body>
```

- **urls.py**

```
path('displayitems/',views.display_items_view)
```

Limitations of Cookies:

1.By using cookies we can store very less amount of information. The size of cookies is fixed. Hence if we want to store huge amount of information then cookies is not best choice.
2.Cookies can hold only string information. If we want to store non-string objects we cannot use cookies.

3.Cookies information is stored on client side and hence there is no security.
4.Every time with every request, browser will send all cookies related to that application, which creates network traffic problems.
5). There is a limit on max number of cookies supported by browser.

To overcome these limitations, we should go for Session Framework.

Django provides in-built support for session api

```
INSTALLED_APPS = [
   'django.contrib.sessions',
]

MIDDLEWARE = [
           'django.contrib.sessions.middleware.SessionMiddleware',
]
```

<u>Session Management by using session API(Django Session Framework)</u>
Coockies---->RMP Doctor
Session Framework--->Bank Locker | Temple Chappal Stand | Super speciality hospitals

<u>Useful methods for session management:</u>
1).To add data to the session
```
request.session['key'] = value
```

2).To get the data from the session
```
value = request.session['key']
```

3).request.session.set_expiry(seconds)
set expiry time for the session.

4).request.session.get_expiry_age()
Returns expiry age in seconds(The number of seconds until this session expires)

5).request.session.get_expiry_date()
Return the date on which session will be expired.

App:

```
D:\Django_20MAR_7PM>django-admin startproject sessionproject5
D:\Django_20MAR_7PM>cd sessionproject5
D:\Django_20MAR_7PM\sessionproject5>py manage.py startapp testapp
```

-->Add app in settings.py
-->Makemigrations and migrate

- **views.py**

```
def page_count_view(request):
   print(request.COOKIES)
   count = request.session.get('count',0)
   count += 1
   request.session['count'] = count
   request.session.set_expiry(120)
   print(request.session.get_expiry_age())
   print(request.session.get_expiry_date())
   return render(request,'testapp/pagecount.html',{'count':count})
```

- **pagecount.html**

```
<body>
   <h1>The Page Count:<span>{{count}}</span></h1>
</body>
```

- **urls.py**

```
path('count/',views.page_count_view)
```

Apr 23

Profile Application:

```
            Name:
            Age:
            GF:
            Please check your complete info
```

```
D:\Django_20MAR_7PM>django-admin startproject sessionproject6
D:\Django_20MAR_7PM>cd sessionproject6
D:\Django_20MAR_7PM\sessionproject6>py manage.py startapp testapp
```

-->Add app in settings.py

-->makemigrations and migrate

- **forms.py**

```
from django import forms
class NameForm(forms.Form):
   name = forms.CharField()
```

- **views.py**

```
from testapp.forms import NameForm
def name_view(request):
   form = NameForm()
   return render(request,'testapp/name.html',{'form':form})
```

- **name.html**

```
<body>
   <h1>Name Registration Form</h1>
   <form action="">
     {{form.as_p}}
     {% csrf_token %}
     <input type="submit" name="" value="Submit Name">
   </form>
</body>
```

- **urls.py**

```
path('name/',views.name_view)
```

- **forms.py**

```
class AgeForm(forms.Form):
   age = forms.IntegerField()
```

- **views.py**

```
def age_view(request):
   name = request.GET['name']
   request.session['name'] = name
   form = AgeForm()
   return render(request,'testapp/age.html',{'form':form,'name':name})
```

- **age.html**

```
<body>
   <h1>Hello {{name}}, Provide Age</h1>
   <form action="">
```

```
     {{form.as_p}}
     {% csrf_token %}
     <input type="submit" name="" value="Submit Age">
   </form>
</body>
```

- **urls.py**

```
path('age/',views.age_view)

name.html
<form action='/age'>
```

- **forms.py**

```
class GfForm(forms.Form):
   name = forms.CharField()
```

- **views.py**

```
def gf_view(request):
   age = request.GET['age']
   request.session['age'] = age
   name = request.session['name']
   form = GfForm()
   return render(request,'testapp/gf.html',{'form':form,'name':name})
```

- **gf.html**

```
<body>
   <h1>Hello {{name}}, Provide Girl Friend Name</h1>
   <form action="">
     {{form.as_p}}
     {% csrf_token %}
     <input type="submit" name="" value="Submit GF Name">
   </form>
  </body>
```

- **views.py**

```
def results_view(request):
   gf = request.GET['gf']
          request.session['gf'] = gf
   name = request.session['name']
   age = request.session['age']
   return render(request,'testapp/results.html')
```

- **results.html**

```
<body>
  {% if request.session %}
  <h1>Thanks for providing information...please confirm once</h1>
  <ul>
  {% for key,value in request.session.items %}
  <li>{{key | upper}}:{{value | title}}</li>
  {% endfor %}
  </ul>
  {% else %}
  <h1>No information available</h1>
  {% endif %}
</body>
```

- **urls.py**

```
path('results/',views.results_view)
```

Cart Application with Session API

CHAPTER-9

Authentication and Authorization:

Authentication: The process of validating user.
Authorization: The process of validating access permission of a user.

login page
uname and password--->should be stored for future purpose
validation must be required

auth application
1).django.contrib.auth
2).django.contrib.contenttypes

Demo app:

```
D:\Django_20MAR_7PM>django-admin startproject authproject
D:\Django_20MAR_7PM>cd authproject
D:\Django_20MAR_7PM\authproject>py manage.py startapp testapp
```

-->Add app in settings.py
-->makemigrations and migrate

- **base.html**

```
<body>
  <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
   <div class="container">
    <a class="navbar-brand" href="#">MAHESHEXAMS</a>
    <ul class="navbar-nav mr-auto">
     <li class="nav-item active">
      <a class="nav-link" href="#">Home <span class="sr-only"></span></a>
     </li>
     <li class="nav-item">
      <a class="nav-link" href="#">Java Exams</a>
     </li>
     <li class="nav-item">
      <a class="nav-link" href="#">Python Exams</a>
     </li>
```

```
          <li class="nav-item">
            <a class="nav-link" href="#">Aptitude Exams</a>
          </li>
        </ul>
        <ul class="navbar-nav ml-auto">
          <li class="nav-item">
            <a class="nav-link" href="#">Signup</a>
          </li>
          <li class="nav-item">
            <a class="nav-link" href="#">Login</a>
          </li>
          <li class="nav-item">
            <a class="nav-link" href="#">Logout</a>
          </li>
          </ul>
      </div>
    </nav>
      {% block body_block%}
      {% endblock %}
  </body>
```

- **home.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block %}
<div class="jumbotron">
  <div class="container">
    <h1>Welcome To MAHESH EXAMS</h1>
    <h2>Rules:</h2>
    <ul>
      <li>Rule-1:You should write only one exam per day</li>
    </ul>
  </div>
</div>
{% endblock %}
```

- **views.py**

```
def home_page_view(request):
    return render(request,'testapp/home.html')
```

- **urls.py**

```
path('', views.home_page_view),
```

Apr 24

- **views.py**

```
def java_page_view(request):
   return render(request,'testapp/javaexams.html')
```

- **javaexams.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block%}
<div class="jumbotron">
  <div class="container" align='center'>
    <h1>Welcome To Java Exams</h1>
  </div>
</div>
{% endblock %}
```

- **urls.py**

```
path('java/', views.java_page_view)
```

- **views.py**

```
def python_page_view(request):
   return render(request,'testapp/pythonexams.html')
```

- **pythonexams.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block%}
<div class="jumbotron">
  <div class="container" align='center'>
    <h1>Welcome To Python Exams</h1>
  </div>
</div>
{% endblock %}
```

- **urls.py**

```
path('python/', views.python_page_view)
```

- **views.py**

```
def aptitude_page_view(request):
   return render(request,'testapp/aptitudeexams.html')
```

- **aptitudeexams.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block%}
<div class="jumbotron">
  <div class="container" align='center'>
    <h1>Welcome To Aptitude Exams</h1>
  </div>
</div>
{% endblock %}
```

- **urls.py**

```
path('aptitude/', views.aptitude_page_view),
```

auth application will use one table:user
Create super user and go to admin interface then add user.

```
from django.contrib.auth.decorators import login_required
@login_required
def java_page_view(request):
   return render(request,'testapp/javaexams.html')
```

start server send request:

    http://127.0.0.1:8000/
    Click on javaexams

Problem-1

```
Page not found (404)
Request Method:        GET
Request URL:    http://127.0.0.1:8000/accounts/login/?next=/java/
```

Solved this problem by including auth application url

project level urls.py

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

Problem-2
TemplateDoesNotExist at /accounts/login/
Exception Value:
registration/login.html

Create a folder under templates with the name registration and create a file login.html

- **login.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block%}
<div class="jumbotron">
  <div class="container" align='center'>
    <h1>Please Login To Write Exams....</h1>
    <form method="post">
      {{form.as_p}}
      {% csrf_token %}
      <input type="submit" name="" value="Login">
    </form>
  </div>
</div>
{% endblock %}
```

Implementing login functionality
Note:
login page url pattern from auth application:accounts/login

If we click on login button, after login the next page is profile page bydefault. If we want to configure our own destination page, we have to write a line in settings.py

Step-1: Update base.html as:

```
<a class="nav-link" href="/accounts/login">Login</a>
```

Step-2: Configure destination page in settings.py

```
LOGIN_REDIRECT_URL = '/'
```

Implement logout functionality:
Step-1: Update logout link with the url pattern:/accounts/logout

```
                              <a class="nav-link"
href="/accounts/logout">Logout</a>
```

Step-2: The default destination page for logout is admin logout page.
We can configure our own logout destination page

```
LOGOUT_REDIRECT_URL = '/'
```

Here logout destination page:home page(/)

Defining separate logout destination page:

- **logout.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block%}
<div class="jumbotron">
  <div class="container" align='center'>
    <h1>Thanks for Using MAHESHEXAMS</h1>
      <p>Anyway we are feeling sad b'z you logout, atleast 24-hrs per day you
have to use our app....</p>
    <h2>Please login again to enjoy........</h2>
    <a href="/accounts/login" class="btn btn-success">Login</a>
  </div>
</div>
{% endblock %}
```

- **views.py**

```
def logout_view(request):
   return render(request,'testapp/logout.html')
```

- **urls.py**

```
path('logout/', views.logout_view)
```

- **settings.py**

```
LOGOUT_REDIRECT_URL = '/logout'
```

Apr 25

Implementing signup button functionality

-->auth application having form class to provide login form.
-->But auth application does not contain any form class for signup functionality.

-->If a user signup, compulsory that information should be stored in database(user table)
-->Display form to signup and that information should be stored inside database directly. For such type of requirement it is highly recommended to go for model based form.

- **forms.py**

```
from django import forms
from django.contrib.auth.models import User
class SignUpForm(forms.ModelForm):
   class Meta:
      model = User
      fields = ['username','password','email','first_name','last_name']
```

- **views.py**

```
from testapp.forms import SignUpForm
def signup_view(request):
   form = SignUpForm()
   return render(request,'testapp/signup.html',{'form':form})
```

- **signup.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block%}
<div class="jumbotron">
  <div class="container">
    <h1>Please SignUp To Write Exams....</h1>
    <form method="post">
      {{form.as_p}}
      {% csrf_token %}
      <input type="submit" name="" value="SignUp">
    </form>
  </div>
</div>
{% endblock %}
```

- **urls.py**

```
path('signup/',views.signup_view)
```

- **base.html**

```
<a class="nav-link" href="/signup">Signup</a>
```

- **views.py**

```
from testapp.forms import SignUpForm
from django.http import HttpResponseRedirect
def signup_view(request):
    form = SignUpForm()
    if request.method == 'POST':
        form = SignUpForm(request.POST)
        user = form.save()
        user.set_password(user.password)#to hash password
        user.save()
        return HttpResponseRedirect('/accounts/login')
    return render(request,'testapp/signup.html',{'form':form})
```

-->In django auth application, User model, the password should not be saved directly. It should be hashed by using some security algorithm. The default password hashing algorithm:pbkdf2_sha256

form.save():We are trying to save password in the plain text form.
Invalid password format or unknown hashing algorithm.

Password hashers:

The default password hasher:pbkdf2_sha256
We can use other more secured password hashers also
like argon2, bcrypt etc....

```
pip install bcrypt
pip install django[argon2]
```

More secured algorithm is argon2 followed by bcrypt and then pbkdf2_sha256.
In settings.py we have to configure password hashers as...........

```
PASSWORD_HASHERS = [
'django.contrib.auth.hashers.Argon2PasswordHasher',
'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
'django.contrib.auth.hashers.BCryptPasswordHasher',
'django.contrib.auth.hashers.PBKDF2PasswordHasher',
'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
```

]

CHAPTER-10

Class Based Views and CRUD operations by using both CBVs and FBVs.

CRUD Operations

FBVs--->Function Based View
CBVs-->Class Based Views
Django ORM

R-->Retrieve/Read(select)
U-->Update(update)
D-->Delete(delete)

CRUD/CURD

CRUD Operations on FBV's

```
D:\Django_20MAR_7PM>django-admin startproject fbvcrudproject
D:\Django_20MAR_7PM>cd fbvcrudproject
D:\Django_20MAR_7PM\fbvcrudproject>py manage.py startapp testapp
```

-->Add app in settings.py

- **models.py**

```
class Employee(models.Model):
   eno = models.IntegerField()
   ename = models.CharField(max_length=64)
   esal = models.FloatField()
   eaddr = models.CharField(max_length=128)
```

-->makemigrations and migrate

- **admin.py**

```
from testapp.models import Employee
class EmployeeAdmin(admin.ModelAdmin):
   list_display = ['eno','ename','esal','eaddr']
admin.site.register(Employee, EmployeeAdmin)
```

- **populate.py**

```
import os
```

```
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'fbvcrudproject.settings')
import django
django.setup()

from testapp.models import Employee
from faker import Faker
from random import *
faker = Faker()
def populate(n):
  for i in range(n):
    feno = randint(1001,9999)
    fename = faker.name()
    fesal = randint(10000,20000)
    feaddr = faker.city()
    emp_record = Employee.objects.get_or_create(
      eno = feno,
      ename = fename,
      esal = fesal,
      eaddr = feaddr)
n = int(input('Enter number of employees:'))
populate(n)
print(f'{n} Records Inserted Successfully....')
```

Apr 26

- **views.py**

```
from testapp.models import Employee
def retrieve_view(request):
  emp_list = Employee.objects.all()
  return render(request,'testapp/index.html',{'emp_list':emp_list})
```

- **base.html**

```
<body>
  <div class="container" align='center'>
   {% block body_block %}
   {% endblock %}
  </div>
</body>
```

- **index.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block %}
```

```
  <h1>Welcome To Employee List</h1>
  <table border="3">
    <thead>
      <th>Employee Number</th>
      <th>Employee Name</th>
      <th>Employee Salary</th>
      <th>Employee Address</th>
      <th>Actions</th>
    </thead>
    {% for emp in emp_list %}
    <tr>
      <td>{{emp.eno}}</td>
      <td>{{emp.ename}}</td>
      <td>{{emp.esal}}</td>
      <td>{{emp.eaddr}}</td>
      <td><a href="#">Update</a> &nbsp&nbsp&nbsp
      <a href="#">Delete</a></td>
    </tr>
    {% endfor %}
  </table>
  <br>
  <a class="btn btn-success" href="#">Insert New Employee</a>
{% endblock %}
```

- **urls.py**

```
path('',views.retrieve_view)
```

- **forms.py**

```
from django import forms
from testapp.models import Employee
class EmployeeForm(forms.ModelForm):
    class Meta:
        model = Employee
        fields = '__all__'
```

- **views.py**

```
from testapp.forms import EmployeeForm
def insert_view(request):
    form = EmployeeForm()
    return render(request,'testapp/insert.html',{'form':form})
```

- **insert.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block %}
 <h1>Employee Insert Form</h1>
 <form method="post">
   {{form.as_p}}
   {% csrf_token %}
   <input type="submit" class="btn btn-success btn-lg" name="" value="Insert
Record">
 </form>
{% endblock %}
```

- **urls.py**

```
path('insert/',views.insert_view)
```

- **index.html**

```
<a href="/insert">Insert New Employee</a>
```

- **views.py**

```
from django.shortcuts import redirect
def insert_view(request):
   form = EmployeeForm()
   if request.method == 'POST':
      form = EmployeeForm(request.POST)
      if form.is_valid():
         form.save()
      return redirect('/')
   return render(request,'testapp/insert.html',{'form':form})
```

DELETE:

```
def delete_view(request,id):
   employee = Employee.objects.get(id = id)
   employee.delete()
   return redirect('/')
```

step-1:

```
<a href="/delete/{{emp.id}}">Delete</a></td>
```

step:2:

```
path('delete/<int:id>', views.delete_view),
```

UPDATE:

step-1:

```
<a href="/update/{{emp.id}}">Update</a></td>
```

step:2:

```
path('update/<int:id>', views.update_view),
```

- **views.py**

```
def update_view(request,id):
    employee = Employee.objects.get(id = id)
    form = EmployeeForm(instance=employee)
    return render(request,'testapp/update.html',{'form':form})
```

- **update.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block %}
<h1>Employee Update Form</h1>
  <form method="post">
    {{form.as_p}}
    {% csrf_token %}
    <input type="submit" class="btn btn-success btn-lg" name="" value="Update
Record">
  </form>
{% endblock %}
```

If employee want to update record

```
def update_view(request,id):
    employee = Employee.objects.get(id = id)
    form = EmployeeForm(instance=employee)
    if request.method == 'POST':
        form = EmployeeForm(request.POST,instance=employee)
        if form.is_valid():
            form.save()
        return redirect('/')
    return render(request,'testapp/update.html',{'form':form})
```

Apr 27

Types of Views:

1.FBV's
2.CBV's

Class Based View(CBV):

-->FBVs are old where as CBVs are new.
-->CBVs are very very easy to use when compared with FBVs. The most commonly used type of views in realtime is CBVs.
-->FBVs are more powerful when compared with CBVs. If we are unable to handle with CBVs then only we have to go for FBVs.

CBVs meant for common requirement.

Ex:

Read data from Employee table--->CBVs
Complex operations over Employee and Customer tables simultaneously--->FBVs

bootstrap(CBV)
css(FBV)

Ex:

```
D:\Django_20MAR_7PM>django-admin startproject cbvproject
D:\Django_20MAR_7PM>cd cbvproject
D:\Django_20MAR_7PM\cbvproject>py manage.py startapp testapp
```

-->Add app in settings.py

- **views.py**

```
from django.views.generic import View
from django.http import HttpResponse
class HelloWorldView(View):
   def get(self,request):
      return HttpResponse('<h1>This response is from class based view</h1>')
```

- **urls.py**

```
path('hello/',views.HelloWorldView.as_view())
```

Note:

1.While defining class based view we have to extend View class.

2.To provide response to GET request Djnago will always call get() method. Hence we have to override this mrthod in our view class. Similarly other http methods like post(), put(), delete().......
3.While defining url pattern we have to use as_view() method.

Template based application by using CBV:

```
from django.views.generic import TemplateView
class TemplateCBV(TemplateView):
   template_name = 'testapp/results.html'
```

- **results.html**

```
<body>
   <h1>Hello this response from template based CBV</h1>
</body>
```

- **urls.py**

```
path('tt/',views.TemplateCBV.as_view()),
```

How to send context parameter:

- **views.py**

```
class TemplateCBV2(TemplateView):
   template_name = 'testapp/results2.html'
   def get_context_data(self,**kwargs):
      context = super().get_context_data(**kwargs)
      context['name'] = 'Radhika'
      context['marks'] = 98
      context['subject'] = 'Python'
      return context
```

- **results2.html**

```
<body>
   <h1>Student Information</h1>
   <h2>Student Name:{{name}}</h2>
   <h2>Student Marks:{{marks}}</h2>
   <h2>Student Subject:{{subject}}</h2>
</body>
```

- **urls.py**

```
path('tt2/',views.TemplateCBV2.as_view())
```

Model Related View classes to perform CRUD operations
View
TemplateView

To perform CRUD operations, predefined View classes are:

ListView	-->To select all records(R)
DetailView	-->To get details of a particular record(R)
CreateView	-->To insert a record(C)
DeleteView	-->To delete a record(D)
UpdateView	-->To update record(U)

1).ListView:

We can use ListView class to list out all records from the database(Model).

It is alternative way to:ModelClassname.objects.all()

Default template file name:modelname_list.html
Default context object name:modelname_list

Ex:ListView class by using CBV's:

```
D:\Django_20MAR_7PM>django-admin startproject cbvproject2
D:\Django_20MAR_7PM>cd cbvproject2
D:\Django_20MAR_7PM\cbvproject2>py manage.py startapp testapp
```

-->Add app in settings.py

- **models.py**

```
class Book(models.Model):
   title = models.CharField(max_length=30)
   author = models.CharField(max_length=30)
   pages = models.IntegerField()
   price = models.FloatField()
```

-->makemigrations and migrate

- **admin.py**

```
from testapp.models import Book
class BookAdmin(admin.ModelAdmin):
   list_display = ['title','author','pages','price']
admin.site.register(Book,BookAdmin)
```

- **views.py**

```
from django.views.generic import ListView
from testapp.models import Book
class BookListView(ListView):
   model = Book
          #default template file: book_list.html
          #default context object name: book_list
```

- **urls.py**

```
path('list/', views.BookListView.as_view())
```

- **book_list.html**

```
<body>
   <div class="container">
     <h1>All Books Information</h1>
     {% for book in book_list %}
     <ul>
       <li>Title:<strong>{{book.title}}</strong></li>
       <li>Author:<strong>{{book.author}}</strong></li>
       <li>Pages:<strong>{{book.pages}}</strong></li>
       <li>Price:<strong>{{book.price}}</strong></li>
     </ul>
     <hr>
     {% endfor %}
   </div>
</body>
```

Apr 29

<u>How to configure our own template file:</u>

By using template_name variable we have to specify our own template file.

<u>How to configure our own context object:</u>

We have to use: context_object_name variable.

- **views.py**

```
class BookListView(ListView):
   model = Book
   template_name = 'testapp/books.html'
   context_object_name = 'books'
```

- **books.html**

```
<body>
  <div class="container">
    <h1>All Books Information from customized template file</h1>
    {% for book in books %}
    <ul>
      <li>Title:<strong>{{book.title}}</strong></li>
      <li>Author:<strong>{{book.author}}</strong></li>
      <li>Pages:<strong>{{book.pages}}</strong></li>
      <li>Price:<strong>{{book.price}}</strong></li>
    </ul>
    <hr>
    {% endfor %}
  </div>
</body>
```

<u>DetailView:</u>

To get the details of a particular record.
Default template_file = book_detail.html
Default context_object_name = book or object

- **views.py**

```
class BookListView2(ListView):
  model = Book
  template_name = 'testapp/books.html'
  context_object_name = 'books'
```

- **books.html**

```
<body>
  <div class="container">
    <h1>All Books Information</h1>
    {% for book in books %}
    <li><a href="/{{book.id}}">{{book.title}}</a></li>
    {% endfor %}
  </div>
</body>
```

- **urls.py**

```
path('list2/', views.BookListView2.as_view()),
```

- **views.py**

```
class BookDetailView(DetailView):
   model = Book
```

- **urls.py**

```
path('<int:pk>/', views.BookDetailView.as_view()),
```

- **book_detail.html**

```
<body>
   <div class="container">
     <h1>{{book.title}} Information</h1>
     <ul>
       <li>Title:<strong>{{book.title}}</strong></li>
       <li>Author:<strong>{{book.author}}</strong></li>
       <li>Pages:<strong>{{book.pages}}</strong></li>
       <li>Price:<strong>{{book.price}}</strong></li>
     </ul>
   </div>
</body>
```

CreateView:
To insert data into our database table(model).

- **views.py**

```
class BookCreateView(CreateView):
   model = Book
```

- **urls.py**

```
path('/', views.BookCreateView.as_view()),
```

If we send request:

    http://127.0.0.1:8000/create/

```
ImproperlyConfigured at /create/
Using ModelFormMixin (base class of BookCreateView) without the 'fields'
attribute is prohibited.
```

- **views.py**

```
fields = ('title','author','pages','price')
```

Now send the request:
```
http://127.0.0.1:8000/create/
```

```
TemplateDoesNotExist at /create/
testapp/book_form.html
```

Note:

The default template is display form for create operation is:book_form.html

We have to create this template file

- **book_form.html**

```
<body>
  <div class="container" align='center'>
    <h1>Book Insert/Create Form</h1>
    <form method="post">
      {{form.as_p}}
      {% csrf_token %}
      <input type="submit" name="" class="btn btn-success" value="Insert New Book">
    </form>
  </div>
</body>
```

If we fill the form and submit

-->The record will be inserted into database, but we will get an error.
-->After inserting to which page, control has to go, we did not define anywhere.
This is the reason for error.

```
ImproperlyConfigured at /create/
No URL to redirect to.  Either provide a url or define a get_absolute_url method on the Model.
```

- **models.py**

```
from django.urls import reverse
class Book(models.Model):
	----
	def get_absolute_url(self):
      return reverse('detail',kwargs={'pk':self.pk})
```

- **urls.py**

```
path('<int:pk>/', views.BookDetailView.as_view(),name='detail'),
```

<u>UpdateView:</u>
To update an existing record.

- **views.py**

```
class BookUpdateView(UpdateView):
   model = Book
   fields = ('pages','price')
```

- **urls.py**

```
path('update/<int:pk>', views.BookUpdateView.as_view()),
```

The default template is: book_form.html

<u>Add update button in book_details.html page</u>

```
<a class="btn btn-warning" href="/update/{{book.id}}">Update This Book
Information</a>
```

<u>DeleteView:</u>

- **views.py**

```
from django.urls import reverse_lazy
class BookDeleteView(DeleteView):
   model = Book
   success_url = reverse_lazy('listbooks')
```

success_url represents the target page which should be displayed after delete operation.
reverse_lazy() function will wait until deleteing the record.

- **urls.py**

```
path('delete/<int:pk>', views.BookDeleteView.as_view()),
```

- **book_confirm_delete.html**

```
<body>
   <div class="container" align='center'>
     <h1>Do you want to really delete book:{{book.title}}???</h1>
```

```
    <form method="post">
      {{form.as_p}}
      {% csrf_token %}
      <input type="submit" name="" class="btn btn-danger" value="Delete
Book">
      <a class="btn btn-success" href="/list2">Cancel(Don't Delete)</a>
    </form>
  </div>
</body>
```

```
success_url = reverse_lazy('listbooks')
path('list2/', views.BookListView2.as_view(),name='listbooks')
```

Add delete button in book_detail.html

```
<a class="btn btn-danger" href="/delete/{{book.id}}">Delete This Book
Information</a>
```

Add Insert button in books.html

```
<a class="btn btn-primary" href="/create">Insert New Book</a>
```

Apr 30

Project for CRUD operations by using CBV's:

ListView	-->To list out all records information
DetailView	-->To get information about a particular object
CreateView	-->To insert/create a record into the database
UpdateView	-->To update a an existing record
DeleteView	-->To delete a particular record

Ex:

```
D:\Django_20MAR_7PM>django-admin startproject cbvproject3
D:\Django_20MAR_7PM>cd cbvproject3
D:\Django_20MAR_7PM\cbvproject3>py manage.py startapp testapp
```

-->Add app in settings.py

- **models.py**

```
class Company(models.Model):
   name = models.CharField(max_length=128)
   location = models.CharField(max_length=64)
   ceo = models.CharField(max_length=64)
```

-->Makemigrations and migrate

- **admin.py**

```
from testapp.models import Company
class CompanyAdmin(admin.ModelAdmin):
   list_display = ['name','location','ceo']
admin.site.register(Company,CompanyAdmin)
```

- **views.py**

```
from django.views.generic import ListView
from testapp.models import Company
class CompanyListView(ListView):
   model = Company
         #default template file name:company_list.html
         #default context object name:company_list
```

- **base.html**

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.2/dist/css/bootstrap.min.cs
s" integrity="sha384-
xOolHFLEh07PJGoPkLv1IbcEPTNtaed2xpHsD9ESMhqIYd0nLMwNLD69Npy4HI+
N" crossorigin="anonymous">
  </head>
  <body>
    <div class="container">
      {% block body_block %}
      {% endblock %}
    </div>
  </body>
</html>
```

- **company_list.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block %}
<h1>List of all companies</h1>
<ul>
  {% for company in company_list %}
  <li>{{company.name}}</li>
  <li>{{company.location}}</li>
  <li>{{company.ceo}}</li>
  {% endfor %}
</ul>
{% endblock %}
```

- **urls.py**

```
path('', views.CompanyListView.as_view()),
```

- **views.py**

```
class CompanyDetailView(DetailView):
    model = Company
            #default template file name:company_detail.html
            #default context object name:company or object
```

- **urls.py**

```
path('<int:pk>/', views.CompanyDetailView.as_view()),
```

- **company_list.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block %}
<h1>List of all companies</h1>
<ul>
  {% for company in company_list %}
  <li><a href="/{{company.id}}">{{company.name}}</a></li>
  {% endfor %}
</ul>
{% endblock %}
```

- **company_detail.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block %}
<h1>{{company.name}} Information</h1>
<ul>
  <h2><li>Company Name:{{company.name}}</li></h2>
  <h2><li>Company Location:{{company.location}}</li></h2>
  <h2><li>Company CEO:{{company.ceo}}</li></h2>
</ul>
{% endblock %}
```

- **views.py**

```
class CompanyCreateView(CreateView):
    model = Company
    fields = '__all__'
```

- **company_form.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block %}
<h1>Company Insert/Update Form </h1>
<form method="post">
  {{form.as_p}}
  {% csrf_token %}
  <input type="submit" name="" value="Insert / Update Record">
</form>
{% endblock %}
```

- **urls.py**

```
path('create/', views.CompanyCreateView.as_view()),
```

```
ImproperlyConfigured at /create/
No URL to redirect to.  Either provide a url or define a get_absolute_url
method on the Model.
```

- **models.py**

```
from django.urls import reverse
# Create your models here.
class Company(models.Model):
    name = models.CharField(max_length=128)
    location = models.CharField(max_length=64)
    ceo = models.CharField(max_length=64)
    def get_absolute_url(self):
        return reverse('detail',kwargs={'pk':self.pk})
```

- **views.py**

```
class CompanyUpdateView(UpdateView):
    model = Company
    fields = ('location','ceo')
```

- **urls.py**

```
path('update/<int:pk>/', views.CompanyUpdateView.as_view()),
```

- **add update link in company_detail.html**

```
<a class="btn btn-warning" href="/update/{{company.id}}">Update
Record</a>
<a class="btn btn-success" href="/">Company List</a>
```

- **views.py**

```
from django.urls import reverse_lazy
class CompanyDeleteView(DeleteView):
    model = Company
    success_url = reverse_lazy('list')
```

- **urls.py**

```
path('delete/<int:pk>/', views.CompanyDeleteView.as_view()),
```

```
TemplateDoesNotExist at /delete/5/
testapp/company_confirm_delete.html
```

- **company_confirm_delete.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block %}
<h1>Do you want to really delete company:{{company.name}}???</h1>
<form method="post">
  {% csrf_token %}
  <input type="submit" class="btn btn-danger" name="" value="Delete
Record">
  <a class="btn btn-success" href="/{{company.id}}">Cancel</a>
</form>
{% endblock %}
```

- **company_detail.html**

```
<a class="btn btn-warning" href="/update/{{company.id}}">Update
Record</a>
<a class="btn btn-danger" href="/delete/{{company.id}}">Delete Record</a>
<a class="btn btn-success" href="/">Company List</a>
```

- **urls.py**

```
urlpatterns = [
   path('admin/', admin.site.urls),
   path('', views.CompanyListView.as_view(),name='list'),
   path('<int:pk>/', views.CompanyDetailView.as_view(),name='detail'),
   path('create/', views.CompanyCreateView.as_view()),
   path('update/<int:pk>/', views.CompanyUpdateView.as_view()),
   path('delete/<int:pk>/', views.CompanyDeleteView.as_view()),
```

May 01

CHAPTER-11

Django ORM

ORM--->Object Relational Mapping
Java:Hibernate, SPring ORM etc......

To select all employees from the employee table
sql query:select * from employee
ORM:Employee.objects.all()

Ex:

```
D:\Django_20MAR_7PM>django-admin startproject ormproject1
D:\Django_20MAR_7PM>cd ormproject1
D:\Django_20MAR_7PM\ormproject1>py manage.py startapp testapp
```

-->Add app in settings.py

- **models.py**

```
class Employee(models.Model):
   eno = models.IntegerField()
   ename = models.CharField(max_length=30)
   esal = models.FloatField()
   eaddr = models.CharField(max_length=64)
```

-->makemigrations and migrate

- **admin.py**

```
from testapp.models import Employee
class EmployeeAdmin(admin.ModelAdmin):
   list_display = ['eno','ename','esal','eaddr']
admin.site.register(Employee,EmployeeAdmin)
```

-->create super user.

- **populate.py**

```
import os
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'ormproject1.settings')
```

```
import django
django.setup()

from testapp.models import Employee
from faker import Faker
from random import *
faker = Faker()
def populate(n):
    for i in range(n):
        feno = randint(1001,9999)
        fename = faker.name()
        fesal = randint(10000,20000)
        feaddr = faker.city()
        emp_record = Employee.objects.get_or_create(
            eno = feno,
            ename = fename,
            esal = fesal,
            eaddr = feaddr)
n = int(input('Enter number of employees:'))
populate(n)
print(f'{n} Records Inserted Successfully....')
```

- **base.html**

```
<body>
    <div class="container">
      {% block body_block %}
      {% endblock %}
    </div>
</body>
```

- **index.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block %}
<h1>Employee Information DashBoard</h1>
<table border="3">
  <thead>
    <th>Employee Number</th>
    <th>Employee Name</th>
    <th>Employee Salary</th>
    <th>Employee Address</th>
  </thead>
  {% for emp in emp_list %}
  <tr>
    <td>{{emp.eno}}</td>
    <td>{{emp.ename}}</td>
    <td>{{emp.esal}}</td>
    <td>{{emp.eaddr}}</td>
  </tr>
  {% endfor %}
</table>
{% endblock %}
```

- **views.py**

```
from testapp.models import Employee
def retrieve_view(request):
    emp_list = Employee.objects.all()
    return render(request,'testapp/index.html',{'emp_list':emp_list})
```

- **urls.py**

```
path('', views.retrieve_view),
```

To select all records:

```
        Employee.objects.all()
        The return type of all() method is:QuerySet
        <class 'django.db.models.query.QuerySet'>
```

To get a particular record:

We have to use get() method.

```
D:\Django_20MAR_7PM\ormproject1>py manage.py shell

>>> from testapp.models import Employee
>>> emp = Employee.objects.get(id=1)
>>> emp #<Employee: Employee object (1)>
>>> emp.eno #5568
>>> emp.ename #'Lauren Griffin'
>>> type(emp) #<class 'testapp.models.Employee'>
```

-->The return type of get() method is Employee object.

How to find query associated with QuerySet:
Every ORM statement will be converted into sql query. We can find query from the QuerySet.

```
>>qs = Employee.objects.all()
>>> qs.query
<django.db.models.sql.query.Query object at 0x0000020FE9274D00>
>>> str(qs.query)
'SELECT "testapp_employee"."id", "testapp_employee"."eno",
"testapp_employee"."ename", "testapp_employee"."esal",
"testapp_employee"."eaddr" FROM "testapp_employee"'
```

How to filter records based on some condition
1).List out all employees whose salaries greater than 15000.

```
                    emp_list= Employee.objects.filter(esal__gt=15000)
```

2).Salaries greater than or equal to 15000

```
                    emp_list= Employee.objects.filter(esal__gte=15000)
```

Similarly we can use __lt and __lte also.

Ex:
1).exact:exact match

```
>>> emp = Employee.objects.get(id__exact=52)
>>> emp.ename #'Radhika'
>>> emp = Employee.objects.get(id=51)
>>> emp.ename #'Sunny'
```

2).contains:case sensitive containment test

```
                    select .....where ename like '%jhon%'
          emp_list = Employee.objects.filter(ename__contains='jhon')
```

3).in:
In a given iterable like tuple or list

```
          emp_list = Employee.objects.filter(id__in=[1,51,52])
```

4).gt:greater than
5).gte:greater than or equal to
6).lt:less than

7).lte:less than or equal to

8).startswith:
    select all employees where ename starts with 'S'
    emp_list = Employee.objects.filter(ename__startswith='S')

9).endswith:
    emp_list = Employee.objects.filter(ename__endswith='s')

10).range:
    range test(inclusive)
    To select all employees where esal in the range 12000 to 15000
    emp_list = Employee.objects.filter(esal__range=[12000,15000])

Q1.Select all employees where ename starts with 'A'
    emp_list = Employee.objects.filter(ename__startswith='A')

Q2.Select all employees whose sal <= 15000
    emp_list= Employee.objects.filter(esal__lte=15000)

Q3.Select all employees where ename starts with 'A' or esla <= 15000.
We can implement OR queries in 2-ways.

<u>1st way:</u>
emp_list = queryset1 | queryset2
Ex:
emp_list = Employee.objects.filter(ename__startswith='A') |

            Employee.objects.filter(esal__lte=11000)
<u>2nd way:</u>
filter(Q(condition1) | Q(condition2))
from django.db.models import Q
emp_list = Employee.objects.filter(Q(condition1) | Q(condition2))
emp_list = Employee.objects.filter(Q(ename__startswith='D') |
Q(esal__lte=12000))

May 02

<u>How to implement AND queries:</u>
AND:all conditions should be satisfied.

3-ways:
1).queryset1 & queryset2
2).filter(Q(condition1) & Q(condition2))
3).filter(condition1,condition2)

Ex:
select all employees where ename starts with 'S' and esal < 15000.
1).emp_list = Employee.objects.filter(ename__startswith='S') & Employee.objects.filter(esal__lt=15000)

2).emp_list = Employee.objects.filter(Q(ename__startswith='A') & Q(esal__lt=18000))

3).emp_list = Employee.objects.filter(ename__startswith='S',esal__lt=18000)

How to implement Not queries in Django ORM:
all() -->To get all records.
filter(condition)-->To get records where condition is satisfied.

We can implement NOT queries in 2-ways:
1st way: exclude(condition)--->To get records where condition is failed.
2nd way: filter(~Q(condition))
Ex: To select all employees whose name not starts with 'S'
```
emp_list = Employee.objects.exclude(ename__startswith='S')
emp_list = Employee.objects.filter(~Q(ename__startswith='D'))
```

How to select only required columns in the query set:
select * from employee;
select ename,esal from employee;
3-ways
1).By using values_list():

- **views.py**

```
emp_list = Employee.objects.all().values_list('ename','esal')
return render(request,'testapp/specificcolumns.html', {'emp_list':emp_list})
```

- **specificcolumns.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block %}
```

```
<h1>Employee Information DashBoard</h1>
<table border="3">
  <thead>
    <th>Employee Name</th>
    <th>Employee Salary</th>
  </thead>
  {% for emp in emp_list %}
  <tr>
    <td>{{emp}}</td>
    <td>{{emp}}</td>
  </tr>
  {% endfor %}
</table>
<br>
{% endblock %}
```

- **changes**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block %}
<h1>Employee Information DashBoard</h1>
<table border="3">
  <thead>
    <th>Employee Name</th>
    <th>Employee Salary</th>
  </thead>
  {% for emp in emp_list %}
  <tr>
    {% for v in emp %}
    <td>{{v}}</td>
    {% endfor %}
  </tr>
  {% endfor %}
</table>
<br>
{% endblock %}
```

2).By using values():

```
        emp_list = Employee.objects.all().values('ename','esal')
```

- **html file**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block %}
<h1>Employee Information DashBoard</h1>
<table border="3">
 <thead>
  <th>Employee Name</th>
  <th>Employee Salary</th>
 </thead>
 {% for emp in emp_list %}
 <tr>
  {% for k,v in emp.items %}
  <td>{{v}}</td>
  {% endfor %}
 </tr>
 {% endfor %}
</table>
<br>
{% endblock %}
```

3).By using only():

```
        emp_list = Employee.objects.all().only('ename','esal')
```

- **html file**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block %}
<h1>Employee Information DashBoard</h1>
<table border="3">
 <thead>
  <th>Employee Name</th>
  <th>Employee Salary</th>
 </thead>
 {% for emp in emp_list %}
 <tr>
  <td>{{emp.ename}}</td>
  <td>{{emp.esal}}</td>
 </tr>
 {% endfor %}
```

```
</table>
<br>
{% endblock %}
```

Note:

values_list()--->QuerySet contains tuple.
values()--->QuerySet contains dict objects
only()--->QuerySet contains Employee objects

-->Hence values() method is recommended to use when compared with others.

Aggregate Functions:

Django ORM defines several functions to perform aggregate operations.
Avg(), Max(),Min(),Sum(),Count()...etc.......

- **views.py**

```
from django.db.models import Avg,Max,Min,Sum,Count
def aggregate_view(request):
   avg = Employee.objects.all().aggregate(Avg('esal'))
   max = Employee.objects.all().aggregate(Max('esal'))
   min = Employee.objects.all().aggregate(Min('esal'))
   sum = Employee.objects.all().aggregate(Sum('esal'))
   count = Employee.objects.all().aggregate(Count('esal'))
   my_dict = {'avg':avg['esal__avg'], 'max':max['esal__max'],
'min':min['esal__min'],'sum':sum['esal__sum'], 'count':count['esal__count']}
   return render(request,'testapp/aggregate.html',my_dict)
```

- **aggregate.html**

```
<!DOCTYPE html>
{% extends 'testapp/base.html' %}
{% block body_block %}
  <h1>Employee Aggregate Information </h1>
  <ul>
    <h2><li>Average Salary:{{avg}}</li></h2>
    <h2><li>Maximum Salary:{{max}}</li></h2>
    <h2><li>Minimum Salary:{{min}}</li></h2>
    <h2><li>Total Salary:{{sum}}</li></h2>
    <h2><li>Number of Employees:{{count}}</li></h2>
  </ul>
{% endblock %}
```

- **urls.py**

```
path('agg/', views.aggregate_view),
```

May 03

How to create, update and delete records?

1st way: py manage.py shell

```
D:\Django_20MAR_7PM\ormproject1>py manage.py shell
>>> from testapp.models import Employee
>>> e = Employee(eno=1234,ename='Mahesh',esal=1234.0,eaddr='Vja')
>>> e.save() #This employee will be inserted into database
```

2nd way:

```
>>>
Employee.objects.create(eno=2345,ename='Kareena',esal=123.0,eaddr='Chen
nai')
```

How to add multiple records at a time:

By using method bulk_create

```
Employee.objects.bulk_create(
[Employee(eno=3333,ename='Sachin',esal=33333.0,eaddr='Mumbai'),
Employee(eno=6666,ename='Kohli',esal=66666.0,eaddr='Delhi'),
)
```

How to delete single record

```
>>> e = Employee.objects.get(eno=8888)
>>> e.delete()
(1, {'testapp.Employee': 1})
```

How to delete multiple records:

```
>>> qs = Employee.objects.filter(esal__gte=15000)
>>> qs.count() #26
>>> qs.delete()
(26, {'testapp.Employee': 26})
>>> qs.count() #0
```

How to delete all records:

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

<u>How to update record:</u>

```
>>> e = Employee.objects.get(eno=6775)
>>> e.ename
>>> e.esal
>>> e.esal=23000
>>> e.save()
>>> e.ename='sunny'
>>> e.save()
```

<u>How to order queries in sorting order</u>

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

1).To display all employees according to ascending order eno.

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

2).To sort all employees according to descending order eno.

```
          emp_list = Employee.objects.all().order_by('-eno')
```

3).How to get highest salaried employee object?

Arrange all employees in descending order and select first employee.

```
          >>> e = Employee.objects.all().order_by('-esal')[0]
          >>> e.ename
          >>> e.esal
```

4).To get all employees based on alphabatical order of names.

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

5).To ignore case?

```
from django.db.models.functions import Lower
emp_list = Employee.objects.all().order_by(Lower('ename'))
```

<u>How to perform union operations for query set:</u>

By using union operation, we can combine results of 2 or more queries from same model or from different models.

```
q1 = Employee.objects.filter(esal__lte=12000)
q2 = Employee.objects.filter(ename__startswith='S')
q3 = q1.union(q2)
emp_list = q3
```

CHAPTER-13

Working with Django Middleware

-->At pre processing of request or at post processing of request, if we want to perform any activity automatically then we should go for middleware.

http://127.0.0.1:8000
http://127.0.0.1:8000/

http://127.0.0.1:8000/agg
http://127.0.0.1:8000/agg/

submit the form--->csrf verification
AuthenticationMiddleware

http====>https===>SecurityMiddleware

-->Middleware is applicable for every incoming request and outgoing response.

<u>Middleware Structure:</u>
Based on our requirement, we can configure our own middleware also.
Every customized middleware is a python class and it is the child class of object.
class A(object):
class A:

This python class should contains 2-mandatory methods.
1).def __init__(self,get_response):
-->get_response is a function which can be used to send request to the next level and to get required response.
-->This method will be executed only once at the time of creating middleware class object, which is mostly happended at the time of server starting.

2).def __call__(self,request):

```
	This method will be executed for every request separately
	#code for preprocessing of request
	response = self.get_response(request) #Trigger request to the next
level
```

```
            #code for post processing of request.
            return response
```
Middleware classes we have to define middleware.py file(inside testapp)

May 04

Middleware:

```
D:\Django_20MAR_7PM>django-admin startproject middlewareproject1
D:\Django_20MAR_7PM>cd middlewareproject1
D:\Django_20MAR_7PM\middlewareproject1>py manage.py startapp testapp
```
-->Add app in settings.py

- **views.py**

```
from django.http import HttpResponse
def welcome_view(request):
   return HttpResponse('<h1>Custome Middleware Demo</h1>')
```

- **urls.py**

```
path('hello/', views.welcome_view)
```

Inside testapp folder

- **middleware.py**

```
class ExecutionFlowMiddleware(object):
   def __init__(self,get_response):
      print('init method execution.....')
      self.get_response = get_response
   def __call__(self,request):
      print('Pre processing of request')
      response = self.get_response(request)
      print('Post processing of request')
      return response
```

- **settings.py**

```
MIDDLEWARE = [
            -----
   'testapp.middleware.ExecutionFlowMiddleware'
]
```

Middleware application to show information saying app is under maintenace

-->create project
-->Create testapp

-->Add app in settings.py

- **views.py**

```
from django.http import HttpResponse
def home_page_view(request):
   return HttpResponse('<h1>Hello this response is from view function
response</h1>')
```

- **urls.py**

```
path('hello/', views.home_page_view)
```

- **middleware.py**

```
from django.http import HttpResponse
class AppMaintenanceMiddleware(object):
   def __init__(self,get_response):
      self.get_response = get_response
   def __call__(self,request):
      return HttpResponse('<h1>Currently application under
maintenance...Please try after 2-days....</h1>')
```

- **settings.py**

```
MIDDLEWARE = [
   'testapp.middleware.AppMaintenanceMiddleware'
]
```

<u>Middleware application to show meaningful response if view function raises any error.</u>

In the middleware we can define process_exception() method, which will be executed if view function raises any error.

process_exception(self,request,exception)

- **views.py**

```
def home_page_view(request):
   print(10/0)
   return HttpResponse('<h1>This is from view function</h1>')
```

- **urls.py**

```
path('hello/', views.home_page_view)
```

- **middleware.py**

```
from django.http import HttpResponse
class ErrorMessageMiddleware(object):
   def __init__(self,get_response):
      self.get_response = get_response

   def __call__(self,request):
      response = self.get_response(request)
      return response

   def process_exception(self,request,exception):
      # return HttpResponse('<h1>Currently we are facing some technical
problem...pls try after some time....</h1>')
      return HttpResponse(f'<h1>Currenty we are facing some technical
problems<br>The Raised Exception:{exception.__class__.__name__}<br> The
Exception Message:{exception}</h1>')
```

- **settings.py**

```
MIDDLEWARE = [
      'testapp.middleware.ErrorMessageMiddleware'
]
```

<u>Configuration of Multiple middleware classes:</u>
We can configure any number of middlewares and all these middlewares will be executed according to order declared inside settings.py

- **views.py**

```
def home_page_view(request):
   print('This line printed by view function')
   return HttpResponse('<h1>This is from view function</h1>')
```

- **urls.py**

```
path('hello/', views.home_page_view)
```

- **middleware.py**

```
from django.http import HttpResponse
class FirstMiddleware(object):
   def __init__(self,get_response):
      self.get_response = get_response
   def __call__(self,request):
      print('This line printed by Middleware-1 before processing request')
      response = self.get_response(request)
      print('This line printed by Middleware-1 after processing request')
      return response

class SecondMiddleware(object):
   def __init__(self,get_response):
      self.get_response = get_response
   def __call__(self,request):
      print('This line printed by Middleware-2 before processing request')
      response = self.get_response(request)
      print('This line printed by Middleware-2 after processing request')
      return response
```

- **settings.py**

```
MIDDLEWARE = [
   'testapp.middleware.SecondMiddleware',
   'testapp.middleware.FirstMiddleware',
]
```

- **test.py**

```
import time
class Test:
        def __init__(self):
                print('Constructor Execution.....')
        def __del__(self):
                print('Destructor Execution.....')
l = [Test(),Test(),Test()]
time.sleep(5)
print('End of application')
```

May 06

CHAPTER-12

Working with Advanced Model concepts

Model Inheritance:

1.Abstract Base Class model inheritance
2.Multi table inheritance
    3.Multi level inheritance
    4.Multiple inheritance
5.Proxy model inheritance

1.Abstract Base Class model inheritance:

-->If several model classes having some common fields, then it is not recommended to write these fields in every model class separately, because it increases length of the code and reduces readability.

-->We have to separate those common fields into a separate model class which is nothing but Base class. If we extend Base class then automatically common fields will be inherited to every child class.

Ex:
1.Create project miproject1
2.Create app
3.Add app in settings.py

- **settings.py**

```
DATABASES = {
   'default': {
      'ENGINE': 'django.db.backends.mysql',
      'NAME': 'midb_7pm',
      'USER': 'root',
      'PASSWORD': 'root'
   }
}
```

- **models.py**

```
class ContactInfo(models.Model):
   name = models.CharField(max_length=30)
   email = models.EmailField()
   address = models.CharField(max_length=30)
   class Meta:
      abstract = True
class Student(ContactInfo):
   rollno = models.IntegerField()
   marks = models.IntegerField()
class Teacher(ContactInfo):
   subject = models.CharField(max_length=30)
   salary = models.FloatField()
```

- **admin.py**

```
from testapp.models import Student,Teacher
admin.site.register(Student)
admin.site.register(Teacher)
```

-->Makemigrations and Migrate.

Note:

ConcatInfo class is a abstract class, hence table wont be created. This type of inheritance is applicable only at code elevel but not at database level.

<u>2).Multi table inheritance:</u>

-->If the Base class is not abstract, then such type of inheritance is called as multi table inheritance.
-->This type of inheritance applicable at code level and DB level.
-->In multi table inheritance, inside database, for both parent and child tables will be created.

- **models.py**

```
class ContactInfo1(models.Model):
   name = models.CharField(max_length=30)
   email = models.EmailField()
   address = models.CharField(max_length=30)
class Student1(ContactInfo1):
   rollno = models.IntegerField()
   marks = models.IntegerField()
class Teacher1(ContactInfo1):
   subject = models.CharField(max_length=30)
   salary = models.FloatField()
```

- **admin.py**

```
admin.site.register(Student1)
admin.site.register(Teacher1)
admin.site.register(ContactInfo1)
```

-->In this case 3-tables will be created and child table will maintain pointer to the parent table to refer the common properties.

3).Multi level inheritance:

Inheritance at multiple levels.

Ex:

```
class Person(models.Model):
   name = models.CharField(max_length=30)
   age = models.IntegerField()
class Employee(Person):
   eno = models.IntegerField()
   esal = models.FloatField()
class Manager(Employee):
   exp = models.IntegerField()
   team_size = models.IntegerField()
```

- **admin.py**

```
admin.site.register(Person)
admin.site.register(Employee)
admin.site.register(Manager)
```

4).Multiple Inheritance:

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

```
class Parent1(models.Model):
   f1 = models.CharField(max_length=30)
   f2 = models.CharField(max_length=30)
class Parent2(models.Model):
   f3 = models.CharField(max_length=30,primary_key=True)
   f4 = models.CharField(max_length=30)
class Child(Parent1,Parent2):
   f5 = models.CharField(max_length=30)
   f6 = models.CharField(max_length=30)
```

Note:

1.Parent classes hould not contain common fields, otherwise we eill get an error.

2.Internally this inheritance also multi table inheritance.

-->Makemigrations and Migrate.

Model Manager:

-->We can use ModelManager to interact with database.
-->We can get default ModelManager by using Model.objects property
-->Model.objects is of type:django.db.models.manager.Manager

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

Q1.What is the purpose of ModelManager?
Ans: To interact with database.

Q2.How to get default Model Manager?
Ans: By using Model.objects property
Q3.ModelManager is of what type?
Ans: django.db.models.manager.Manager

go to shell:

```
>>> from testapp.models import Person
>>> type(Person.objects)
<class 'django.db.models.manager.Manager'>
```

-->Based on our requirement, we can define and use our own custom model managers.

How to define our own custom manager:

How to access private variables outside of the class

```
class Test:
    __x = 100 #static variable
    def __init__(self):
        self.__y = 200 #instance variable
```

```
t = Test()
#print(t.__dict__)
print(t._Test__y)
#print(Test.__dict__)
print(Test._Test__x)
```

May 07

Employee model contains : 1000 records
employees = Employee.objects.all()-->To get all records based on insertion order.
employees = Employee.objects.all()-->To get all records based on ascending order of eno's

<u>How to define our own custom manager:</u>
-->We have to create child class for models.Manager class
-->Whenever we are calling all() method, internally it will call get_queryset() method.
-->To customize behaviour, we have to override this method in our custom manager class.

Ex: To retrieve all employees data according to ascending order of eno, we have to define CustomManager class.

-->Create new project miproject2
-->Create an app
-->Add app in settings.py

- **models.py**

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

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

- **admin.py**

```
from testapp.models import Employee
class EmployeeAdmin(admin.ModelAdmin):
    list_display = ['eno','ename','esal','eaddr']
admin.site.register(Employee,EmployeeAdmin)
```

- **views.py**

```
from testapp.models import Employee
def display_view(request):
    emp_list = Employee.objects.all()
    return render(request,'testapp/index.html',{'emp_list':emp_list})
```

- **index.html**

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.2/dist/css/bootstrap.min.cs
s" integrity="sha384-
xOolHFLEh07PJGoPkLv1IbcEPTNtaed2xpHsD9ESMhqIYd0nLMwNLD69Npy4HI+
N" crossorigin="anonymous">
    <style media="screen">
      body{
        background:red;
        color:white
      }
    </style>
  </head>
  <body>
    <div class="container" align='center'>
      <h1>Welcome To Employee List</h1>
      <table border="3">
        <thead>
          <th>Employee Number</th>
          <th>Employee Name</th>
          <th>Employee Salary</th>
          <th>Employee Address</th>
        </thead>
        {% for emp in emp_list %}
        <tr>
          <td>{{emp.eno}}</td>
          <td>{{emp.ename}}</td>
          <td>{{emp.esal}}</td>
          <td>{{emp.eaddr}}</td>
        </tr>
        {% endfor %}
      </table>
    </div>
  </body>
</html>
```

- **populate.py**

```
import os
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'miproject2.settings')
import django
django.setup()

from testapp.models import Employee
from faker import Faker
from random import *
faker = Faker()
def populate(n):
    for i in range(n):
        feno = randint(1001,9999)
        fename = faker.name()
        fesal = randint(10000,20000)
        feaddr = faker.city()
        emp_record = Employee.objects.get_or_create(
            eno = feno,
            ename = fename,
            esal = fesal,
            eaddr = feaddr)
n = int(input('Enter number of employees:'))
populate(n)
print(f'{n} Records Inserted Successfully....')
```

- **urls.py**

```
path('data/', views.display_view),
```

-->Based on our requirement, we can define our own new methods also inside CustomManager class.

- **models.py**

```
class CustomManager(models.Manager):
   def get_queryset(self):
      qs = super().get_queryset().order_by('eno')
      return qs
   def get_emp_sal_range(self,minsal,maxsal):
      qs = super().get_queryset().filter(esal__range=(minsal,maxsal))
      return qs
   def get_emp_sorted_by(self,param):
      qs = super().get_queryset().order_by(param)
      return qs
```

- **views.py**

```
def display_view(request):
   #emp_list = Employee.objects.all()
   #emp_list = Employee.objects.get_emp_sal_range(18000,20000)
   #emp_list = Employee.objects.get_emp_sorted_by('ename')
   emp_list = Employee.objects.get_emp_sorted_by('-esal')
   return render(request,'testapp/index.html',{'emp_list':emp_list})
```

<u>5).Proxy Model Inheritance:</u>

-->For the same Model, we can provide a customized view without touching the database. This is possible by using proxy model inheritance.

-->In this table, a separate new table wont be created and new proxy model also pointing to the same old table.

```
class Employee:
          fields

class ProxyEmployee(Employee):
          class Meta:
                    proxy = True
```

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

- **models.py**

```
class CustomManager1(models.Manager):
   def get_queryset(self):
      return super().get_queryset().filter(esal__gte=19000)

class CustomManager2(models.Manager):
   def get_queryset(self):
      return super().get_queryset().filter(esal__lte=11000)

class CustomManager3(models.Manager):
   def get_queryset(self):
      return super().get_queryset().order_by('eno')

class Employee(models.Model):
   eno = models.IntegerField()
   ename = models.CharField(max_length=30)
   esal = models.FloatField()
   eaddr = models.CharField(max_length=30)
   objects = CustomManager1()

class ProxyEmployee1(Employee):
   objects = CustomManager2()
   class Meta:
      proxy = True

class ProxyEmployee2(Employee):
   objects = CustomManager3()
   class Meta:
      proxy = True
```

- **admin.py**

```
from testapp.models import Employee,ProxyEmployee1,ProxyEmployee2
class EmployeeAdmin(admin.ModelAdmin):
   list_display = ['eno','ename','esal','eaddr']

class ProxyEmployee1Admin(admin.ModelAdmin):
   list_display = ['eno','ename','esal','eaddr']

class ProxyEmployee2Admin(admin.ModelAdmin):
   list_display = ['eno','ename','esal','eaddr']
```

```
admin.site.register(Employee,EmployeeAdmin)
admin.site.register(ProxyEmployee1,ProxyEmployee1Admin)
admin.site.register(ProxyEmployee2,ProxyEmployee2Admin)
```

- **views.py**

```
def display_view(request):
   #emp_list = Employee.objects.all()
   #emp_list = ProxyEmployee1.objects.all()
   emp_list = ProxyEmployee2.objects.all()
   return render(request,'testapp/index.html',{'emp_list':emp_list})
```

CHAPTER-14

Deployment of our application to the live

Q.Diff b/w str() and repr()?

```
class Student:
	def __init__(self,name,rollno):
		self.name = name
		self.rollno = rollno
	def __str__(self):
		return 'This is Student with Name:{} and 
Rollno:{}'.format(self.name,self.rollno)
s1 = Student('Radhika',101)
s2 = Student('Lilly',102)
print(s1)
print(s2)
```

Ex:

```
import datetime
today = datetime.datetime.now()
print(type(today))
s = repr(today)#converting datetime object to str
print(type(s))
d = eval(s)#converting str object to datetime
print(type(d))
```

May 08

Deployment of our application to the live environment

Deployment:- The process of moving an application to the live/production

There are several deployment options are available for django web application. These options will be based on.

1.Scalability
2.Performance
3.Price
4.Security
5.Easy to use
etc........

Various deployment options:

1.pythonanywhere.com
2.Digital Ocean
3.Heroku
4.Amazon Cloud

1.Version Control System
2.Remote Hosting Server
3.Deployment Platform

Need of Version Control System:

-->To maintain multiple versions of the same project.
-->At any point of time, we can have backup of previous version.
-->We can see the differences between 2 or more versions in our code base.
-->We can run multiple versions of the same project simultaniously.

Version constrol software tools:

Git, CVS, SVN, Mercurial...........

Note: Git is a version control system, which maintain and track different versions.

Git vs GitHub:

-->Git is a version control system that helps to track changes in our code.
-->Github is a remote hosting platform to host our files remotely.
-->Github is to host our remote code repository.

Remote hosting platforms:GitHub, bitbucket, gitlab......

Note:

If our application at remote hosting platform then deployment on various platforms will become very easy.

Deployment platform:

Where we have to deploy our application.

Ex:

pythonanywhere.com
heroku
amazon cloud

Version control system vs Remote hosting platform vs Deployment platform:
How to install GIT:

https://git-scm.com/downloads

Activities related to Git repository:
1). Create a directory named with my_cwd. It is our current working directory.
2). We required a local repository to track our files. For this we have to use the command:

git init:

This command will create a folder named with .git, which acts as local repository.

3). Bydefault git wont track any files. We have to addf iles to the staging area, such files only can be tracked by git.

We can add files to the staging area by using:

```
git add abc.txt
git add .===>. means all files present in current working directory.
```

4). We have to commit the changes in the files present in staging area. Then the files/changes will be moved to local repository.

```
git commit -m 'commit message'
```

5). We can find status of files by using

```
git status
```

6). We can find all commits related to our local repo?

```
git log
```

git init-->To create and initalize local repository
git add files-->To add files to staging area
git commit -m 'commit message'-->To commit changes so that changes will be stored in local repo
git status-->To find status
git log-->To see all commits.

github:
Create a new repository 'djangorepo'

push an existing repository from the command line

```
git remote add origin https://github.com/maheshdj123/djangorepo.git
git branch -M main
git push -u origin main
```

Deployment on pythonanywhere.com
Virtual Environment:
It is an isolated environment where we can install our project specific required softwares. We can develop and run our application in this environment.

Creation of virtual environment:
Consoles-->Bash

```
$ mkvirtualenv --python=python3.10 myproj
```
Here myproj:Name of the virtual environment.

```
(myproj) 14:50 ~ $
```
It means virtual environment created and currently active.

```
$pip list
```

Check django version:
```
C:\Windows\System32>py
>>> import django
>>> django.__version__
'4.1'
```

```
$pip install -U django==4.1
```

Copy our application from github to virtual env(pythonanywhere.com)
github:https://github.com/maheshdj123/djangorepo.git

command:
```
$git clone https://github.com/maheshdj123/djangorepo.git
$ls
$cd djangorepo
$ls
$cd firstproject
$pwd
```

configure on the web tab:
source code path:/home/maheshproject/djangorepo/firstproject

virtual env:/home/maheshproject/.virtualenvs/myproj
.virtualenvs------>. means hidden directory

<u>WSGI configuration</u>
maheshproject_pythonanywhere_com_wsgi.py
Remove all hello world related configuration

```
import os
import sys
path = '/home/maheshproject/djangorepo/firstproject'
if path not in sys.path:
   sys.path.append(path)
os.chdir(path)
os.environ.setdefault('DJANGO_SETTINGS_MODULE','firstproject.settings')
```

We will get an error:
DisallowedHost at/

in settings.py:

```
ALLOWED_HOSTS = ['maheshproject.pythonanywhere.com']
```

Reload app and refresh browser.

May 10

Deployment:
to send our app to live/production.

Steps:
1.create local repo with our project
2.move this to remote repository
3.clone this to pythonanywhere.com
4.do required configuration

1.create a new folder my_cwd.
2.copy sunnyjobs project and paste in my_cwd.
3.go to gitbash:

```
$cd d:
$cd my_cwd
$git init
$git add .
$git status
$git commit -m 'first commit'
$git status
```

-->go to github:

    create a new repository jobsrepo

-->go to gitbash:

```
$ git remote add origin https://github.com/maheshdj123/jobsrepo.git
$ git branch -M main
$ git push -u origin main
```

-->go to pythonanywhere.com

    In dashboard-->New console-->bash console
    create venv

```
$mkvirtualenv --python=python3.10 myproj2

$pip list
$pip install -U django==4.1
$pip list
```

github:

    https://github.com/maheshdj123/jobsrepo.git

go to venv in pythonanywhere.com:

```
$git clone https://github.com/maheshdj123/jobsrepo.git
$cd jobsrepo
$cd sunnyjobs
$pwd
```

source code path:

    /home/maheshproject/jobsrepo/sunnyjobs

virtual env path:

    home/maheshproject/.virtualenvs/myproj2

$python manage.py makemigrations.
-->If any problems came configure with sqlite-3 in settings.py

```
$py manage.py makemigrations
$py manage.py migrate
$py manage.py createsuperuser
```

-->Go to web tab:

Add a new web app-->manual configuration-->select python-3.10
give source path:/home/maheshproject/jobsrepo/sunnyjobs
virtual env:home/maheshproject/.virtualenvs/myproj2

-->Reload application.

- **configure wsgi.py**

```
import os
import sys
path = '/home/maheshproject/jobsrepo/sunnyjobs'
if path not in sys.path:
   sys.path.append(path)
   os.chdir(path)
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'sunnyjobs.settings')
import django
django.setup()
```

-->Reload application then access.

We will get an error:

DisallowedHost at /

- **in settings.py**

```
ALLOWED_HOSTS = ['maheshproject.pythonanywhere.com']
```

Add static files:

URL:/static
Path://home/maheshproject/jobsrepo/sunnyjobs/static

For admin application:

URL:/static/admin
Path:/home/maheshproject/.virtualenvs/myproj2/lib/python3.10/site-packages/django/contrib/admin/static/admin

Go to consoles bash console:

```
$ py populate.py
        -->If faker module is not available.
$pip install faker
$py populate.py
```

-->Reload app:

https://maheshproject.pythonanywhere.com/

--**The End**--