G23

1) Terminal Installation:

File → Settings → Install → In the searchbox just type terminal → platform-ide-terminal

2) Python AutoCompletion:

File → Settings → Install → In the searchbox just type python → autocomplete-python

3) django:

File → Settings → Install → In the searchbox just type django → atom-django

4) How to Change Terminal from Powershell to Normal Command Prompt:

File → Settings → Install → In the searchbox just type terminal → platform-ide-terminal → settings → Shell Override

C:\Windows\System32\cmd.exe

DJango:

- **Django is a free and open-source web framework.**
- **It is written in Python.**
- **It follows the Model-View-Template (MVT) architectural pattern.**
- **It is maintained by the Django Software Foundation (DSF)**

- **It is used by several top websites like Youtube,Google,Dropbox,Yahoo Maps, Mozilla,Instagram,Washington Times,Nasa and many more**

- **https://www.shuup.com/blog/25-of-the-most-popular-python-and-django-websites/**

- **Django was created in 2003 as an internal 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 open source framework on July 21st 2005.**

- **The Django was named in the memory of Guitarist Django Reinhardt.**

- **Official website: djangoproject.com**

Top 5 Features of Django Framework:

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

The following are main important features of Django

1) Fast:

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

2) Fully loaded:

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

3) Security:

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

4) Scalability:

Some of the busiest sites on the planet use Django's 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.

Note:

1) **As Django is specially designed web application framework, the most commonly required activities will takes care automatically by Django and Hence Developer's life will be simplified and we can develop applications very easily.**

2) **As Django invented at news paper,clear documentation is avilable including a sample polling application.**

3) **https://docs.djangoproject.com/en/2.1/contents/**

How to install django:

1. Make sure Python is already installed in our system

python --version

2. Install django by using pip

pip install django
pip install django == 1.11.9

D:\>pip install django
Collecting django
Downloading
https://files.pythonhosted.org/packages/51/1a/6153103322/Django-2.1-py3-none-any.whl (7.3MB) 100% || 7.3MB 47kB/s

Collecting pytz (from django)
Downloading https://files.pythonhosted.org/packages/30/4e/53b898779a/pytz-2018.5-py2.py3-none-any.whl (510kB)
100% || 512kB 596kB/s

Installing collected packages: pytz, django
Successfully installed django-2.1 pytz-2018.5
You are using pip version 9.0.3, however version 18.0 is ava
You should consider upgrading via the 'python -m pip install

3. To check django version:

py -m django --version

Django Project vs Django Application:

A Django project is a collection of applications and configurations which forms a full web application.
Eg: Bank Project

A Dango Application is responsible to perform a particular task in our entire web application.
Eg: loan app
registration app
polling app etc

Diagram

Project = Several Applications + Configuration Information

Note:

1) **The Django applications can be plugged into other projects.ie these are reusable. (Pluggable Django Applications)**
2) **Without existing Django project there is no chance of existing Django Application. Before creating any application first we required to create project.**

How to create Django Project:

Once we installed django in our system, we will get 'django-admin' command line tool, which can be used to create our Django project.

django-admin startproject firstProject

D:\>mkdir djangoprojects

D:\>cd djangoprojects

D:\djangoprojects>django-admin start-project firstProject

The following project structure will be created

```
D:\djangoprojects>
 |
 +---firstProject
    ¦
    ¦---manage.py
    ¦
    +---firstProject
        ¦---settings.py
        ¦---urls.py
        ¦--wsgi.py
        ¦-- __init__.py
```

__init__.py:

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

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

settings.py:

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

urls.py:

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

wsgi.py:

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

manage.py:

- **The most commonly used python script is manage.py**
- **It is a command line utility to interact with Django project in various ways like to run development server, run tests, create migrations etc.**

How to Run Django Development Server:

We have to move to the manage.py file location and we have to execute the following command.

```
py manage.py runserver
```

```
D:\djangoprojects\firstProject>py manage.py runserver

Performing system checks...

System check identified no issues (0 silenced).

You have 13 unapplied migration(s). Your project may not work properly until you apply
the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
August 03, 2018 - 15:38:59
Django version 1.11, using settings 'firstProject.settings'
```

Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.

Now the server started.

How to Send First Request:

Open browser and send request:

http://127.0.0.1:8000/

The following should be response if everything goes fine.

It worked!
Congratulations on your first Django-powered page.
Next, start your first app by running python manage.py startapp [app_label].

You're seeing this message because you have DEBUG = True in your Django settings file and you haven't configured any URLs. Get to work!

Role of Web Server:

- **Web Server provides environment to run our web applications.**
- **Web Server is responsible to receive the request and forward 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. Special Thanks to Django.**

Note: Once we started Server a special database related file will be generated in our project folder structure.

db.sqlite3

Creation of First Web Application:

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

The following is the command to create application.

python manage.py startapp firstApp

D:\djangoprojects\firstProject>python manage.py startapp firstApp

The following is the folder structure got created.

```
D:\djangoprojects>
└──firstProject
   │  db.sqlite3
   │  manage.py
   │
   ├──firstApp
   │  │  admin.py
   │  │  apps.py
   │  │  models.py
   │  │  tests.py
   │  │  views.py
   │  │  __init__.py
   │  │
   │  └──migrations
   │        __init__.py
   │
   └──firstProject
      │  settings.py
      │  urls.py
      │  wsgi.py
      │  __init__.py
      │
```

Note: Observe that Application contains 6 files and project contains 4 files+ one special file: manage.py

1) __init__.py:

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

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 responses.

7) **Migrations Folder:**

This directory stores database specific information related to models.

Note: The most important 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.

In **settings.py:**

```
1) INSTALLED_APPS = [
2)     'django.contrib.admin',
3)     'django.contrib.auth',
4)     'django.contrib.contenttypes',
5)     'django.contrib.sessions',
6)     'django.contrib.messages',
7)     'django.contrib.staticfiles',
8)     'firstApp'
9) ]
```

Activity-2: Create a view for our application in views.py.

- View is responsible 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

- In this application we are using Function based views.

views.py:

```
1) from django.shortcuts import render
2) from django.http import HttpResponse
```

```
3)
4)  # Create your views here.
5)  def display(request):
6)      s='<h1>Hello Students welcome to  Django classes!!!</h1>'
7)      return HttpResponse(s)
```

Note:

1) Each view will be specified as one function in views.py.
2) In the above example display is the name of function which is nothing but one view.
3) Each view should take atleast one argument (request)
4) Each view should return HttpResponse object with our required response.

Diagram

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 file.

- This url-pattern will be used by end-user to send request for our views.
- The 'urlpatterns' list routes URLs to views.

For functional views we have to do the following 2 activities:

1) Add an import: from firstApp import views
2) Add a URL to urlpatterns: url(r'^greeting/', views.display)

urls.py:

```
1)  from django.conf.urls import url
2)  from django.contrib import admin
3)  from firstApp import views
4)
5)  urlpatterns = [
6)      url(r'^admin/', admin.site.urls),
7)      url(r'^greetings/', views.display),
8)  ]
```

Whenever end user sending the request with urlpattern: greeting then disply() function will be executed and provide required response.

Activity-4: Start Server and Send the request

py manage.py runserver

http://127.0.0.1:8000/greetings

Http Request flow in Django Application:

Diagram

1. Whenever end user sending the request first Django development server will get that request.

2. From the Request django will identify urlpattern and by using urls.py, the corresponding view will be identified.

3. The request will be forwared to the view. The corresponding function will be executed and provide required response to the end user.

Summary of Sequence of Activities related to Django Project:

1) **Creation of Django project**
   django-admin startproject firstProject

2) **Creation of Application in that project**
   py manage.py startapp firstApp

3) **Add application to the Project**
   (inside settings.py)

4) **Define view function inside views.py**
5) **Define url-pattern for our view inside urls.py**

6) **Start Server**
   py manage.py runserver

7) **Send the request**

How to change Django Server Port:

By default Django develoment server will run on port number: 8000. But we can change port number based on our requirement as follows.

py manage.py runserver 7777

Now Server running on port number: 7777
We have to send the request with this port number only

http://127.0.0.1:7777/greetings/
http://127.0.0.1:8000/time/

Defining URL Patterns at Application Level instead of Project Level:

A Django project can contain multiple applications and each application can contain multiple views.Defining url-patterns for all views of all applications inside urls.py file of project creates maintenance problems and reduces reusability of applications.

We can solve this problem by defining url-patterns 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() method.

Demo Application:

1) **Creation of Project**
   django-admin startproject urlProject

2) **Creation of Application**
   py manage.py startapp urlApp

3) **Add our application to the Project inside settings.py file**

```
INSTALLED_APPS = [
    .....
```

```
    'urlApp'
]
```

4) Define View Function in views.py

```
1) from django.http import HttpResponse
2) def appurlinfo(request):
3)
4)    s='<h1>Application Level urls Demo</h1>'
5)    return HttpResponse(s)
```

5) Create a seperate urls.py file inside application

```
1) from django.conf.url import url
2)  from urlApp import views
3)  urlpatterns=[
4)     url(r'^test/',views.appurlinfo)
5)     ]
```

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

```
from django.conf.urls. import include
urlpatterns=[
        ....
        url(r'^urlApp/',include('urlApp.urls'),
]
```

7) Run Server
   py manage.py runserver

8) Send Request
   http://127.0.0.1:8000/urlApp/test

Advantages:

The main advantages of defining urlpatterns at application level instead of project level are

1) It promotes reusability of Django Applications across multiple projects
2) Project level urls.py file will be clean and more readable

Django Templates:

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

1) **It reduces readability because Python code mixed with html code**
2) **No seperation of roles. Python developer has to concentrate on both python code and HTML Code.**
3) **It does not promote reusability of code**

We can overcome these problems by seperating html code into a seperate html file.This html file is nothing but template.
From the Python file (views.py file) we can use these templates based on our requirement.

We have to write templates at project level only once and we can use these in multiple applications.

Python Stuff required to develop Template Based Application:

1) **To know the current Python file name**
```
print(__file__)    #test.py
```

2) **To know absolute path of current Python File Name**
```
import os
print(os.path.abspath(__file__))
```
Output: D:\currentdirectory\test.py

3) **To know Base Directory name of the current file**
```
print(os.path.dirname(os.path.abspath(__file__)))
```
Output: D:\classes

4) **Inside D:\currentdirectorythere is one folder named with templates. To know its absolute path**

```
import os
BASE_DIR=os.path.dirname(os.path.abspath(__file__))
TEMPLATE_DIR=os.path.join(BASE_DIR,'templates')
print(TEMPLATE_DIR)
```

Output: D:\classes\templates

Note: **The main advantage of this approach is we are not required to hard code system specific paths (locations) in our python script.**

Steps to develop Template Based Application:

1) **Creation of Project**
   django-admin startproject templateProject

2) **Creation of Application**
   py manage.py startapp testApp

3) **Add this application to the project in settings.py file,so that Django aware of application**

4) **Create a 'templates' folder inside main project folder.**
   In that templates folder create a seperate folder named with testApp to hold that particular application specific templates.

5) **Add templates folder to settings.py file so that Django can aware of our templates.**

```
TEMPLATES = [
{
    ...,
    'DIRS': ['D:\djangoprojects\templateProject\templates'],

},]
```

It is not recommended to hard code system specific locations in settings.py file. To overcome this problem, we can generate templates directory path programatically as follows.

```
import os
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
TEMPLATE_DIR = os.path.join(BASE_DIR,'templates')
```

Specify this TEMPLATE_DIR inside settings.py as follows

```
TEMPLATES = [
++{
   ...,
   'DIRS': [TEMPLATE_DIR,],

},]
```

6) **Create html file inside templateProject/templates/testApp folder. This html file is +nothing but template.**

wish.html:

```
1)  <!DOCTYPE html>
2)  <html lang="en" dir="ltr">
3)    <head>
4)      <meta charset="utf-8">
5)      <title>First Template Page</title>
6)    </head>
7)    <body>
8)      <h1>Hello welcome to Second Hero of MVT: Templates</h1>
9)    </body>
10) </html>
```

7) Define Function based view inside views.py file

```
1)  from django.shortcuts import render
2)  def wish(request):
3)      return render(request,'testApp/wish.html')
```

8) Define URL Pattern either at application level or at project level.

9) Run Server and send Request.

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. Just it is special syntax.

Template Tag Syntax for inserting Text Data: **{ {insert_date} }**

This template tag we have to place inside template file (ie html file) and we have to provide insert_date value from python views.py file.

Diagram

Demo Application to send Date and Time from views.py to Template File:

wish.html:

```
1)  <!DOCTYPE html>
2)  <html lang="en" dir="ltr">
3)    <head>
4)      <meta charset="utf-8">
5)      <title>First Template Page</title>
6)      <style >
7)      h1{
8)        color:white;
9)        background: red;
10)     }
11)     </style>
12)   </head>
13)   <body>
14)     <h1>Hello Server Current Date and Time : <br>
15)       {{insert_date}}
16)     </h1>
17)   </body>
18) </html>
```

views.py:

```
1)  from django.shortcuts import render
2)  import datetime
3)  def wish(request):
4)      date=datetime.datetime.now()
5)      my_dict={'insert_date':date}
6)      return render(request,'testApp/wish.html',context=my_dict)
```

Note: The values to the template variables should be passed from the view in the form of dictionary as argument to context.

Application to wish end user based on time:

wish.html:

```
1)  <!DOCTYPE html>
2)  <html lang="en" dir="ltr">
3)    <head>
```

=;[

```
    <meta charset="utf-8">
    <title>First Template Page</title>
    <style >
    #h11{
      color:red;
    }
    #h12{
      color:green;
    }
    </style>
  </head>
  <body>
    <h1 id=h11>{{insert_msg}}</h1>
    <h1 id=h12>Current Date and Time : {{insert_date}}</h1>
  </body>
</html>
```

views.py:

```
from django.shortcuts import render
import datetime

# Create your views here.
def wish(request):
    date=datetime.datetime.now()
    msg='Hello Guest !!!! Very Very Good '
    h=int(date.strftime('%H'))
    if h<12:
        msg +='Morning!!!'
    elif h<16:
        msg +='AfterNoon!!!'
    elif h<21:
        msg +='Evening!!!'
    else:
        msg='Hello Guest !!!! Very Very Good Night!!!'
    my_dict={'insert_date':date,'insert_msg':msg}
    return render(request,'testApp/wish.html',context=my_dict)
```

Working with Static Files:

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

Process to include Static Files inside Template:

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

settings.py:

```
STATIC_DIR=os.path.join(BASE_DIR,'static')

..
STATIC_URL = '/static/'

STATICFILES_DIRS=[
STATIC_DIR,
]
```

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

5) Use Template Tags to insert image

   At the beginning of HTML just after <!DOCTYPE html> we have to include the following template tag `{% load staticfiles %}`

   Just we are conveying to the Django to load all static files.

   We have to include image file as follows `<img src="{% static "images/divine3.jpg"% }">`

wish.html:

```
<!DOCTYPE html>
{% load staticfiles %}
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>First Template Page</title>
    <style >
    #h11{
      color:red;
    }
    #h12{
      color:green;
    }
```

```
  </style>
</head>
<body>
  <h1 id=h11>{{insert_msg}}</h1>
  <h1 id=h12>Current Date and Time : {{insert_date}}</h1>
  <h1>This climate preferable image is:</h1>
  <img src="{% static "images/divine3.jpg" %}" alt="">
</body>
</html>
```

views.py:

```
from django.shortcuts import render
import datetime

# Create your views here.
def wish(request):
    date=datetime.datetime.now()
    msg=None
    h=int(date.strftime('%H'))
    if h<12:
        msg='Hello Guest !!!! Very Very Good Morning!!!'
    elif h<16:
        msg='Hello Guest !!!! Very Very Good AfterNoon!!!'
    elif h<21:
        msg='Hello Guest !!!! Very Very Good Evening!!!'
    else:
        msg='Hello Guest !!!! Very Very Good Night!!!'
    my_dict={'insert_date':date,'insert_msg':msg}
    return render(request,'testApp/wish.html',context=my_dict)
```

How to include css file:

1) Create a folder 'css' inside static and place our demo.css file in that css folder.

demo.css:

```
img{
  height: 500px;
  width: 500px;
  border: 10px red groove;
  margin:0% 20%;
}
h1{
```

```
    color:blue;
    text-align: center;
}
```

2) In the template html file we have to include this css file. We have to do this by using link tag inside head tag.

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

NEWS PORTAL:

index.html:

```
<!DOCTYPE html>
{% load staticfiles%}
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
    <link rel="stylesheet" href="{% static "css/demo.css"%}">
  </head>
  <body>
    <h1>Welcome to  NEWS PORTAL</h1>
    <ul>
      <li> <a href="/movies">Movies Information</a> </li>
      <li> <a href="/sports">Sports Information</a> </li>
      <li> <a href="/politics">Politics Information</a> </li>
    </ul>
  </body>
</html>
```

news.html:

```
<!DOCTYPE html>
{% load staticfiles %}
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
    <link rel="stylesheet" href="{% static "css/demo.css"%}">
  </head>
  <body>
      <h1>{{head_msg}}</h1>
      <ul>
        <li> <h2>{{sub_msg1}}</h2> </li>
```

```
13)       <li> <h2>{{sub_msg2}}</h2> </li>
14)       <li> <h2>{{sub_msg3}}</h2> </li>
15)      </ul>
16)      <img src="{% static "images/sunny.jpg" %}" alt="">
17)      <img src="{% static "images/guido.jpg" %}" alt="">
18)      <img src="{% static "images/divine3.jpg" %}" alt="">
19)  </body>
20)          </html>
```

views.py:

```
1)  from django.shortcuts import render
2)
3)  # Create your views here.
4)  def moviesInfo(request):
5)     my_dict={'head_msg':'Movies  Information',
6)              'sub_msg1':'Sonali slowly getting cured',
7)              'sub_msg2':'Bahubali-3 is just planning',
8)              'sub_msg3':'Salman Khan ready to marriage',
9)              'photo':'images/sunny.jpg'}
10)    return render(request,'news.html',context=my_dict)
11) def sportsInfo(request):
12)    my_dict={'head_msg':'Sports  Information',
13)             'sub_msg1':'Anushka Sharma Firing Like anything',
14)             'sub_msg2':'Kohli updating in game anything can happend',
15)             'sub_msg3':'Worst Performance by India-Sehwag',
16)             }
17)    return render(request,'news.html',context=my_dict)
18) def politicsInfo(request):
19)    my_dict={'head_msg':'Politics  Information',
20)             'sub_msg1':'Achhce Din Aaa gaya',
21)             'sub_msg2':'Rupee Value now 1$:70Rs',
22)             'sub_msg3':'In India Single Paisa Black money No more',
23)             }
24)    return render(request,'news.html',context=my_dict)
25) def index(request):
26)    return render(request,'index.html')
```

settings.py:

```
1)  import os
2)
3)  BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
4)  TEMPLATES_DIR=os.path.join(BASE_DIR,'templates')
5)  STATIC_DIR=os.path.join(BASE_DIR,'static')
```

```
6)
7)  INSTALLED_APPS = [
8)      'django.contrib.admin',
9)      'django.contrib.auth',
10)     'django.contrib.contenttypes',
11)     'django.contrib.sessions',
12)     'django.contrib.messages',
13)     'django.contrib.staticfiles',
14)     'newsApp'
15) ]
16)
17) TEMPLATES = [
18)     {
19)         '
20)         'DIRS': [TEMPLATES_DIR,],
21)
22)           ],
23)         },
24)     },
25) ]
26)
27) STATIC_URL = '/static/'
28) STATICFILES_DIRS=[
29) STATIC_DIR,
30) ]
```

urls.py:

```
1)  from django.conf.urls import url
2)  from django.contrib import admin
3)  from newsApp import views
4)
5)  urlpatterns = [
6)      url(r'^admin/', admin.site.urls),
7)      url(r'^movies/', views.moviesInfo),
8)      url(r'^sports/', views.sportsInfo),
9)      url(r'^politics/', views.politicsInfo),
10)     url(r'^$', views.index),
11) ]
```

Single Project with Multiple Applications:

urls.py:

```
1)  from django.conf.urls import url
2)  from django.contrib import admin
3)  from firstApp import views as v1
4)  from secondApp import views as v2
5)
6)  urlpatterns = [
7)      url(r'^admin/', admin.site.urls),
8)      url(r'^firstwish/', v1.wish1),
9)      url(r'^secondwish/', v2.wish2),
10) ]
```

Working with Models and Databases:

- As the part of web application development, compulsory we required to interact with database to store our data and to retrieve our stored data.
- Django provides a big in-built support for database operations. Django provides one inbuilt database sqlite3.
- For small to medium applications this database is more enough. Django can provide support for other databases also like oracle, mysql,postgresql etc

Database Configuration:

- Django by default provides sqlite3 database. If we want to use this database,we are not required to do any configurations.
- The default sqllite3 configurations in settings.py file are declared as follows.

settings.py:

```
1)  DATABASES = {
2)      'default': {
3)          'ENGINE': 'django.db.backends.sqlite3',
4)          'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
5)      }
6)  }
```

- If we don't want sqlite3 database then we have to configure our own database with the following parameters.

1) ENGINE: Name of Database engine
2) NAME: Database Name
3) USER: Database Login user name
4) PASSWORD: Database Login password
5) HOST: The Machine on which database server is running
6) PORT: The port number on which database server is running

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

How to Check Django Database Connection:

- We can check whether django database configurations are properly configured or not by using the following commands from the shell

- D:\djangoprojects\modelProject>py manage.py shell
  >>> from django.db import connection
  >>> c = connection.cursor()

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

Configuration of MySQL Database:

- First we have to create our own logical database in the mysql.
  mysql> create database employeedb;
- We have to install mysqlclient by using pip as follows
  pip install --only-binary :all: mysqlclient

settings.py:

```
1) DATABASES = {
2)     'default': {
3)         'ENGINE': 'django.db.backends.mysql',
4)         'NAME': 'employeedb',
5)         'USER':'root',
6)         'PASSWORD':'root'
7)     }
8) }
```

Checking Configurations:

D:\djangoprojects\modelProject>py manage.py shell

>>> from django.db import connection
>>> c = connection.cursor()

Configuration of Oracle Database:

```
1) DATABASES = {
2)     'default': {
3)         'ENGINE': 'django.db.backends.oracle',
4)         'NAME': 'XE',
```

```
        'USER':'scott',
        'PASSWORD':'tiger'
    }
}
```

Note: We can find oracle database name by using the following command.

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

Model Class:

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

1. Create a project and application and link them.

```
django-admin startproject modelProject
cd modelProject/
python manage.py startapp testApp
```

After creating a project and application, in the models.py file, write the following code:

models.py:

```
from django.db import models

# Create your models here.

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

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

table_name: appName_Employee
fields: eno, ename, esal and eaddr. And one extra field: id

behaviors: eno is of type Integer, ename is of type Char and max_length is 30 characters.

Hence,

Model Class = Database Table Name + Field Names + Field Behaviors

Converting Model Class into Database specific SQL Code:

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

Python manage.py make migrations

It results the following :
Migrations for 'testApp':
testApp/migrations/0001_initial.py
- Create model Employee

How to see corresponding SQL Code of Migrations:

To see the generated SQL Code, we have to use the following command "sqlmigrate"
python manage.py sqlmigrate testApp 0001

```
1)  BEGIN;
2)  --
3)  -- Create model Employee
4)  --
5)  CREATE TABLE "testApp_employee" ("id" integer NOT NULL PRIMARY KEY AUTOIN
    CREMENT, "eno" integer NOT NULL, "ename" varchar(30) NOT NULL, "esal" real N
    OT NULL, "eaddr" varchar(30) NOT NULL);
6)  COMMIT;
```

Note: **Here 0001 is the file passed as an argument**

"id" field:

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

How to execute generated SQL Code (migrate Command):

After generating sql code, we have to execute that sql code to create table in database. For this, we have to use 'migrate' command.

python manage.py migrate

Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions, testApp
Running migrations:

- Applying contenttypes.0001_initial... OK
- Applying auth.0001_initial... OK
- Applying admin.0001_initial... OK
- Applying admin.0002_logentry_remove_auto_add... OK
- Applying contenttypes.0002_remove_content_type_name... OK
- Applying auth.0002_alter_permission_name_max_length... OK
- Applying auth.0003_alter_user_email_max_length... OK
- Applying auth.0004_alter_user_username_opts... OK
- Applying auth.0005_alter_user_last_login_null... OK
- Applying auth.0006_require_contenttypes_0002... OK
- Applying auth.0007_alter_validators_add_error_messages... OK
- Applying auth.0008_alter_user_username_max_length... OK
- Applying sessions.0001_initial... OK
- Applying testApp.0001_initial... OK

Note: Now tables will be created in the database.

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 my 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:

```
1) from django.contrib import admin
2) from testApp.models import Employee
3)
4) # Register your models here.
5)
6) admin.site.register(Employee)
```

Creation of Super User to login to admin Interface:

- We can create super user with the following command by providing username, mailid, password.
- python manage.py createsuperuser
- We can login to admin interface → Start the server and login to admin interface using the created credentials.
- python manage.py runserver
- Open the following in browser: http://127.0.0.1:8000/admin/

Difference between makemigrations and migrate:

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

To Display Data in admin Interface in Browser:

models.py:

```
1)  from django.db import models
2)
3)  # Create your models here.
4)
5)  class Employee(models.Model):
6)      eno=models.IntegerField()
7)      ename=models.CharField(max_length=30)
8)      esal=models.FloatField()
9)      eaddr=models.CharField(max_length=30)
10)
11)     def __str__(self):
12)         return 'Employee Object with eno: +str(self.no)'
```

admin.py:

```
1)  from django.contrib import admin
2)  from testApp.models import Employee
3)
4)  # Register your models here.
5)
6)  class EmployeeAdmin(admin.ModelAdmin):
7)      list_display=['eno','ename','esal','eaddr']
8)
9)  admin.site.register(Employee,EmployeeAdmin)
```

Now we can write views to get data from the database and send to template.
Before writing views.py file, create “templates” and “static” folder with respective application folders and HTML and CSS files and link them in settings.py file.

Views.py:

```
1)  from django.shortcuts import render
2)  from testApp.models import Employee
3)
4)  # Create your views here.
5)  def empdata(request):
6)      emp_list=Employee.objects.all()
7)      my_dict={'emp_list':emp_list}
8)      return render(request, 'testApp/emp.html', context=my_dict)
```

emp.html

```
1)  <!DOCTYPE html>
2)  {% load staticfiles %}
3)  <html lang="en" dir="ltr">
4)    <head>
5)      <meta charset="utf-8">
6)      <title></title>
7)      <link rel="stylesheet" href="{% static '/css/demo.css'%}">
8)    </head>
9)
10)  <body>
11)    <h1> The employees list is : </h1>
12)
13)    {% if emp_list %}
14)    <table>
15)      <thead>
16)        <th> eno </th>
17)        <th> ename </th>
18)        <th> esal </th>
19)        <th> eaddr </th>
20)      </thead>
21)
22)      {% for emp in emp_list %}
23)      <tr>
24)        <td> {{emp.eno}}</td>
25)        <td>{{emp.ename}}</td>
26)        <td>{{emp.esal}}</td>
27)        <td> {{emp.eaddr}}</td>
28)      </tr>
```

```
    {% endfor %}

  </table>
  {%else%}
  <p> No records found </p>
  {% endif %}

 </body>
</html>
```

MVT Diagram:

FAQs:

1) How to configure database inside settings.py?
2) How to check connections?
3) How to define Model class inside models.py
4) How we can perform makemigrations?
5) How we can perform migrate?
6) How to add our model to admin interface inside admin.py
7) To display total data how to write ModelAdmin class inside admin.py
8) How to createsuperuser?
9) How to login to admin interface and add data to our tables?
10) How to see generated sqlcode b'z of makemigrations

Faker Module:

We can use Faker Module to generate fake data for our database models.

```
from faker import Faker
from random import *
fakegen=Faker()
name=fakegen.name()
print(name)
first_name=fakegen.first_name()
last_name=fakegen.last_name()
print(first_name)
print(last_name)
date=fakegen.date()
print(date)
```

```
number=fakegen.random_number(5)
print(number)
email=fakegen.email()
print(email)
print(fakegen.city())
print(fakegen.random_int(min=0, max=9999))
print(fakegen.random_element(elements=('Project Manager', 'TeamLead', 'Software Engineer')))
```

Note: Working with mysql db(studentinfo project)

Django Forms:

- It is the very important concept in web development.
- The main purpose of forms is to take user input.

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

Advantages of Django Forms over HTML Forms:

1) We can develop forms very easily with python code
2) We can generate HTML Form widgets/components (like textarea, email, pwd etc) 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 on forms will become easy etc.

Process to generate Django Forms:

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

forms.py:

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

Note: name and marks are the field names which will be available in html form

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

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

views.py:

```
1)  from django.shortcuts import render
2)  from . import forms
3)
4)  # Create your views here.
5)  def studentinputview(request):
6)      form=forms.StudentForm()
7)      my_dict={'form':form}
8)      return render(request,'testapp/input.html',context=my_dict)
```

Alternative Short Way:

```
1)  def studentinputview(request):
2)      form=forms.StudentForm()
3)      return render(request,'testapp/input.html',{'form':form})
```

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

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

Inside template file we have to use template tag to inject form {{form}}

It will add only form fields. But there is no <form> tag and no submit button.
Even the fields are not arranged properly.It is ugly form.

We can make proper form as follows

```
1)  <h1>Registration Form</h1>
2)   <div class="container" align="center">
3)      <form   method="post">
4)    {{form.as_p}}
5)    <input type="submit" class="btn btn-primary" name="" value="Submit">
6)      </form>
7)
8)   </div>
```

input.html:

```
1)  <!DOCTYPE html>
2)  {%load staticfiles%}
```

```
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <link rel="stylesheet" href="{%static "css/bootstrap.css"%}">
    <link rel="stylesheet" href="{%static "css/demo2.css"%}">
    <title></title>
  </head>
  <body>
    <h1>Registration Form</h1>
    <div class="container" align="center">
      <form   method="post">
          {{form.as_p}}
          <input type="submit" class="btn btn-primary" name="" value="Submit">
      </form>
    </div>
  </body>
</html>
```

If we submit this form we will get 403 status code response
Forbidden (403)
CSRF verification failed. Request aborted.

Help
Reason given for failure:

CSRF token missing or incorrect.

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

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

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

<input type='hidden' name='csrfmiddlewaretoken' value='1ZqIJJqTLMVa6RFAyPJh7pwzyFmdiHzytLxJIDzAkKULJz4qHcetLoKEsRLwyz4h'/>

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

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

How to process Input Data from the form inside views.py File:

We required to modify views.py file. The end user provided input is available in a dictionary named with 'cleaned_data'

views.py:

```
1)  from django.shortcuts import render
2)  from . import forms
3)
4)  # Create your views here.
5)  def studentinputview(request):
6)      form=forms.StudentForm()
7)      if request.method=='POST':
8)          form=forms.StudentForm(request.POST)
9)          if form.is_valid():
10)             print('Form validation success and printing data')
11)             print('Name:',form.cleaned_data['name'])
12)             print('Marks:',form.cleaned_data['marks'])
13)     return render(request,'testapp/input.html',{'form':form})
```

project: formproject

Student FeedBack Form Project

forms.py:

```
1)  from django import forms
2)
3)  class FeedBackForm(forms.Form):
4)      name=forms.CharField()
5)      rollno=forms.IntegerField()
6)      email=forms.EmailField()
7)      feedback=forms.CharField(widget=forms.Textarea)
```

views.py:

```
1)  from django.shortcuts import render
2)  from . import forms
3)
4)  def feedbackview(request):
5)      form=forms.FeedBackForm()
6)      if request.method=='POST':
7)          form=forms.FeedBackForm(request.POST)
8)          if form.is_valid():
9)              print('Form Validation Success and printing information')
10)             print('Name:',form.cleaned_data['name'])
11)             print('Roll No:',form.cleaned_data['rollno'])
12)             print('Email:',form.cleaned_data['email'])
13)             print('FeedBack:',form.cleaned_data['feedback'])
14)     return render(request,'testapp/feedback.html',{'form':form})
```

feedback.html:

```
1)  <!DOCTYPE html>
2)  {% load staticfiles%}
3)  <html lang="en" dir="ltr">
4)    <head>
5)      <meta charset="utf-8">
6)      <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7
    /css/bootstrap.min.css" integrity="sha384-
    BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" c
    rossorigin="anonymous">
7)      <link rel="stylesheet" href="{% static "css/demo4.css" %}">
8)      <title></title>
9)    </head>
10)   <body>
11)     <div class="container" align='center'>
12)       <h1>Student Feedback Form</h1><hr>
13)         <form class="" action="index.html" method="post">
14)           {{form.as_p}}
15)           {% csrf_token %}
16)           <input type="submit"  class="btn btn-primary" value="Submit Feedback">
17)         </form>
18)     </div>
19)   </body>
20) </html>
```

Form Validations:

Once we submit the form we have to perform validations like

1) Length of the field should not be empty
2) The max number of characters should be 10
3) The first character of the name should be 'd' etc

We can implement validation logic by using the following 2 ways.

1) Explicitly by the programmer by using clean methods
2) By using Django inbuilt validators

Note: All validations should be implemented in the forms.py file

1) Explicitly by the Programmer by using Clean Methods:

- The syntax of clean method: `clean_fieldname(self)`
- In the FormClass for any field if we define clean method then at the time of submit the form, Django will call this method automatically to perform validations. If the clean method won't raise any error then only form will be submitted.

```
1)  from django import forms
2)
3)  class FeedBackForm(forms.Form):
4)      name=forms.CharField()
5)      rollno=forms.IntegerField()
6)      email=forms.EmailField()
7)      feedback=forms.CharField(widget=forms.Textarea)
8)
9)      def clean_name(self):
10)         inputname=self.cleaned_data['name']
11)         if len(inputname) < 4:
12)             raise forms.ValidationError('The Minimum no of characters in the name field
    should be 4')
13)         return inputname
```

- The returned value of clean method will be considered by Django at the time of submitting the form.

forms.py:

```
1)  from django import forms
2)  from django.core import validators
3)
4)  class FeedBackForm(forms.Form):
5)      name=forms.CharField()
6)      rollno=forms.IntegerField()
```

```
    email=forms.EmailField()
    feedback=forms.CharField(widget=forms.Textarea)

  def clean_name(self):
      print('validating name')
      inputname=self.cleaned_data['name']
      if len(inputname) < 4:
          raise forms.ValidationError('The Minimum no of characters in the name field should be 4')
      return inputname+'durga'
  def clean_rollno(self):
      inputrollno=self.cleaned_data['rollno']
      print('Validating rollno field')
      return inputrollno
  def clean_email(self):
      inputemail=self.cleaned_data['email']
      print('Validating email field')
      return inputemail

  def clean_feedback(self):
      inputfeedback=self.cleaned_data['feedback']
      print('Validating feedback field')
      return inputfeedback
```

Server Console:

validating name
Validating rollno field
Validating email field
Validating feedback field
Form Validation Success and printing information
Name: Durgadurga
Roll No: 101
Email: durgaadvjava@gmail.com
FeedBack: This is sample feedback

Note:

1) Django will call these filed level clean methods automatically and we are not required to call explicitly.
2) Form validation by using clean methods is not recommended.

2) Django's Inbuilt Core Validators:

- Django provides several inbuilt core validators to perform very common validations. We can use these validators directly and we are not required to implement.
- Django's inbuilt validators are available in the django.core module.
- from django.core import validators
- To validate Max number of characters in the feedback as 40,we have to use inbuilt validators as follows.

forms.py:

```
1) from django import forms
2) from django.core import validators
3)
4) class FeedBackForm(forms.Form):
5)     name=forms.CharField()
6)     rollno=forms.IntegerField()
7)     email=forms.EmailField()
8)     feedback=forms.CharField(widget=forms.Textarea,validators=[validators.MaxLe
   ngthValidator(40)])
```

Note: We can use any number of validators for the same field

```
feedback = forms.CharField(widget = forms.Textarea,validators =
[validators.MaxLengthValidator(40),validators.MinLengthValidator(10)])
```

Note: Usage of built in validators is very easy when compared with clean methods.

How to implement Custom Validators by using the same Validator Parameter:

The value of name parameter should starts with 'd' or 'D'. We can implement this validation as follows

```
1) def starts_with_d(value):
2)     if value[0].lower() != 'd':
3)         raise forms.ValidationError('Name should be starts with d | D')
4)
5) class FeedBackForm(forms.Form):
6)     name=forms.CharField(validators=[starts_with_d])
7)     rollno=forms.IntegerField()
```

Validation of Total Form directly by using a Single Clean Method:

Whenever we are submitting the form Django will call clean() method present in our Form class. In that method we can implement all validations.

forms.py:

```
1)  from django import forms
2)  from django.core import validators
3)
4)  class FeedBackForm(forms.Form):
5)      name=forms.CharField()
6)      rollno=forms.IntegerField()
7)      email=forms.EmailField()
8)      feedback=forms.CharField(widget=forms.Textarea)
9)
10)     def clean(self):
11)         print('Total Form  Validation...')
12)         total_cleaned_data=super().clean()
13)         inputname=total_cleaned_data['name']
14)         if inputname[0].lower() != 'd':
15)             raise forms.ValidationError('Name parameter should starts with d')
16)         inputrollno=total_cleaned_data['rollno']
17)         if inputrollno <=0:
18)             raise forms.ValidationError('Rollno should be > 0')
```

How to Check Original pwd and reentered pwd are same OR not:

forms.py:

```
1)  from django import forms
2)  from django.core import validators
3)
4)  class FeedBackForm(forms.Form):
5)      name=forms.CharField()
6)      password=forms.CharField(widget=forms.PasswordInput)
7)      rpassword=forms.CharField(label='Re Enter Password',widget=forms.PasswordIn
    put)
8)      rollno=forms.IntegerField()
9)      email=forms.EmailField()
10)     feedback=forms.CharField(widget=forms.Textarea)
```

```
11)
12)    def clean(self):
13)        print('validating passwords match...')
14)        total_cleaned_data=super().clean()
15)        fpwd=total_cleaned_data['password']
16)        spwd=total_cleaned_data['rpassword']
17)        if fpwd != spwd:
18)            raise forms.ValidationError('Both passwords must be matched')
```

How to prevent Requests from BOT:

Generally form requests can be send by end user. Sometimes we can write automated programming script which is responsible to fill the form and submit. This automated programming script is nothing but BOT.

The main objectives of BOT requests are:

1) **To create unnecessary heavy traffic to the website, which may crash our application.**
2) **To spread malware (viruses)**

Being web developer compulsory we have to think about BOT requests and we have to prevent these requests.

How to prevent BOT Requests:

- **In the form we will place one hidden form field, which is not visible to the end user. Hence there is no chance of providing value to this hidden field.**
- **But BOT will send the value for this hidden field also.If hidden field got some value means it is the request from BOT and prevent that form submission.**

```
1)  from django import forms
2)  from django.core import validators
3)
4)  class FeedBackForm(forms.Form):
5)      name=forms.CharField()
6)      password=forms.CharField(widget=forms.PasswordInput)
7)      rpassword=forms.CharField(label='Re Enter Password',widget=forms.PasswordIn
    put)
8)      rollno=forms.IntegerField()
9)      email=forms.EmailField()
10)     feedback=forms.CharField(widget=forms.Textarea)
11)     bot_handler=forms.CharField(required=False,widget=forms.HiddenInput)
12)
13)     def clean(self):
14)         print('validating passwords match...')
15)         total_cleaned_data=super().clean()
```

```
        fpwd=total_cleaned_data['password']
        spwd=total_cleaned_data['rpassword']
        if fpwd != spwd:
            raise forms.ValidationError('Both passwords must be matched')
        bot_handler_value=total_cleaned_data['bot_handler']
        if len(bot_handler_value)>0:
            raise forms.ValidationError('Request from BOT...cannot be submitted!!!')
```

Notes with BOT Field:

```
class FeedBackForm(forms.Form):
    normal fields..
    bot_handler=forms.CharField(required=False,widget=forms.HiddenInput)

    def clean(self):
        total_cleaned_data=super().clean()
        bot_handler_value=total_cleaned_data['bot_handler']
        if len(bot_handler_value)>0:
            raise forms.ValidationError('Request from BOT...cannot be submitted!!!')
```

Note: Other ways to prevent BOT requests

1) By using Captchas
2) By using image recognizers
   (like choose 4 images where car present)

Note: Other way to create project → py -m django startproject modelformproject

Model Forms (Forms based on Model):

- Sometimes we can create form based on Model, such type of forms are called model based forms or model forms.
- The main advantage of model forms is we can grab end user input and we can save that input data very easily to the database.
- Django provides inbuilt support to develop model based forms very easily.

How to develop Model based Forms:

1) While develop FormClass instead of inheriting forms.Form class,we have to inherit forms.ModelForm class.

   class RegisterForm(forms.ModelForm):
   ....

2) We have to write one nested class (Meta class) to specify Model information and required fields.

```
class RegisterForm(forms.ModelForm):
  # field declarations if we are performing any custom validations.If we are not defining any custom validations then here we are not required to specify any field.

  class Meta:
    # we have to specify Model class name and requied fields
    model=Student
    fields='__all__'
```

Case-1: Instead of all fields if we want only selected fields, then we have to specify as follows

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

In the form only 3 fields will be considered.
If Model class contains huge number of fields and we required to consider very less number of fields in the form then we should use this approach.

Case-2:
Instead of all fields if we want to exclude certain fields,then we have to specify as follows

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

In the form all fields will be considered except field1 and field2.

If the Model class contains huge number of fields and if we want to exclude very few fields then we have to use this approach.

Q) In Model based Forms, how many ways are there to specify Fields Information

Ans: 3 ways

1) All fields
2) Include certain fields
3) Exclude certain fields

Note: The most commonly used approach is to include all fields.

How to Save User's Input Data to Database in Model based Forms:

We have to use save() method.

```
def student_view(request):
    ...
    if request.method=='POST':
        form=RegisterForm(request.POST)
            if form.is_valid():
                form.save(commit=True)
            ..
```

Demo project-1 (modelformproject):

models.py:

```
1)  from django.db import models
2)
3)  # Create your models here.
4)  class Student(models.Model):
5)      name=models.CharField(max_length=30)
6)      marks=models.IntegerField()
```

forms.py:

```
1)  from django import forms
2)  from testapp.models import Student
3)  class StudentForm(forms.ModelForm):
4)      #fields with validations
5)      class Meta:
6)          model=Student
7)          fields='__all__'
```

views.py:

```
1)  from django.shortcuts import render
2)  from . import forms
3)
4)  # Create your views here.
5)  def student_view(request):
6)      form=forms.StudentForm
7)      if request.method=='POST':
```

```
        form=forms.StudentForm(request.POST)
        if form.is_valid():
            form.save(commit=True)
    return render(request,'testapp/studentform.html',{'form':form})
```

Demo Project-2: movieproject

Advanced Templates:

1) Template Inheritance
2) Template Filters
3) Template tags for relative URLs

1) Template Inheritance:

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

base.html:

```
<!DOCTYPE html>
 html,css,bootstrap links
 <body>
   common code required for every child tempalte
   {% block child_block%}
     Anything outside of this block available to child tag.
     in child template the specific code should be in this block
   {%endblock%}
 </body>
</html>
```

child.html:

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

```
4)      child specific extra code
5)   {%endblock%}
```

Demo program: advtempproject

base.html:

```
1)  <!DOCTYPE html>
2)  {%load staticfiles%}
3)  <html lang="en" dir="ltr">
4)    <head>
5)      <meta charset="utf-8">
6)      <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7
    /css/bootstrap.min.css" integrity="sha384-
    BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" c
    rossorigin="anonymous">
7)      <link rel="stylesheet" href="{%static "css/advtemp.css"%}">
8)      <title></title>
9)    </head>
10)   <body>
11)     <nav class="navbar navbar-default navbar-fixed-top navbar-inverse">
12)       <div class="container-fluid">
13)
14)         <div class="navbar-header">
15)           <a class="navbar-brand" href="/">DURGA NEWS</a>
16)         </div>
17)           <ul class="nav navbar-nav">
18)             <li class="active"><a href="/">Home <span class="sr-
    only">(current)</span></a></li>
19)             <li><a href="/movies">Movies</a></li>
20)             <li><a href="/sports">Sports</a></li>
21)             <li><a href="/politics">Politics</a></li>
22)           </ul>
23)       </div><!-- /.container-fluid -->
24)  </nav>
25) <div class="container">
26)   {%block body_block%}
27)     <!-- outside of this block everything available to child tempaltes -->
28)   {%endblock%}
29) </div>
30) </body>
31) </html>
```

index.html:

```
1) <!DOCTYPE html>
2) {%extends 'testapp/base.html'%}
3)  {% block body_block %}
4)    <br><br><br><br><br><br>
5)    <h1>Welcome to DURGA NEWS PORTAL</h1>
6)  {%endblock%}
```

sports.html:

```
1) <!DOCTYPE html>
2) {%extends 'testapp/base.html'%}
3)  {% block body_block %}
4)  <br><br><br>
5)    <h1>Sports Information</h1>
6)  {%endblock%}
```

Advantages of Template Inheritance:

1) What ever code available in base template is by default available to child templates and we are not required to write again.Hence it promotes Code Reusability (DRY).
2) It reduces length of the code and improves readability.
3) It reduces development time.
4) It provides unique and same look and feel for total web application.

Note: Based on our requirement we can extend any number of base templates.i.e Multiple Inheritance is applicable for templates.

2)Tempalte Filters:

- In the template file, the injected data can be displayed by using template tags.
- {{emp.eno}}
- Before displaying to the end user if we want to perform some modification to the injected text, like cut some information or converting to title case etc, then we should go for Template filters.

Syntax of Template Filter: {{value|filtername:"argument"}}

Filter may take or may not take arguments.i.e arguments are optional.

Eg:

```
<li>{{msg1|lower}}</li>
```

msg1 will be displayed in lower case

```
<li>{{msg3|add:"Durga"}}</li>
```

"Durga" will be added to msg3 and then display the result to the end user

{{ msg|title }}

{{ my_date|date:"Y-m-d" }}

Note: There are tons of built in filters are available.
https://docs.djangoproject.com/en/2.1/ref/templates/builtins/#ref-templates-builtins-filters

How to Create our own Filter:

Based on our requirement we can create our own filter.

Steps:

1) Create a folder 'templatetags' inside our application folder.
2) Create a special file named with __init__.py inside templatetags folder, so that Django will consider this folder as a valid python package
3) Create a python file inside templatetags folder to define our own filters
   cust_filters.py -->any name

cust_filters.py:

```
1) from django import template
2) register=template.Library()
3)
4) def first_eight_upper(value):
5)     '''This is my own filter'''
6)     result=value[:8].upper()
7)     return result
8)
9) register.filter('f8upper',first_eight_upper)
```

Note: We can also register filter with the decorator as follows.

```
1) from django import template
2) register=template.Library()
3)
4) @register.filter(name='f8upper')
5) def first_eight_upper(value):
6)     '''This is my own filter'''
7)     result=value[:8].upper()
8)     return result
```

f8upper is the name of the filter which can be used inside template file.

4) Inside template file we have to load the filter file as follows(In the child template bunot in base template) {%load cust_filters%}

5) We can invoke the filter as follows {{msg|f8upper}}

movies.html:

```
1)  <!DOCTYPE html>
2)  {%extends 'testapp/base.html'%}
3)   {% block body_block %}
4)     <h1>Movies Information</h1><hr>
5)     {%load cust_filters%}
6)     <ul>
7)       <li>{{msg1|lower}}</li>
8)       <li>{{msg2|upper}}</li>
9)       <li>{{msg3|add:"--Durga"}}</li>
10)      <li>{{msg4|f8upper}}</li>
11)      <li>{{msg5}}</li>
12)  </ul>
13)  {%endblock%}
```

Eg 2: Custom Filter with argument

```
@register.filter(name='c_and_c')
def cut_and_concate(value,arg):
   result=value[:4]+str(arg)
   return result
```

Note: The main advantage of template filters is we can display the same data in different styles based on our requirement.

advtemp project

3) Template Tags for URLs:

- Thank You
- Thank You
- Thank You
- Thank You

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 remember client information for future purpose across multiple requests. Every request to the server is treated as new request.
- Hence some mechanism must be required at server side to remember client information across multiple requests.This mechanism is nothing but session management mechanism.
- The following are various session management mechanisms.

1) Cookies
2) Session API
3) URL Rewriting
4) Hidden Form Fields etc

Session Management By using Cookies:

- Cookie is a very small amount of information created by Server and maintained by client.

Diagram

Whenever client sends a request to the server,if server wants to remember client information for the future purpose then server will create cookie object with the required information. Server will send that Cookie object to the client as the part of response. Client will save that cookie in its local machine and send to the server with every consecutive request. By accessing cookies from the request server can remember client information.

How to Test our Browser Supports Cookies OR not:

We have to use the following 3 methods on the request object.

1) set_test_cookie()
2) test_cookie_worked()
3) delete_test_cookie()

views.py:

```
1)  from django.shortcuts import render
2)  from django.http import HttpResponse
3)
4)  # Create your views here.
5)  def index(request):
6)      request.session.set_test_cookie()
7)      return HttpResponse('<h1>index Page</h1>')
8)
9)  def check_view(request):
10)     if request.session.test_cookie_worked():
11)         print('cookies are working properly')
12)         request.session.delete_test_cookie()
13)         return HttpResponse('<h1>Checking Page</h1>')
```

Note: Before executing this program compulsory we should perform migrate.

Session Management by using Cookies:

views.py:

```
1)  from django.shortcuts import render
2)
3)  # Create your views here.
4)  def count_view(request):
5)      if 'count' in request.COOKIES:
6)          newcount=int(request.COOKIES['count'])+1
7)      else:
8)          newcount=1
9)      response=render(request,'testapp/count.html',{'count':newcount})
10)     response.set_cookie('count',newcount)
11)     return response
```

count.html:

```
1)  <!DOCTYPE html>
2)  <html lang="en" dir="ltr">
3)    <head>
4)      <meta charset="utf-8">
5)      <title></title>
6)      <style >
7)      span{
8)        font-size: 200px;
9)        font-weight: 900;
10)     }
```

```
11)
12)    </style>
13)  </head>
14)  <body>
15)    <h1>Page Count is: <span> {{count}}<span></h1>
16)  </body>
17) </html>
```

Session Management by using Cookies Demo Program-3:

```
1)  from django.shortcuts import render
2)  from testapp.forms import LoginForm
3)  import datetime
4)
5)  # Create your views here.
6)  def home_view(request):
7)      form=LoginForm()
8)      return render(request,'testapp/home.html',{'form':form})
9)
10) def date_time_view(request):
11)     # form=LoginForm(request.GET)
12)     name=request.GET['name']
13)     response=render(request,'testapp/datetime.html',{'name':name})
14)     response.set_cookie('name',name)
15)     return response
16)
17) def result_view(request):
18)     name=request.COOKIES['name']
19)     date_time=datetime.datetime.now()
20)     my_dict={'name':name,'date_time':date_time}
21)     return render(request,'testapp/result.html',my_dict)
```

home.html:

```
1)  <!DOCTYPE html>
2)  <html lang="en" dir="ltr">
3)    <head>
4)      <meta charset="utf-8">
5)      <title></title>
6)    </head>
7)    <body>
8)      <h1>Welcome to DURGASOFT</h1>
9)      <form  action="/second">
10)       {{form.as_p}}
11)       {%csrf_token%}
```

```
    <input type="submit" name="" value="Enter Name">
  </form>
 </body>
</html>
```

datetime.html:

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h1>Hello {{name}}</h1> <hr>
    <a href="/result">Click Here to get Date and Time</a>
  </body>
</html>
```

result.html:

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h1>Hello {{name}}</h1><hr>
    <h1>Current Date and Time:{{date_time}}</h1>
    <a href="/result">Click Here to get Updated Date and Time</a>
  </body>
</html>
```

Cookie Example-4:

Diagram

views.py:

```python
from django.shortcuts import render

# Create your views here.
def name_view(request):
    return render(request,'testapp/name.html')

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

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

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

name.html:

```html
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h1>Welcome to DURGASOFT</h1>
    <form action='/age'>
      Enter Name: <input type="text" name="name" value=""><br><br>
      <input type="submit" name="" value="Submit Name">
    </form>

  </body>
</html>
```

age.html:

```
1)  <!DOCTYPE html>
2)  <html lang="en" dir="ltr">
3)    <head>
4)      <meta charset="utf-8">
5)      <title></title>
6)    </head>
7)    <body>
8)      <h1>Hello {{name}}..</h1><hr>
9)        <form action='/gf'>
10)         Enter Age: <input type="text" name="age" value=""><br><br>
11)         <input type="submit" name="" value="Submit Age">
12)       </form>
13)   </body>
14)         </html>
```

gf.html:

```
1)  <!DOCTYPE html>
2)  <html lang="en" dir="ltr">
3)    <head>
4)      <meta charset="utf-8">
5)      <title></title>
6)    </head>
7)    <body>
8)      <h1>Hello {{name}}..</h1><hr>
9)
10)       <form action='/results'>
11)         Enter Girl Friend Name: <input type="text" name="gfname" value=""><br><br
    >
12)         <input type="submit" name="" value="Submit GFName">
13)       </form>
14)
15)   </body>
16) </html>
```

results.html:

```
1)  <!DOCTYPE html>
2)  <html lang="en" dir="ltr">
3)    <head>
4)      <meta charset="utf-8">
5)      <title></title>
6)    </head>
```

```
7)  <body>
8)    <h1>Hello {{name}} Thanks for providing info</h1>
9)    <h2>Please cross check your data and confirm</h2><hr>
10)   <ul>
11)     <li>Name:{{name}}</li>
12)     <li>Age:{{age}}</li>
13)     <li>Girl Friend Name:{{gfname}}</li>
14)   </ul>
15) </body>
16) </html>
```

Limitations of Cookies:

1) By using cookies we can store very less amount of information. The size of the cookie is fixed.Hence if we want to store huge amount of information then cookie is not best choice.
2) Cookie can hold only string information. If we want to store non-string objects then we should not use cookies.
3) Cookie information is stored at client side and hence there is no security.
4) Everytime with every request, browser will send all cookies related to that application,which creates network traffic problems.
5) There is a limit on the max number of cookies supported by browser.

To overcome all these limitations we should go for sessions.

Temporary vs Permanent Cookies:

- If we are not setting any max_age for the cookie,then the cookies will be stored in browsers cache.Once we closed browser automatically the cookies will be expired.Such type of cookies are called temporary Cookies.
- We can set temporary Cookie as follows: `response.set_cookie(name,value)`
- If we are setting max_age for the cookie,then cookies will be stored in local file system permanently.Once the specified max_age expires then only cookies will be expired.Such type of cookies are called permanent or persistent cookies. We can set Permanent Cookies as follows

```
response.set_cookie(name,value,max_age=180)
response.set_cookie(name,value,180)
```

- The time unit for max_age is in seconds.

Demo Program-3:

views.py:

```
1)  from django.shortcuts import render
2)  from testapp.forms import ItemAddForm
3)
4)  # Create your views here.
5)  def index(request):
6)      return render(request,'testapp/home.html')
7)  def additem(request):
8)      form=ItemAddForm()
9)      response=render(request,'testapp/additem.html',{'form':form})
10)     if request.method=='POST':
11)         form=ItemAddForm(request.POST)
12)         if form.is_valid():
13)             name=form.cleaned_data['itemname']
14)             quantity=form.cleaned_data['quantity']
15)             response.set_cookie(name,quantity,180)
16)             # return index(request)
17)     return response
18) def displayitem_view(request):
19)             return render(request,'testapp/showitems.html')
```

home.html:

```
1)  <!DOCTYPE html>
2)  <html lang="en" dir="ltr">
3)    <head>
4)      <meta charset="utf-8">
5)      <!-- Latest compiled and minified CSS -->
6)  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/cs
    s/bootstrap.min.css" integrity="sha384-
    BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" c
    rossorigin="anonymous">
7)      <title></title>
8)    </head>
9)    <body>
10)     <div class="container" align='center'>
11)       <div class="jumbotron">
12)         <h1>DURGASOFT ONLINE SHOPPING APP</h1>
13)         <a class="btn btn-primary btn-
    lg" href="/add" role="button">ADD ITEM</a>
14)         <a class="btn btn-primary btn-
    lg" href="/display" role="button">Display ITEMS</a>
15)       </div>
16)
17)     </div>
18)   </body>
```

```
19) </html>
```

additem.html:

```
1)  <!DOCTYPE html>
2)  <html lang="en" dir="ltr">
3)    <head>
4)      <meta charset="utf-8">
5)      <title></title>
6)      <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7
    /css/bootstrap.min.css" integrity="sha384-
    BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" c
    rossorigin="anonymous">
7)    </head>
8)    <body>
9)      <div class="container" align='center'>
10)       <h1>Add Item Form</h1>
11)       <form  method="post">
12)         {{form.as_p}}
13)         {%csrf_token%}
14)         <input type="submit" name="" value="Add Item">
15)       </form><br><br><br>
16)       <a class="btn btn-primary btn-
    lg" href="/display" role="button">Display ITEMS</a>
17)     </div>
18)   </body>
19) </html>
```

showitems.html:

```
1)  <!DOCTYPE html>
2)  <html lang="en" dir="ltr">
3)    <head>
4)      <meta charset="utf-8">
5)      <title></title>
6)    </head>
7)    <body>
8)      <h1>Total Cookies Information:</h1>
9)      {%if request.COOKIES %}
10)     <table border=2>
11)       <thead>
12)         <th>Cookie Name</th>
13)         <th>Cookie Value</th>
14)       </thead>
15)
```

```
16)     {% for key,value in request.COOKIES.items %}
17)     <tr>
18)       <td>{{key}}</td>
19)       <td>{{value}}</td>
20)     </tr>
21)     {% endfor %}
22)     </table>
23)    {%else%}
24)    <p>Cookie Information is not available</p>
25)   {%endif%}
26) </body>
27)         </html>
```

Session Management By using Session API:

(Django Session Framework)

Diagram

- Once client sends request to the server,if server wants to remember client information for the future purpose then server will create session object and store required information in that object. For every session object a unique identifier available which is nothing but sessionid.
- Server sends the corresponding session id to the client as the part of response. Client retrieves the session id from the response and save in the local file system. With every consecutive request client will that session id. By accessing that session id and corresponding session object server can remember client. This mechanism is nothing but session management by using session api.

Note: Session information will be stored in one of following possibilities

1. Inside a File
2. Inside a database
3. Inside Cache

The most straight forward approach is to use django.contrib.sessions application to store session information in a Django Model/database.

The Model Name is: django.contrib.sessions.models.Session

Note: To use this approach compulsory the following applicaiton should be configured inside INSTALLED_APPS list of settings.py file.
django.contrib.sessions

If it is not there then we can add, but we have to synchronize database
python manage.py syncdb

Note:

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

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

Useful Methods for Session Management:

1) **request.session['key'] = value**
   To Add Data to the Session.

2) **value = request.session['key']**
   To get Data from the Session

3) **request.session.set_expiry(seconds)**
   Sets the expiry Time for the Session.

4) **request.session.get_expiry_age()**
   Returns the expiry age in seconds(the number of seconds until this session expire)

5) **request.session.get_expiry_date()**
   Returns the data on which this session will expire

Note: Before using session object in our application, compulsory we have to migrate. Otherwise we will get the following error.
no such table: django_session

Session Demo1:

views.py:

```
1)  from django.shortcuts import render
2)
3)  # Create your views here.
4)  def page_count_view(request):
5)      count=request.session.get('count',0)
6)      newcount=count+1
7)      request.session['count']=newcount
8)      print(request.session.get_expiry_age())
9)      print(request.session.get_expiry_date())
10)     return render(request,'testapp/pagecount.html',{'count':newcount})
```

pagecount.html:

```
1)  <!DOCTYPE html>
2)  <html lang="en" dir="ltr">
3)    <head>
4)      <meta charset="utf-8">
5)      <title></title>
6)      <style >
7)      span{
8)        font-size: 300px;
9)      }
10)
11)     </style>
12)   </head>
13)   <body>
14)     <h1>The Page Count:<span>{{count}}</span></h1>
15)   </body>
16) </html>
```

Session Demo Application-2

forms.py

```
1)  from django import forms
2)  class NameForm(forms.Form):
3)      name=forms.CharField()
4)
5)  class AgeForm(forms.Form):
6)      age=forms.IntegerField()
```

```
7)
8)  class GFForm(forms.Form):
9)      gf=forms.CharField()
```

views.py

```
1)  from django.shortcuts import render
2)  from testapp.forms import *
3)
4)  # Create your views here.
5)  def name_view(request):
6)      form=NameForm()
7)      return render(request,'testapp/name.html',{'form':form})
8)
9)  def age_view(request):
10)     name=request.GET['name']
11)     request.session['name']=name
12)     form=AgeForm()
13)     return render(request,'testapp/age.html',{'form':form})
14)
15) def gf_view(request):
16)     age=request.GET['age']
17)     request.session['age']=age
18)     form=GFForm()
19)     return render(request,'testapp/gf.html',{'form':form})
20)
21) def results_view(request):
22)     gf=request.GET['gf']
23)     request.session['gf']=gf
24)     return render(request,'testapp/results.html')
```

name.html

```
1)  <!DOCTYPE html>
2)  <html lang="en" dir="ltr">
3)    <head>
4)      <meta charset="utf-8">
5)      <title></title>
6)    </head>
7)    <body>
8)      <h1>Name Registration Form</h1><hr>
9)      <form  action="/age" >
10)       {{form}}
11)       {%csrf_token%}<br><br>
12)       <input type="submit" name="" value="Submit Name">
```

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

age.html

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h1>Age Registration Form</h1><hr>
    <form  action="/gf" >
      {{form}}
      {%csrf_token%}<br><br>
      <input type="submit" name="" value="Submit Age">
    </form>
  </body>
</html>
```

gf.html

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h1>Girl Friend Registration Form</h1><hr>
    <form  action="/results" >
      {{form}}
      {%csrf_token%}<br><br>
      <input type="submit" name="" value="Submit GF Info">
    </form>
  </body>
</html>
```

results.html

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

```
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    {%if request.session %}
    <h1>Thanks for providing information..Plz confirm once</h1>
    <ul>
      {%for key,value in request.session.items %}
      <li> <h2>{{key|upper}}:    {{value|title}}</h2> </li>
      {%endfor%}
    </ul>
    {%else%}
    <p>No Information available</p>
    {%endif%}
</body>
</html>
```

Session Demo3 Shopping Cart Application

settings.py

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

forms.py

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

views.py

```
from django.shortcuts import render
from testapp.forms import *

# Create your views here.
def add_item_view(request):
    form=AddItemForm()
```

```
    if request.method=='POST':
        name=request.POST['name']
        quantity=request.POST['quantity']
        request.session[name]=quantity
    return render(request,'testapp/additem.html',{'form':form})

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

additem.html

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">
  </head>
  <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><br><br><br>
      <a class="btn btn-primary btn-lg" href="/display" role="button">Display ITEMS</a>
    </div>
  </body>
</html>
```

displayitems.html

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h1>Your Shopping Cart Information:</h1>
```

```
{%if request.session %}
<table border=2>
  <thead>
    <th>Item Name</th>
    <th>Quantity</th>
  </thead>

  {% for key,value in request.session.items %}
  <tr>
    <td>{{key}}</td>
    <td>{{value}}</td>
  </tr>
  {% endfor %}
  </table>
 {%else%}
  <p>No Items in your shopping cart</p>
 {%endif%}
</body>
</html>
```

Note: The default max_age of the session is 14 days. But based on our requirement we can set our own expiry time.

```
request.session.set_expiry(time in seconds)
```

Important Methods related to Session:

1) set_expiry(time_in_seconds)

- If we are not performing any operation on the session specified amount of time (max inactive interval) then session will be expired automatically.

```
request.session.set_expiry(120)
```

- If we set 0 as argument, then the session will expire once we closed browser.

2) get_expiry_age()
3) get_expiry_date()

Note: We can observer that 0 value and 120 values are perfectly working in our application

```
if request.method=='POST':
    name=request.POST['name']
    quantity=request.POST['quantity']
    request.session[name]=quantity
    request.session.set_expiry(0)
```

How to Delete Session Data:

del request.session[sessionkey]

```
for key in request.session.keys():
  del request.session[key]
```

In settings.py File

SESSION_SAVE_EVERY_REQUEST=True

OR

In views.py

request.session.modified = True

Browser Length Sessions and Persistent Sessions

- **If the session information stored inside browsers cache such type of sessions are called browser length sessions.**
- **If the session information stored persistently inside file/database/cache, such type of sessions are called Persistent sessions.**

Note: By default sessions are persistent sessions.

User Authentication and Authorization

Authentication: The process of validating user is called authentication.

Authorization: The process of validating access permissions of user is called authorization.

Generally our web pages can be accessed by any person without having any restrictions. But some times our business requirement is to access a web page compulsory we have to register and login.Then only end user can able to access our page. To fulfill such of requirements we should go for Django authentication and authorization module. (auth application)

Django provides the following 2 in built applications for user authentication.

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

auth application is authentication application of Django.
This auth application internally uses contenttypes application to track models installed in our database.

Note: To use Django in built authentication facility, compulsory these 2 applications should be in INSTALLED_APPS list. But from Django 1.10 onwards automatically these are available and we are not required to add explicitly.

Django uses PBKDF2_Sha256 algorithm to encrypt passwords and hence passwords won't be stored in plain text form and we can expect more security. Even superuser also cannot see any user's password.

Based on our requirement, we can use more secure hashing algorithms also like bcrypt and argon2. We can install with pip as follows.

pip install bcrypt
pip install django[argon2]

More secured algorithm is argon2 and then bcrypt followed PBKDF2.

In settings.py we have to configure password hashers as follows.

```
1) PASSWORD_HASHERS=[
2) 'django.contrib.auth.hashers.Argon2PasswordHasher',
3) 'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
4) 'django.contrib.auth.hashers.BCryptPasswordHasher',
5) 'django.contrib.auth.hashers.PBKDF2PasswordHasher',
6) 'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
7) ]
```

Django will always consider from first to last. ie order is important.

Just like templates and static folder, we have to create media folder also.

Difference between Static and Media Folders:

- Static folder contains images, CSS files etc which are provided by application to the end user.
- But media folder contains the resources like images provided by end user to the application (like profile image etc)

How to Configure Media Folder in settings.py File:

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

MEDIA_URL = '/media/'

from django.contrib.auth.decorators import login_required

```
@login_required
def java_exams_view(request):
   return render(request,'testapp/java.html')
```

If we use @login_required decorator for any view function,then auth application will check whether user login or not. If the user not login then the control will be forwarded to login page.

http://127.0.0.1:8000/accounts/login/?next=/java/

We have to configure auth application url patterns in urls.py file.

from django.conf.urls import url,include

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

In our project auth application urls also included.

TemplateDoesNotExist at /accounts/login/registration/login.html

login link of auth application: /accounts/login/
logout link of auth application: /accounts/logout

After logout then Django default logout page will be displayed. Instead of this default target page we can configure our own target page inside settings.py as follows.

LOGOUT_REDIRECT_URL='/'

If we click login link explicitly and after login by default the control will goes to http://127.0.0.1:8000/accounts/profile/
But we can configure our own target page after login inside settings.py as follows.

LOGIN_REDIRECT_URL='/'

Configure signup form:

forms.py

```python
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

```python
def signup_view(request):
    form=SignUpForm()
    if request.method=='POST':
        form=SignUpForm(request.POST)
        user=form.save()
        user.set_password(user.password)
        user.save()
        return HttpResponseRedirect('/accounts/login')
    return render(request,'testapp/signup.html',{'form':form})
```

Future Assignments:

1) How to customize our own login form
2) How to use auth application provided default signup form
3) Social Login

Class Based Views (CBVs):

There are 2 types of views

1) Function Based Views
2) Class Based Views

Note:

1) Class Based Views introduced in Django 1.3 Version to implement Generic Views.
2) When compared with Function Based views,class Based views are very easy to use.Hence Class Based Views are Most frequently used views in real time.
3) Interanally Class Based Views will be converted into Function Based Views.Hence Class Based Views are simply acts as wrappers to the Function based views to hide complexity.
4) Function Based views are more powerful when compared with Class Based Views.

Q) Explain the Sceneario where we should use Function based Views only and we cannot use Class based Views?

- For simple operations like listing of all records or display details of a particular record then we should go for Class Based Views.

- For complex operations like handling multiple forms simulataneously then we should go for Function Based Views.

  Eg: KFC

HelloWorld Application By using ClassBasedViews

views.py

```
1)  from django.views.generic import View
2)  from django.http import HttpResponse
3)
4)  # Create your views here.
5)  class HelloWorldView(View):
6)      def get(self,request):
7)          return HttpResponse('<h1>This is from ClassBasedView</h1>')
```

urls.py

```
1)  from testapp import views
2)
3)  urlpatterns = [
4)      ...
5)      url(r'^$', views.HelloWorldView.as_view()),
6)  ]
```

Note:

1) While defining Class Based Views we have to extend View class.
2) To provide response to GET request, Django will always call get() method.Hence we have to override this method to provide response to the GET request.Similarly for other HTTP Methods also like POST,HEAD etc
3) While defining url pattern we have to use as_view() method.

Template Based Application by using Class Based Views:

```
class TemplateCBV(TemplateView)
   template_name = 'home.html'
```

How to send Context Paramters:

```
class TemplateCBV(TemplateView)
   template_name = 'home.html'

def get_context_data(self,**kwargs)
       context = super().get_context_data(**kwargs)
       context['name'] = 'durga'
```

```
        context['age'] = 30
        return context
```

In template file we can access context parameters as follows
{{name}}
{{age}}

ListView:

We can use ListView class to list out all records from database table(model).
It is alternative to `ModelClassName.objects.all()`

models.py

```
1)  from django.db import models
2)
3)  # Create your models here.
4)  class Book(models.Model):
5)      title=models.CharField(max_length=300)
6)      author=models.CharField(max_length=30)
7)      pages=models.IntegerField()
8)      price=models.FloatField()
```

views.py

```
1)  from testapp.models import Book
2)  from django.views.generic import ListView
3)
4)  # Create your views here.
5)  class BookListView(ListView):
6)      model=Book
```

How to create template file for ListView:

Django will identify template automatically and we are not required to configure anywhere. But Django will always search for template file with the name `modelclassname_list.html` like `book_list.html`

Django will always search for template file in the following location.
projectname/appname/templates/appname/
Eg: cbvproject5/testapp/templates/test/book_list.html

Note: bydefault django will provide context object to the template file with the name: modelclassname_list
Eg: book_list

book_list.html

```
1)  <!DOCTYPE html>
2)  <html lang="en" dir="ltr">
3)    <head>
4)      <meta charset="utf-8">
5)      <title></title>
6)    </head>
7)    <body>
8)      <h1>All Books Information</h1><hr>
9)      {% for book in book_list%}
10)     <ul>
11)       <li> <strong>Title</strong>:{{book.title}}</li>
12)       <li> <strong>Author</strong>:{{book.author}}</li>
13)       <li> <strong>Pages</strong>:{{book.pages}}</li>
14)       <li> <strong>Price</strong>:{{book.price}}</li>
15)     </ul>
16)     <hr>
17)     {%endfor%}
18)   </body>
19) </html>
```

How to provide our own Context Object Name:

The default context object name is: modelclassname_list
But we can customize this name based on our requirement as follows

```
class BookListView(ListView):
    context_object_name = 'books'
    model = Book
```

How to configure our own Template File at Project Level:

ofcourse this approach is not recommended

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

Note: Even if we are not specifying template_name variable,still django can recognize project level template file. But name should be modelclassname_list.html

DetailView:

We can use ListView to list of all records present in the database table.
But to get details of a particular record, we should go for DetailView.

models.py

```
1)  from django.db import models
2)
3)  class Company(models.Model):
4)      name=models.CharField(max_length=128)
5)      location=models.CharField(max_length=64)
6)      ceo=models.CharField(max_length=64)
```

admin.py

```
1)  from django.contrib import admin
2)  from testapp.models import Company
3)
4)  # Register your models here.
5)  class CompanyAdmin(admin.ModelAdmin):
6)      list_display=['name','location','ceo']
7)
8)  admin.site.register(Company,CompanyAdmin)
```

views.py

```
1)  from django.shortcuts import render
2)  from testapp.models import Company
3)  from django.views.generic import ListView,DetailView
4)
5)  # Create your views here.
6)  class CompanyListView(ListView):
7)      model=Company
8)      #default template_name is company_list.html
9)      #defult context_object_name is company_list
10)
11) class CompanyDetailView(DetailView):
12)     model=Company
13)     #default template_name is company_detail.html
14)     #defult context_object_name is company or object
```

base.html

```
1)  <!DOCTYPE html>
2)  <html lang="en" dir="ltr">
3)    <head>
```

```
    <meta charset="utf-8">
    <title></title>
    <!-- Latest compiled and minified CSS -->
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" 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><hr>
        <ol>
          {%for company in company_list%}
       <h2><li> <a href="/{{company.id}}">{{company.name|upper}}</a> </li></h2>
          {%endfor%}
        </ol>
      {%endblock %}
```

company_detail.html

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

urls.py

```
from django.conf.urls import url
```

```
2) from django.contrib import admin
3) from testapp import views
4)
5) urlpatterns = [
6)     url(r'^admin/', admin.site.urls),
7)     url(r'^companies/', views.CompanyListView.as_view()),
8)     url(r'^(?P<pk>\d+)/$', views.CompanyDetailView.as_view()),
9) ]
```

Adding Employee Information also

models.py

```
1) from django.db import models
2)
3) # Create your models here.
4) class Company(models.Model):
5)     name=models.CharField(max_length=128)
6)     location=models.CharField(max_length=64)
7)     ceo=models.CharField(max_length=64)
8)
9)     def __str__(self):
10)        return self.name
11)
12) class Employee(models.Model):
13)     eno=models.IntegerField()
14)     name=models.CharField(max_length=128)
15)     salary=models.FloatField()
16)     company=models.ForeignKey(Company,related_name='employees')
```

company_detail.html

```
1) <!DOCTYPE html>
2) {%extends 'testapp/base.html'%}
3)         {%block body_block%}
4)           <h1>Company Information</h1><hr>
5)           <ol>
6)             <h2><li>Company Name: {{company.name|upper}}</li></h2>
7)             <h2><li>Company Location: {{company.location|title}}</li></h2>
8)             <h2><li>Company CEO: {{company.ceo|title}}</li></h2>
9)             <h2>Employee Information</h2>
10)            {%for emp in company.employees.all%}
11)             <ul>
12)               <li>Employee Number:{{emp.eno}}</li>
```

```
13)          <li>Employee Name:{{emp.name}}</li>
14)          <li>Employee Salary:{{emp.salary}}</li> <hr>
15)         </ul>
16)         {%endfor%}
17)       </ol>
18) {%endblock %}
```

Django CRUD Operations

1) C → Create (Insert Query)
2) R → Retrieve (Select Query)
3) U → Update (Update Query)
4) D → Delete (Delete Query)

For any web application,it is very common requirement to perform CRUD operations.

Case Study: BookMyshow Application

1) Add New Movie Information (Create)
2) Show Movie Information (Retrieve)
3) Update New timings for existing Movie (Update)
4) Delete old Movie Information (Delete)

By using the following ClassBased Views we can perform CRUD operations very easily.
ListView, DetailView → Retrieve Operation
CreateView → Create Operation (Insert Data)
UpdateView → Update Operation
DeleteView → Delete Operation

CreateView Class:

We can use this CreateView class to insert data into our models.

views.py

```
1) class CompanyCreateView(CreateView):
2)     model=Company
```

urls.py

```
1) urlpatterns = [
2)     url(r'^admin/', admin.site.urls),
3)     url(r'^companies/', views.CompanyListView.as_view()),
4)     url(r'^(?P<pk>\d+)/$', views.CompanyDetailView.as_view(),name='detail'),
5)     url(r'^create/', views.CompanyCreateView.as_view(),name='create'),
6) ]
```

If we send Request:

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

We can solve this problem by defining fields attribute in CreateView class

```
class CompanyCreateView(CreateView):
    model = Company
    fields = ('name','location','ceo')
```

If we send Request:
TemplateDoesNotExist at /create/
testapp/company_form.html

By default CreateView class will always search for template file named with modelclassname_form.html
Eg: company_form.html

company_form.html

```
1)  <!DOCTYPE html>
2)  {%extends 'testapp/base.html'%}
3)        {%block body_block%}
4)          <h1>Company Create Form</h1><hr>
5)          <form  method="post">
6)                {{form.as_p}}
7)               {%csrf_token%}
8)           <input type="submit" class='btn btn-primary btn-lg' value="Insert Record">
9)         </form>
10)  {%endblock %}
```

If we fill form and submit:

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

How to define get_absolute_url() in Model class:

```
1)  from django.db import models
2)  from django.core.urlresolvers import reverse
3)
4)  # Create your models here.
```

```
5)  class Company(models.Model):
6)      name=models.CharField(max_length=128)
7)      location=models.CharField(max_length=64)
8)      ceo=models.CharField(max_length=64)
9)      def __str__(self):
10)         return self.name
11)
12)     def get_absolute_url(self):
13)         return reverse('detail',kwargs={'pk':self.pk})
```

UpdateView:

We can use UpdateView to update existing record.

views.py

```
1)  class CompanyUpdateView(UpdateView):
2)      model=Company
3)      fields=('name','ceo')
```

urls.py

Define URL for this updateview

http://localhost:8000/update/7

```
1)  from django.conf.urls import url
2)  from django.contrib import admin
3)  from testapp import views
4)
5)  urlpatterns = [
6)      url(r'^admin/', admin.site.urls),
7)      url(r'^companies/', views.CompanyListView.as_view()),
8)      url(r'^(?P<pk>\d+)/$', views.CompanyDetailView.as_view(),name='detail'),
9)      url(r'^create/', views.CompanyCreateView.as_view(),name='create'),
10)     url(r'^update/(?P<pk>\d+)/$', views.CompanyUpdateView.as_view(),name='upd
    ate'),
11) ]
```

Add Update Button in Details Page

company_detail.html

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

```
4)        <h1>Company Information</h1><hr>
5)        <ol>
6)          <h2><li>Company Name: {{company.name|upper}}</li></h2>
7)          <h2><li>Company Location: {{company.location|title}}</li></h2>
8)          <h2><li>Company CEO: {{company.ceo|title}}</li></h2>
9)        </ol>
10)       <a href="/update/{{company.id}}" class='btn btn-warning'>Update</a>
11)     {%endblock %}
```

DeleteView:

We can use DeleteView to delete records

views.py

```
1)  from django.core.urlresolvers import reverse_lazy
2)  class CompanyDeleteView(DeleteView):
3)      model=Company
4)      success_url=reverse_lazy('/companies')
```

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

urls.py

url(r'^delete/(?P<pk>\d+)/$', views.CompanyDeleteView.as_view(),name='delete')

http://localhost:8000/delete/7

TemplateDoesNotExist at /delete/7/
testapp/company_confirm_delete.html

If we are trying to delete, then DeleteView will provide confirmation template.
The default template name is model_confirm_delete.html

Eg: company_confirm_delete.html
We have to provide this template file.

company_confirm_delete.html

```
1)  <!DOCTYPE html>
2)  {%extends 'testapp/base.html'%}
3)        {%block body_block%}
4)          <h1>Delete {{company.name}} ???</h1><hr>
5)          <form method="post">
6)            {%csrf_token%}
7)            <input type="submit" class='btn btn-danger' value="Delete Record">
```

```
8)          <a href="/{{company.id}}" class='btn btn-success'>Cancel</a>
9)        </form>
10)     {%endblock %}
```

To Place Delete Button on Details Page

```
1)  <!DOCTYPE html>
2)  {%extends 'testapp/base.html'%}
3)      {%block body_block%}
4)        <h1>Company Information</h1><hr>
5)        <ol>
6)          <h2><li>Company Name: {{company.name|upper}}</li></h2>
7)          <h2><li>Company Location: {{company.location|title}}</li></h2>
8)          <h2><li>Company CEO: {{company.ceo|title}}</li></h2>
9)        </ol>
10)       <a href="/update/{{company.id}}" class='btn btn-warning'>Update</a>
11)       <a href="/delete/{{company.id}}" class='btn btn-danger'>Delete</a>
12)
13)     {%endblock %}
```

Python Class Based Views Complete Example (cbvproject7)

models.py

```
1)  from django.db import models
2)  from django.core.urlresolvers import reverse
3)
4)  # Create your models here.
5)  class Beer(models.Model):
6)      name=models.CharField(max_length=128)
7)      taste=models.CharField(max_length=128)
8)      color=models.CharField(max_length=128)
9)      price=models.FloatField()
10)     def __str__(self):
11)         return self.name
12)
13)     def get_absolute_url(self):
14)         return reverse('detail',kwargs={'pk':self.pk})
```

admin.py

```
1)  from django.contrib import admin
2)  from testapp.models import Company
3)
```

```
4)  # Register your models here.
5)  class CompanyAdmin(admin.ModelAdmin):
6)      list_display=['name','location','ceo']
7)
8)  admin.site.register(Company,CompanyAdmin)
```

views.py

```
1)  from django.shortcuts import render
2)  from testapp.models import Company
3)  from django.core.urlresolvers import reverse_lazy
4)  from django.views.generic import ListView,DetailView,CreateView,UpdateView,De
    leteView
5)
6)  # Create your views here.
7)  class CompanyListView(ListView):
8)      model=Company
9)      #default template_name is company_list.html
10)     #defult context_object_name is company_list
11) class CompanyDetailView(DetailView):
12)     model=Company
13)     #default template_name is company_detail.html
14)     #defult context_object_name is company or object
15) class CompanyCreateView(CreateView):
16)     model=Company
17)     fields=('name','location','ceo')
18)
19) class CompanyUpdateView(UpdateView):
20)     model=Company
21)     fields=('name','ceo')
22)
23) class CompanyDeleteView(DeleteView):
24)     model=Company
25)     success_url=reverse_lazy('companies')
```

urls.py

```
1)  from django.conf.urls import url
2)  from django.contrib import admin
3)  from testapp import views
4)
5)  urlpatterns = [
6)      url(r'^admin/', admin.site.urls),
7)      url(r'^companies/', views.CompanyListView.as_view(),name='companies'),
8)      url(r'^(?P<pk>\d+)/$', views.CompanyDetailView.as_view(),name='detail'),
```

```
    url(r'^create/', views.CompanyCreateView.as_view(),name='create'),
    url(r'^update/(?P<pk>\d+)/$', views.CompanyUpdateView.as_view(),name='update'),
    url(r'^delete/(?P<pk>\d+)/$', views.CompanyDeleteView.as_view(),name='delete'),
]
```

base.html

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
    <!-- Latest compiled and minified CSS -->
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" integrity="sha384-
BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" 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><hr>
          <ol>
            {%for company in company_list%}
            <h2><li> <a href="/{{company.id}}">{{company.name|upper}}</a> </li></h2>
            {%endfor%}
          </ol>
        {%endblock %}
```

company_detail.html

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

```
4)          <h1>Company Information</h1><hr>
5)          <ol>
6)            <h2><li>Company Name: {{company.name|upper}}</li></h2>
7)            <h2><li>Company Location: {{company.location|title}}</li></h2>
8)            <h2><li>Company CEO: {{company.ceo|title}}</li></h2>
9)          </ol>
10)         <a href="/update/{{company.id}}" class='btn btn-warning'>Update</a>
11)         <a href="/delete/{{company.id}}" class='btn btn-danger'>Delete</a>
12)
13)       {%endblock %}
```

company_form.html

```
1)  <!DOCTYPE html>
2)  {%extends 'testapp/base.html'%}
3)        {%block body_block%}
4)          <h1>Company Create Form</h1><hr>
5)          <form  method="post">
6)            {{form.as_p}}
7)            {%csrf_token%}
8)            <input type="submit" class='btn btn-primary btn-lg' value="Insert Record">
9)          </form>
10)       {%endblock %}
```

company_confirm_delete.html

```
1)  <!DOCTYPE html>
2)  {%extends 'testapp/base.html'%}
3)        {%block body_block%}
4)          <h1>Delete {{company.name}} ???</h1><hr>
5)          <form method="post">
6)            {%csrf_token%}
7)            <input type="submit" class='btn btn-danger' value="Delete Record">
8)            <a href="/{{company.id}}" class='btn btn-success'>Cancel</a>
9)          </form>
10)       {%endblock %}
```

Python Class Based Views Complete Example (cbvfinalproject)

models.py

```
1)  from django.db import models
2)  from django.core.urlresolvers import reverse
3)
```

```
4)  # Create your models here.
5)  class Beer(models.Model):
6)      name=models.CharField(max_length=128)
7)      taste=models.CharField(max_length=128)
8)      color=models.CharField(max_length=128)
9)      price=models.FloatField()
10)
11)     def __str__(self):
12)         return self.name
13)
14)     def get_absolute_url(self):
15)         return reverse('detail',kwargs={'pk':self.pk})
```

admin.py

```
1)  from django.db import models
2)  from django.core.urlresolvers import reverse
3)
4)  # Create your models here.
5)  class Beer(models.Model):
6)      name=models.CharField(max_length=128)
7)      taste=models.CharField(max_length=128)
8)      color=models.CharField(max_length=128)
9)      price=models.FloatField()
10)
11)     def __str__(self):
12)         return self.name
13)
14)     def get_absolute_url(self):
15)         return reverse('detail',kwargs={'pk':self.pk})
```

views.py

```
1)  from django.shortcuts import render
2)  from testapp.models import Beer
3)  from django.core.urlresolvers import reverse_lazy
4)  from django.views.generic import ListView,DetailView,CreateView,UpdateView,De
    leteView
5)
6)  # Create your views here.
7)  class BeerListView(ListView):
8)      model=Beer
9)
10) class BeerDetailView(DetailView):
11)     model=Beer
```

```
12)
13) class BeerCreateView(CreateView):
14)     model=Beer
15)     #fields=('name','taste','color','price')
16)     fields='__all__'
17) class BeerUpdateView(UpdateView):
18)     model=Beer
19)     fields=('taste','color','price')
20) class BeerDeleteView(DeleteView):
21)     model=Beer
22)     success_url=reverse_lazy('home')
```

urls.py

```
1)  from django.conf.urls import url
2)  from django.contrib import admin
3)  from testapp import views
4)
5)  urlpatterns = [
6)      url(r'^admin/', admin.site.urls),
7)      url(r'^$', views.BeerListView.as_view(),name='home'),
8)      url(r'^(?P<pk>\d+)/$', views.BeerDetailView.as_view(),name='detail'),
9)      url(r'^create/', views.BeerCreateView.as_view()),
10)     url(r'^update/(?P<pk>\d+)/$',  views.BeerUpdateView.as_view()),
11)     url(r'^delete/(?P<pk>\d+)/$',  views.BeerDeleteView.as_view()),
12) ]
```

base.html

```
1)  <!DOCTYPE html>
2)  <html lang="en" dir="ltr">
3)    <head>
4)      <meta charset="utf-8">
5)      <title></title>
6)      <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7
    /css/bootstrap.min.css" integrity="sha384-
    BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" c
    rossorigin="anonymous">
7)    </head>
8)    <body>
9)      <div class="container">
10)       {%block body_block%}
11)       {%endblock%}
12)     </div>
13)   </body>
```

```
14) </html>
```

beer_list.html

```
1)  <!DOCTYPE html>
2)  {% extends 'testapp/base.html'%}
3)      {%block body_block%}
4)        <h1>Beer Information Dash Board</h1><hr>
5)        <table border='2'>
6)          <thead>
7)            <th>Beer Name</th>
8)            <th>Details</th>
9)            <th>Update</th>
10)           <th>Delete</th>
11)         </thead>
12)         {%for  beer in beer_list %}
13)         <tr>
14)           <td>{{beer.name|title}}</td>
15)           <td> <a href="/{{beer.id}}">Details</a> </td>
16)           <td><a href="/update/{{beer.id}}">Update</a></td>
17)           <td><a href="/delete/{{beer.id}}">Delete</a></td>
18)         </tr>
19)         {%endfor%}
20)       </table><br><br><br>
21)       <a href="/create" class='btn btn-primary btn-
    lg'>Do You Want to Insert New Beer</a>
22)             {%endblock%}
```

beer_detail.html

```
1)  <!DOCTYPE html>
2)  {% extends 'testapp/base.html'%}
3)      {%block body_block%}
4)        <div class="jumbotron">
5)          <h1>Beer Details...</h1><hr>
6)          <ul>
7)            <li>Beer Name: {{beer.name}}</li>
8)            <li>Beer Taste: {{beer.taste}}</li>
9)            <li>Beer Color: {{beer.color}}</li>
10)           <li>Beer Price: {{beer.price}}</li>
11)         </ul>
12)       </div>
13)     {%endblock%}
```

beer_form.html


```
1)  <!DOCTYPE html>
2)  {% extends 'testapp/base.html'%}
3)      {%block body_block%}
4)        <h1>Add New Beer Here</h1><hr>
5)        <form method="post">
6)          {{form.as_p}}
7)          {%csrf_token%}
8)          <input type="submit" class='btn btn-primary btn-
    lg' name="" value="Insert/Update">
9)        </form>
10)     {%endblock%}
```

beer_confirm_delte.html

```
1)  <!DOCTYPE html>
2)  {% extends 'testapp/base.html'%}
3)      {%block body_block%}
4)        <h1>Do You Want to Delete {{beer.name}} Record</h1><hr>
5)        <form method="post">
6)          {%csrf_token%}
7)          <input type="submit" class='btn btn-lg btn-danger' value="DELETE">
8)          <a href="/" class='btn btn-lg btn-success'>CANCEL</a>
9)        </form>
10)     {%endblock%}
```

CRUD Operations by using Function Based Views (FBVs)

1) Start Project
2) Start App
3) Templates → testapp → *.html
4) Add Application and Templates Path to settings.py
5) Create Employee Model Class
6) Make Migrations and Migrate
7) Register this Model inside admin.py and Create Super User
8) Execute populate Script

views.py

```
1)  from django.shortcuts import render,redirect
2)  from testapp.forms import EmployeeForm
3)  from testapp.models import Employee
4)
5)  # Create your views here.
6)  def show_view(request):
7)      employees=Employee.objects.all()
```

```
8)      return render(request,'testapp/index.html',{'employees':employees})
9)
10) def insert_view(request):
11)     form=EmployeeForm()
12)     if request.method=='POST':
13)         form=EmployeeForm(request.POST)
14)         if form.is_valid():
15)             form.save()
16)         return redirect('/')
17)     return render(request,'testapp/insert.html',{'form':form})
```

models.py

```
1)  from django.db import models
2)
3)  # Create your models here.
4)  class Employee(models.Model):
5)      eno=models.IntegerField()
6)      ename=models.CharField(max_length=64)
7)      esal=models.FloatField()
8)      eaddr=models.CharField(max_length=256)
```

forms.py

```
1)  from django import forms
2)  from testapp.models import Employee
3)  class EmployeeForm(forms.ModelForm):
4)      class Meta:
5)          model=Employee
6)          fields='__all__'
```

populate.py

```
1)  import os
2)  os.environ.setdefault('DJANGO_SETTINGS_MODULE','fbvproject1.settings')
3)  import Django
4)  django.setup()
5)
6)  from testapp.models import *
7)  from faker import Faker
8)  from random import *
9)  faker=Faker()
10) def populate(n):
11)     for i in range(n):
12)         feno=randint(1001,9999)
```

```
13)       fename=faker.name()
14)       fesal=randint(10000,20000)
15)       feaddr=faker.city()
16)       emp_record=Employee.objects.get_or_create(eno=feno,ename=fename,esal=f
    esal,eaddr=feaddr)
17) populate(10)
```

views.py (Delete & Update)

```
1)  def delete_view(request,id):
2)     employee=Employee.objects.get(id=id)
3)     employee.delete()
4)     return redirect('/')
5)
6)  def update_view(request,id):
7)     employee=Employee.objects.get(id=id)
8)     if request.method=='POST':
9)        form=EmployeeForm(request.POST,instance=employee)
10)       if form.is_valid():
11)          form.save()
12)          return redirect('/')
13)    return render(request,'testapp/update.html',{'employee':employee})
```

Note: In the following line if we are not using instance then a new record will be created.

form = EmployeeForm(request.POST,instance = employee)

form = EmployeeForm(request.POST) → New Record will be created
form = EmployeeForm(request.POST, instance = employee) → Existing Record will be updated

Complete Application (fbvproject1)

models.py

```
1)  from django.db import models
2)
3)  # Create your models here.
4)  class Employee(models.Model):
5)     eno=models.IntegerField()
6)     ename=models.CharField(max_length=64)
7)     esal=models.FloatField()
8)     eaddr=models.CharField(max_length=256)
```

admin.py

```
1)  from django.db import models
2)
3)  # Create your models here.
4)  class Employee(models.Model):
5)      eno=models.IntegerField()
6)      ename=models.CharField(max_length=64)
7)      esal=models.FloatField()
8)      eaddr=models.CharField(max_length=256)
```

views.py

```
1)  from django.shortcuts import render,redirect
2)  from testapp.forms import EmployeeForm
3)  from testapp.models import Employee
4)
5)  # Create your views here.
6)  def show_view(request):
7)      employees=Employee.objects.all()
8)      return render(request,'testapp/index.html',{'employees':employees})
9)
10) def insert_view(request):
11)     form=EmployeeForm()
12)     if request.method=='POST':
13)         form=EmployeeForm(request.POST)
14)         if form.is_valid():
15)             form.save()
16)         return redirect('/')
17)     return render(request,'testapp/insert.html',{'form':form})
18)
19) def delete_view(request,id):
20)     employee=Employee.objects.get(id=id)
21)     employee.delete()
22)     return redirect('/')
23)
24) def update_view(request,id):
25)     employee=Employee.objects.get(id=id)
26)     if request.method=='POST':
27)         form=EmployeeForm(request.POST,instance=employee)
28)         if form.is_valid():
29)             form.save()
30)             return redirect('/')
31)     return render(request,'testapp/update.html',{'employee':employee})
```

urls.py

```
1)  from django.conf.urls import url
2)  from django.contrib import admin
3)  from testapp import views
4)
5)  urlpatterns = [
6)      url(r'^admin/', admin.site.urls),
7)      url(r'^$', views.show_view),
8)      url(r'^insert/', views.insert_view),
9)      url(r'^delete/(?P<id>\d+)/$',  views.delete_view),
10)     url(r'^update/(?P<id>\d+)/$',  views.update_view),
11) ]
```

base.html

```
1)  <!DOCTYPE html>
2)  <html lang="en" dir="ltr">
3)    <head>
4)      <meta charset="utf-8">
5)      <title></title>
6)      <!-- Latest compiled and minified CSS -->
7)  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/cs
    s/bootstrap.min.css" integrity="sha384-
8)  BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" c
    rossorigin="anonymous">
9)
10)   </head>
11)   <body>
12)     <div class="container" align='center'>
13)       {%block body_block%}
14)       {%endblock%}
15)     </div>
16)   </body>
17) </html>
```

index.html

```
1)  <!DOCTYPE html>
2)  {% extends 'testapp/base.html'%}
3)       {%block body_block%}
4)       <h1>Employee Information Dash Board</h1><hr>
5)       <table border='2'>
6)         <thead>
7)           <th>Employee Number</th>
```

```
8)          <th>Employee Name</th>
9)          <th>Employee Salary</th>
10)         <th>Employee Address</th>
11)         <th>Actions</th>
12)       </thead>
13)       {%for  emp in employees %}
14)       <tr>
15)         <td>{{emp.eno}}</td>
16)         <td>{{emp.ename}}</td>
17)         <td>{{emp.esal}}</td>
18)         <td>{{emp.eaddr}}</td>
19)         <td>
20)           <a href="/update/{{emp.id}}">Update</a>
21)           <a href="/delete/{{emp.id}}">Delete</a>
22)         </td>
23)       </tr>
24)       {%endfor%}
25)     </table><br><br><br>
26)     <a href="/insert" class='btn btn-primary btn-
    lg'>Do You Want to Insert New Employee</a>
27)
28)     {%endblock%}
```

insert.html

```
1)  <!DOCTYPE html>
2)  {% extends 'testapp/base.html'%}
3)       {%block body_block%}
4)       <h1>Employee Insert Form</h1><hr>
5)       <form method="post">
6)         <table border='1'>
7)           {{form}}
8)         </table>
9)         {%csrf_token%}
10)        <br>
11)        <input type="submit" class='btn btn-success btn-lg' value="Insert Record">
12)      </form>
13)      {%endblock%}
```

update.html

```
1)  <!DOCTYPE html>
2)  {% extends 'testapp/base.html'%}
3)       {%block body_block%}
4)       <h1>Employee Update Form</h1><hr>
```

```
5)      <form  method="post">
6)        {%csrf_token%}
7)        Employee Number: <input type="text" name="eno" value="{{employee.eno}}"
    ><p></p>
8)        Employee Name: <input type="text" name="ename" value="{{employee.enam
    e}}"><p></p>
9)        Employee Salary: <input type="text" name="esal" value="{{employee.esal}}">
    <p></p>
10)       Employee Address: <input type="text" name="eaddr" value="{{employee.ead
    dr}}"><p></p>
11)       <input type="submit" class='btn btn-warning btn-lg' value="Update Record">
12)     </form>
13)     {%endblock%}
```

How to use django form for update:

In forms.py, create the form with instance employee as
"form = EmployeeForm(instance = employee) "
Send the form object intead of employee object in render function of update_views as
"{'form':form} "
{{form.as_p}}

Differences between CBVs and FBVs

CBVs	FBVs
1) CBVs can be easily extended	1) FBVs cannot extended easily
2) CBVs promotes Reusability of the Code	2) FBVs cannot promote Reusability of the Code
3) CBVs can use Object Oriented Techniques such as Mixins (Multiple Inheritance)	3) FBVs cannot use Object Oriented Techniques
4) In CBVs, Less Coding	4) In FBVs, More Coding
5) Default Context Dictionary and Default Template Files Support Available	5) Default Context Dictionary and Default Template Files Support not Available
6) Handling HTTP Methods by seperate Class Methods such as get() and post()	6) Handling HTTP Methods via Conditional Braching if request.method == 'POST'
7) There is a Restriction on Functionality and hence Less Power.	7) Based on Requirement we can implement any Functionality and hence these are more Powerful
8) Implicit Execution Flow and hence reduces Readability.	8) Explicit Execution Flow and hence improves Readability.

Note: In Real Time the most commonly used views are CBVs.If CBV can not handle our requirement then only we should go for FBVs.

How to develop web application by using Flask Framework:

pip Install Flask

app.py:

```
1) from flask import Flask
2) app=Flask(__name__)
3) @app.route('/hello')
4) def hello_world():
5)     return '<h1>Hello this is from Flask Application</h1>'
6) app.run()
```

How to Run Development Server:

py app.py

How to send Request:

http://127.0.0.1:5000/test

Note: The default server port number in Flask Framework is 5000. But based on our requirement we can change.

app.run(port=7777)

Django ORM:

ORM → Object Relation Mapping

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

The return type of all() method is QuerySet.

```
qs=Employee.objects.all()
print(type(qs)) # <class 'django.db.models.query.QuerySet'>
```

If we want to get only one record then we can use get() method.

```
emp = Employee.objects.get(id=1)
print(type(emp)) #<class 'testapp.models.Employee'>
```

The return type of get() method is Employee Object.

Case-1: How to find Query associated with QuerySet

```
qs = Employee.objects.all()
print(qs.query)
```

Case-2: How to Filter Records based on some Condition

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

It returns all employees whose salary greater than 15000

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

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

Similarly we can use __lt and __lte

Various possible Field Look ups are:

1) exact → Exact Match

```
Entry.objects.get(id__exact=14)
```

2) iexact → Case-insensitive exact Match

```
Blog.objects.get(name__iexact='beatles blog')
Blog.objects.get(name__iexact=None)
```

The equivalent queries are:

```
SELECT ... WHERE name ILIKE 'beatles blog';
SELECT ... WHERE name IS NULL;
```

3) contains → Case-sensitive Containment Test

```
Entry.objects.get(headline__contains='Lennon')
```

SQL equivalent:

```
SELECT ... WHERE headline LIKE '%Lennon%';
```

4) icontains → Case-insensitive Containment Test

Example: Entry.objects.get(headline__icontains='Lennon')

5) in

In a given iterable; often a List, Tuple OR queryset.

Examples:

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

Entry.objects.filter(headline__in='abc')

6) gt

Greater than

Example: Entry.objects.filter(id__gt=4)

7) gte

Greater than or equal to.

8) lt

Less than.

9) lte

Less than or equal to.

10) startswith

Case-sensitive starts-with.

Example: Entry.objects.filter(headline__startswith='Lennon')

11) istartswith

Case-insensitive starts-with.

Example: Entry.objects.filter(headline__istartswith='Lennon')

12) endswith

Case-sensitive ends-with.

Example: Entry.objects.filter(headline__endswith='Lennon')

13) iendswith

Case-insensitive ends-with.

Example: Entry.objects.filter(headline__iendswith='Lennon')

14) range

Range test (inclusive).

Example:

```
1) import datetime
2) start_date = datetime.date(2005, 1, 1)
3) end_date = datetime.date(2005, 3, 31)
4) Entry.objects.filter(pub_date__range=(start_date, end_date))
```

Eg-2: employees=Employee.objects.filter(esal__range=(12000,16000))

Note: There are several other field lookups are possible. (Documentation)
https://docs.djangoproject.com/en/2.1/ref/models/querysets/#id4

Case-3: How to implement OR Queries in Django ORM

2 ways are available

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

Eg 1: To get all employees whose name startswith 'A' OR salary < 15000

```
employees = Employee.objects.filter(ename__startswith='A') |
Employee.objects.filter(esal__lt=15000)

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

Case-4: How to implement AND Queries in Django ORM

3 ways

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

Eg: To get all employees whose name startswith 'J' AND salary < 15000

1) employees= Employee.objects.filter(ename__startswith='J',esal__lt=15000)
2) employees= Employee.objects.filter(ename__startswith='J') &
   Employee.objects.filter(esal__lt=15000)
3) employees= Employee.objects.filter(Q(ename__startswith='J') & Q(esal__lt=15000))

Case-5: How to implement NOT Queries in Django ORM

2 ways

1) exclude(condition)
2) filter(~Q(condition))

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

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

Case-6: How to perform Union Operation for Query Sets of the same OR different Models

By using union operation, we can combine results of 2 or more query sets.

```
q1=Employee.objects.filter(esal__lt<15000)
q2=Employee.objects.filter(ename__endswith='J')
```

q3=q1.union(q2)

Note: The union operator can be performed only with the querysets having the same fields and data types. Otherwise we will get error saying

django.db.utils.OperationalError: SELECTs to the left and right of UNION do not have the same number of result columns.

We can perform union operation on common columns.

Eg: Student(name, mailid, aadharnumber, marks)
Teacher(name, mailid, aadharnumber, subject, salary)

```
q1 = Student.objects.all().values_list('name','mailid','aadharnumber')
q2 = Teacher.objects.all().values_list('name','mailid','aadharnumber')
q3 = q1.union(q2)
```

Case-7: How to select only some columns in the queryset

3 Ways

1) **By using values_list()**
   Eg: q1 = Student.objects.all().values_list('name','mailid','aadharnumber')
2) **By using values()**
   Eg: q1 = Student.objects.all().values('name','mailid','aadharnumber')
3) **By using only():**
   Eg: q1 = Student.objects.all().only('name','mailid','aadharnumber')

Note: Difference between values() and only() Methods
In the case of values() only specified columns will be selected. But in the case of only() in addition to specified columns 'id' column also will be selected.

Case-8: Aggregate Functions
Django ORM defines several functions to perform aggregate operations.
Avg(),Max(),Min(),Sum(),Count()

views.py

```
1) from django.shortcuts import render
2) from django.db.models import Q
3) from django.db.models import Avg,Sum,Max,Min,Count
4) from testapp.models import Employee
5)
6) # Create your views here.
7) def display_view(request):
8)     avg1=Employee.objects.all().aggregate(Avg('esal'))
9)     max=Employee.objects.all().aggregate(Max('esal'))
```

```
10)  min=Employee.objects.all().aggregate(Min('esal'))
11)  sum=Employee.objects.all().aggregate(Sum('esal'))
12)  count=Employee.objects.all().aggregate(Count('esal'))
13)  my_dict={'avg':avg1,'max':max,'min':min,'sum':sum,'count':count}
14)  return render(request,'testapp/aggregate.html',my_dict)
```

aggregate.html

```
1)  <!DOCTYPE html>
2)  {% extends 'testapp/base.html'%}
3)      {%block body_block%}
4)      <h1>Employee Aggregate Information</h1><hr>
5)      <ul>
6)        <li>Average Salary:{{avg}}</li>
7)        <li>Max Salary:{{max}}</li>
8)        <li>Min Salary:{{min}}</li>
9)        <li>Total Salary:{{sum}}</li>
10)       <li>Number of Employees :{{count}}</li>
11)     </ul>
12)     {%endblock%}
```

Case-9: How to Create, Update, Delete Records

How to Add Record

```
1)  >>> from testapp.models import Employee
2)  >>> Employee.objects.all().count()
3)  30
4)  >>> e=Employee(eno=1234,ename='Dheeraj',esal=1234.0,eaddr='Delhi')
5)  >>> e.save()
6)  >>> Employee.objects.all().count()
7)  31
```

2nd Way:

```
>>> Employee.objects.create(eno=2222,ename='Sreeram',esal=10000,eaddr='Bangalore')
```

How to Add Multiple Records at a Time:

By using bulk_create() method.

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

How to Delete a Single Record:

```
1) >>> e=Employee.objects.get(eno=1)
2) >>> e.eno
3) 1
4) >>> e.ename
5) 'DDD'
6) >>> e.delete()
```

How to Delete Multiple Records:

```
1) >>> qs=Employee.objects.filter(esal__gte=15000)
2) >>> qs.count()
3) 14
4) >>> qs.delete()
5) (14, {'testapp.Employee': 14})
```

How to Delete all Records(Truncate Operation in SQL):

```
>>> Employee.objects.all().delete()
```

How to Update Field of a Particular Record:

```
1) >>> from testapp.models import Employee
2) >>> e=Employee.objects.get(eno=7014)
3) >>> e.ename
4) 'Peter Lewis'
5) >>> e.ename='Durga'
6) >>> e.save()
7) >>> e.ename
8) 'Durga'
```

How to Order queryset in Sorting Order:

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

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

For Descending order we have to use '-'

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

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

Returns highest salary employee object

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

Returns Second highest salary employee object

employees = Employee.objects.all().order_by('-esal')[0:3]
Returns list of top 3 highest salary employees info

But in the case of strings for alphabetical order:
employees = Employee.objects.all().order_by('ename')
In this case, case will be considered.
If we want to ignore case then we should use Lower() Function

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

Working with Multiple Databases Simultaneosuly:

Model Inheritance

It is very useful and powerful feature of django.

There are 4 types of Model Inheritance.
1) Abstract Base Class Model Inheritance
2) Multi table Inheritance
3) Proxy Model Inheritance
4) Multiple Inheritance

1. Abstract Base Class Model Inheritance:

If several Model classes having common fields, then it is not recommended to write these fields seperately in every Model class. It increases length of the code and reduces readability.

We can seperate these common fields into another Model class,which is also known as Base Class. If we extend Base class automatically common fields will be inherited to the child classes.

Without Inheritance

```
1)  class Student(models.Model):
2)      name=models.CharField(max_length=64)
3)      email=models.EmailField()
4)      address=models.CharField(max_length=256)
5)      rollno=models.IntegerField()
6)      marks=models.IntegerField()
7)
8)  class Teacher(models.Model):
9)      name=models.CharField(max_length=64)
10)     email=models.EmailField()
```

```
11)    address=models.CharField(max_length=256)
12)    subject=models.CharField(max_length=64)
13)            salary=models.FloatField()
```

With Inheritance

```
1)  class ContactInfo(models.Model):
2)      name=models.CharField(max_length=64)
3)      email=models.EmailField()
4)      address=models.CharField(max_length=256)
5)      class Meta:
6)          abstract=True
7)
8)  class Student(ContactInfo):
9)      rollno=models.IntegerField()
10)     marks=models.IntegerField()
11)
12) class Teacher(ContactInfo):
13)     subject=models.CharField(max_length=64)
14)     salary=models.FloatField()
```

In this case only Student and Teacher tables will be created which includes all the fields of ContactInfo.

Note: CotnactInfo class is an abstract class and hence table won't be created.
It is not possible to register abstract model classes to the admin interface.If we are trying to do then we will get error.

2. Multi table Inheritance:

If the base class is not abstract then such type of inheritance is called multi table inheritance.

In Multitable inheritance,inside database tables will be created for both Parent and Child classes. Multi table inheritance uses an implicit OneToOneField to link Parent and Child. i.e by using one-to-one relationship multi table inheritance is internally implemented.

Django hides internal structure and creates feeling that both tables are independent.

```
1)  class BasicModel(models.Model):
2)      f1=models.CharField(max_length=64)
3)      f2=models.CharField(max_length=64)
4)      f3=models.CharField(max_length=64)
5)
6)  class StandardModel(BasicModel):
```

```
7)    f4=models.CharField(max_length=64)
8)    f5=models.CharField(max_length=64)
```

Corresponding Database Tables are:

```
mysql> desc testapp_basicmodel;

+-------+-------------+------+-----+---------+ --------------- +
| Field | Type        | Null | Key | Default | Extra           |
+-------+-------------+------+-----+---------+ --------------- +
| id    | int(11)     | NO   | PRI | NULL    | auto_increment  |
| f1    | varchar(64) | NO   |     | NULL    |                 |
| f2    | varchar(64) | NO   |     | NULL    |                 |
| f3    | varchar(64) | NO   |     | NULL    |                 |
+-------+-------------+------+-----+---------+ --------------- +
4 rows in set (0.00 sec)

mysql> desc testapp_standardmodel;

+------------------+-------------+------+-----+---------+------ +
| Field            | Type        | Null | Key | Default | Extra |
+------------------+-------------+------+-----+---------+------ +
| basicmodel_ptr_id | int(11)     | NO   | PRI | NULL    |       |
| f4               | varchar(64) | NO   |     | NULL    |       |
| f5               | varchar(64) | NO   |     | NULL    |       |
+------------------+-------------+------+-----+---------+------ +
3 rows in set (0.01 sec)
```

Model Manager:

Model Manager can be used to interact with the database. By default Model Manager is available through the Model.objects property.i.e Model.objects is of type django.db.models.manager.Manager.

1. What is the purpose of Model Manager?
   To interact with database

2. How to get Default Model Manager?
   By using Model.objects property

3. Model Manager is what type?
   django.db.models.manager.Manager

```
>>> from testapp.models import Employee
>>> type(Employee.objects)
```

<class 'django.db.models.manager.Manager'>
We can customize the defualt behaviour of Model Manager by defining our own Customer Manager.

How to define our own Custom Manager:

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

Eg: To retrieve all employees data according to ascending order of eno, we have to define Custom Manager class as follows.

models.py

```
1)  from django.db import models
2)  class CustomManager(models.Manager):
3)      def get_queryset(self):
4)          return super().get_queryset().order_by('eno')
5)  # Create your models here.
6)
7)  class Employee(models.Model):
8)      eno=models.IntegerField()
9)      ename=models.CharField(max_length=64)
10)     esal=models.FloatField()
11)     eaddr=models.CharField(max_length=256)
12)     objects=CustomManager()
```

When ever we are using all() method it will always get employees in ascending order of eno

Based on our requirement we can define our own new methods also inside Custom Manager class.

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

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

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

Q) To Customize all() Method Behaviour, which Method we have to override inside Custom Manager Class?

get_queryset() Method

Q) In Custom Manager Class, is it Possible to define New Methods?

Yes

views.py

------ from django.shortcuts import render

```
1)  from testapp.models import Employee
2)
3)  # Create your views here.
4)  def display_view(request):
5)      #employees=Employee.objects.get_emp_sal_range(12000,20000)
6)  employees=Employee.objects.get_employees_sorted_by('esal')
7)      my_dict={'employees':employees}
8)      return render(request,'testapp/index.html',my_dict)
```

index.html

```
1)  <!DOCTYPE html>
2)  {% extends 'testapp/base.html'%}
3)       {%block body_block%}
4)       <h1>Employee Information Dash Board</h1><hr>
5)       <table border='2'>
6)         <thead>
7)           <th>Employee Number</th>
8)           <th>Employee Name</th>
9)           <th>Employee Salary</th>
10)          <th>Employee Address</th>
11)
12)        </thead>
13)        {%for  emp in employees %}
14)        <tr>
15)          <td>{{emp.eno}}</td>
16)          <td>{{emp.ename}}</td>
17)          <td>{{emp.esal}}</td>
18)          <td>{{emp.eaddr}}</td>
19)
20)        </tr>
21)        {%endfor%}
22)      </table><br><br><br>
23)  {%endblock%}
```

Proxy Model Inheritance:

For the same Model we can provide a customized view without touching the database is possible by using Proxy Model Inheritance.

In this inheritance a seperate new table won't be created and the new model also pointing to the same old table.

Eg:

```
1)  class Employee(models.Model):
2)    fields
3)
4)  class ProxyEmployee(Employee):
5)    class Meta:
6)      proxy=True
```

Both Employee and ProxyEmployee are pointing to the same table only.
In the admin interface if we add a new record to either Employee or ProxyEmployee, then automatically those changes will be reflected to other model view.

Demo Application:

models.py

```
1)  from django.db import models
2)
3)  class CustomManager1(models.Manager):
4)     def get_queryset(self):
5)        return super().get_queryset().filter(esal__gte=15000)
6)
7)  class CustomManager2(models.Manager):
8)     def get_queryset(self):
9)        return super().get_queryset().order_by('ename')
10)
11) class CustomManager3(models.Manager):
12)    def get_queryset(self):
13)       return super().get_queryset().filter(eno__lt=1000)
14)
15) # Create your models here.
16) class Employee(models.Model):
17)    eno=models.IntegerField()
18)    ename=models.CharField(max_length=64)
19)    esal=models.FloatField()
```

```
    eaddr=models.CharField(max_length=256)
    objects=CustomManager1()

class ProxyEmployee(Employee):
    objects=CustomManager2()
    class Meta:
        proxy=True

class ProxyEmployee2(Employee):
    objects=CustomManager3()
    class Meta:
        proxy=True
```

admin.py

```
from django.contrib import admin
from testapp.models import Employee,ProxyEmployee,ProxyEmployee2

# Register your models here.
class EmployeeAdmin(admin.ModelAdmin):
    list_display=['eno','ename','esal','eaddr']

class ProxyEmployeeAdmin(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(ProxyEmployee,ProxyEmployeeAdmin)
admin.site.register(ProxyEmployee2,ProxyEmployee2Admin)
```

views.py

```
from django.shortcuts import render
from testapp.models import Employee,ProxyEmployee,ProxyEmployee2

# Create your views here.
def display_view(request):
    # employees=Employee.objects.all()
    # employees=ProxyEmployee.objects.all()
    employees=ProxyEmployee2.objects.all()
    my_dict={'employees':employees}
    return render(request,'testapp/index.html',my_dict)
```

index.html

```
<!DOCTYPE html>
{% extends 'testapp/base.html'%}
    {%block body_block%}
    <h1>Employee Information Dash Board</h1><hr>
    <table border='2'>
     <thead>
       <th>Employee Number</th>
       <th>Employee Name</th>
       <th>Employee Salary</th>
       <th>Employee Address</th>

     </thead>
     {%for  emp in employees %}
     <tr>
       <td>{{emp.eno}}</td>
       <td>{{emp.ename}}</td>
       <td>{{emp.esal}}</td>
       <td>{{emp.eaddr}}</td>

     </tr>
     {%endfor%}
   </table><br><br><br>

    {%endblock%}
```

Middleware:

Middleware is a framework of hooks into Django's request/response processing.
It is a ligth, low level 'plugin' system for globally altering Django's input or output.

If we want to perform any activity at the time of pre processing of the request or post processing of the request then we should go for middleware.

Diagram
client-->request-->middleware-->modifiedrequest
Server-->response-->middleware-->modifiedresponse

Django contains several inbuilt middlewares which are configured inside settings.py

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

```
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

All these middlewares will be executed before and after processing of every request.

1. SecurityMiddleware provides security enhancements like SSL Redirects(like from http request to https request) etc

2. SessionMiddleware enables session support.

3. CommonMiddleware provides a common set of features like adding slash at the end of the url

4. CsrfViewMiddleware is responsible to verify whether POST request has csrf_token or not

5. AuthenticationMiddleware is responsible to add user attribute to the request object.
If we comment this middleware in settings.py then we cannot access user attribute in our view function.If we are trying to access you will get error.
print(request.user)

AttributeError at /first/
'WSGIRequest' object has no attribute 'user'

Note: Middlewares are applicable for every incoming request and for every outgoing response.

Middleware Structure:

Based on our requirement we can define our own customized middlewares.
Every customized middleware is a python class and it is the child class of object,contains 2 mandatory methods and 3 optional methods.

class LoginMiddleware(object):

```
  def __init__(self,get_response):
    #one time configuration and initialization on start-up,get_response is a reference to previous middleware response
    self.get_response=get_response
```

def __call__(self,request):

#This code will be executed before the view(and other middleware) is called
response=self.get_response(request) #It triggers next phase
#This code will be executed after the view(and other middleware) is called
return response # to finish middleware sequence

def process_view(self,request,view_func,view_args,view_kwargs):

Logic will be executed before a call to View
Gives access to the view itself and arguments

def process_exception(self,request,exception):

#Logic will be executed if an exception/error occurs in the view

def process_template_response(self,request,response):

#Logic is executed after view is called.
It is required to alter the response itself to perform additional logic on it like modifying context or template.

Demo Application for Custom Middleware Execution Flow:

middleware.py:(inside application folder)

```
1)  class ExecutionFlowMiddleware(object):
2)      def __init__(self,get_response):
3)          self.get_response=get_response
4)
5)      def __call__(self,request):
6)          print('This line added at pre-processing of request')
7)          response=self.get_response(request)
8)          print('This line added at post-processing of request')
9)          return response
```

settings.py

```
1)  MIDDLEWARE = [
2)..... ,
3)      'testapp.middleware.ExecutionFlowMiddleware'
4)  ]
```

views.py

```
1)  from django.http import HttpResponse
2)
```

```
3)  # Create your views here.
4)  def welcome_view(request):
5)      print('This line added by view function')
6)      return HttpResponse('<h1>Custom Middleware Demo</h1>')
```

Results:

If we send a request in the server console we can see:

This line added at pre-processing of request
This line added by view function
This line added at post-processing of request

Before and After processing every request middleware will be executed.

Execution Process for a Single Middleware Class:

1)__init__() method will be called only once at the time of server start-up.
2)__call__() method will be called for every request.
3) If we declare process_view() method then it will be called
4) Inside __call__() method whenever we are using self.get_response(request) then view function starts its execution
5) If we declare process_exception() method, then it will be executed if any exception/error occurs inside view function.
6) View Method Finishes.
7) If we declare process_template_response() then it will be executed whenever view returns TemplateResponse.

Middleware application to show information saying application under maintenance:

middleware.py

```
1)  from django.http import HttpResponse
2)  class AppMaintenanceMiddleware(object):
3)      def __init__(self,get_response):
4)          self.get_response=get_response
5)
6)      def __call__(self,request):
7)          return HttpResponse('<h1>Currently Application under maintenance...plz try a
    fter 2 days!!!')
```

settings.py

```
1) MIDDLEWARE = [
2)     ...
3)     'testapp.middleware.AppMaintenanceMiddleware'
4) ]
```

views.py

```
1) from django.http import HttpResponse
2)
3) # Create your views here.
4) def home_page_view(request):
5)     return HttpResponse('<h1>Hello This is from home page view</h1>')
```

Middleware application to show meaningful response if view function raises any error:

In this case we have to use process_exception() method which will be executed if view function raising any exception/error.

middleware.py

```
1) from django.http import HttpResponse
2) class ErrorMessageMiddleware(object):
3)     def __init__(self,get_response):
4)         self.get_response=get_response
5)
6)     def __call__(self,request):
7)         return self.get_response(request)
8)
9)     def process_exception(self,request,exception):
10)        return HttpResponse('<h1>Currently we are facing some technical problems pl
    z try after some time!!!</h1>')
```

settings.py

```
1) MIDDLEWARE = [
2)     ...
3)     'testapp.middleware.ErrorMessageMiddleware'
4) ]
```

views.py

```
1)  from django.http import HttpResponse
2)
3)  # Create your views here.
4)  def home_page_view(request):
5)      print(10/0)
6)      return HttpResponse('<h1>Hello This is from home page view</h1>')
```

How to display raised exception information:

```
def process_exception(self,request,exception):
  return HttpResponse('<h1>Currently we are facing some technical problems plz try after some time!!!</h1><h2>Raised Exception:{}</h2><h2>Exception Message:{}</h2>'.format(exception.__class__.__name__,exception))
```

Configuration of multiple middleware classes:

We can configure any number of middlewares and all these middlewares will be executed according to order declared inside settings

diagram

middleware.py

```
1)  class FirstMiddleware(object):
2)      def __init__(self,get_response):
3)          self.get_response=get_response
4)
5)      def __call__(self,request):
6)          print('This line printed by FirstMiddleware at pre-processing of request')
7)          response=self.get_response(request)
8)          print('This line printed by FirstMiddleware at post-processing of request')
9)          return response
10) class SecondMiddleware(object):
11)     def __init__(self,get_response):
12)         self.get_response=get_response
13)
14)     def __call__(self,request):
15)         print('This line printed by SecondMiddleware at pre-processing of request')
16)         response=self.get_response(request)
17)         print('This line printed by SecondMiddleware at post-processing of request')
18)         return response
```

settings.py

```
1) MIDDLEWARE = [
2)..... ,
3)     'testapp.middleware.FirstMiddleware',
4)     'testapp.middleware.SecondMiddleware'
5) ]
```

views.py

```
1) def home_page_view(request):
2)     print('This line printed by view function')
3)     return HttpResponse('<h1>Hello This is from home page view</h1>')
```

In the server console:

This line printed by FirstMiddleware at pre-processing of request
This line printed by SecondMiddleware at pre-processing of request
This line printed by view function
This line printed by SecondMiddleware at post-processing of request
This line printed by FirstMiddleware at post-processing of request

Note: If we change the order of middlewares inside settings.py then the output at server console is :

This line printed by SecondMiddleware at pre-processing of request
This line printed by FirstMiddleware at pre-processing of request
This line printed by view function
This line printed by FirstMiddleware at post-processing of request
This line printed by SecondMiddleware at post-processing of request

Deployment:

There are several deployment options are available to deploy our django web application. These options will be based on

1. Scalability
2. Performance
3. Price
4. Security
5. Easy to use
etc

The following are various deployment options
1. PythonAnywhere.com

It is very simple and easy to host
2. Digital Ocean --->VPS(Virtual Privater Server)
3. Heroku
4. Amazon Cloud
etc

Note: For every platform clear documentation steps are available.

Need of Version Control Systems:

1. To maintain multiple versions of the same product
2. At any point of time we can have a backup of previous version
3. We can see the difference between 2 or more versions of our code base
4. We can run mutliple versions of the same product simultaneously
5. It helps us to track project history over time and to collaborate easily with others.
etc

The following is the list most popular version control systems
1. GIT
2. Apache Subversion
3. Mercurial
4. Concurrent Version System(CVS)
5. GNU Bazaar
etc

Git vs GitHub:

Git is a version control system that helps to track changes in our code
GitHub is a company/website that helps manage git and and host our files on their site. i.e GitHub is remove hosting service to host our code repository.
Similar to GitHub there are several hosting platforms like Gitlab,BitBucket etc

Note: If our application at remote hosting platform then deployment on various paltforms will become very easy.

Version Control vs Hosting Platform vs Deployment Platform

How to install git:

https://git-scm.com/downloads

How to create account in github.com

Git Respository:
Git is a set of layers.

Each layer has a function. We can use git to move files between these layers.

Activtities related to Git Repository:

1. create a folder named with my_repo which acts as Git Repository.
2. Copy the required files to this folder for tracking purposes
3. initialize Git by using the following command
   git init
4. By default git won't track any files. We have to add files to the Staging area,such files only can be tracked by GIT.
We can add files to the staging area by using the following command

git add filename1

To add all files present in current working directory we have to use
git add .

5. Whenever we perform commit,for all files present in staging area,snapshots will be created by git. We can perform commit as follows

git commit -m 'comment'

6. We can check the status by using the command
git status

```
LENOVO@LENOVO-PC MINGW64 /e
$ cd my_repo1

LENOVO@LENOVO-PC MINGW64 /e/my_repo1
$ git init
Initialized empty Git repository in E:/my_repo1/.git/

LENOVO@LENOVO-PC MINGW64 /e/my_repo1 (master)
$ git add test.py

LENOVO@LENOVO-PC MINGW64 /e/my_repo1 (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   test.py
```

```
LENOVO@LENOVO-PC MINGW64 /e/my_repo1 (master)
$ git commit -m 'firstcommit'
[master (root-commit) b53bd68] firstcommit
 1 file changed, 3 insertions(+)
 create mode 100644 test.py

LENOVO@LENOVO-PC MINGW64 /e/my_repo1 (master)
$ git status
On branch master
nothing to commit, working tree clean

LENOVO@LENOVO-PC MINGW64 /e/my_repo1 (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   test.py

no changes added to commit (use "git add" and/or "git commit -a")

LENOVO@LENOVO-PC MINGW64 /e/my_repo1 (master)
$ git add test.py

LENOVO@LENOVO-PC MINGW64 /e/my_repo1 (master)
$ git commit -m 'second'
[master 1677eb2] second
 1 file changed, 2 insertions(+), 1 deletion(-)

LENOVO@LENOVO-PC MINGW64 /e/my_repo1 (master)
$ git status
On branch master
nothing to commit, working tree clean

----------------------------
…or create a new repository on the command line
echo "# django-deployment-first-application" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/djangodurga/django-deployment-first-
application.git
git push -u origin master
```

```
----------------------------------------------------
LENOVO@LENOVO-PC MINGW64 ~
$ cd e:

LENOVO@LENOVO-PC MINGW64 /e
$ cd my_codebase/

LENOVO@LENOVO-PC MINGW64 /e/my_codebase
$ git init
Initialized empty Git repository in E:/my_codebase/.git/

LENOVO@LENOVO-PC MINGW64 /e/my_codebase (master)
$ git add .

LENOVO@LENOVO-PC MINGW64 /e/my_codebase (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   firstproject/db.sqlite3
        new file:   firstproject/firstproject/__init__.py
        new file:   firstproject/firstproject/__pycache__/__init__.cpython-36.pyc
        new file:   firstproject/firstproject/__pycache__/settings.cpython-36.pyc
        new file:   firstproject/firstproject/__pycache__/urls.cpython-36.pyc
        new file:   firstproject/firstproject/__pycache__/wsgi.cpython-36.pyc
        new file:   firstproject/firstproject/settings.py
        new file:   firstproject/firstproject/urls.py
        new file:   firstproject/firstproject/wsgi.py
        new file:   firstproject/manage.py
        new file:   firstproject/testapp/__init__.py
        new file:   firstproject/testapp/__pycache__/__init__.cpython-36.pyc
        new file:   firstproject/testapp/__pycache__/admin.cpython-36.pyc
        new file:   firstproject/testapp/__pycache__/models.cpython-36.pyc
        new file:   firstproject/testapp/__pycache__/views.cpython-36.pyc
        new file:   firstproject/testapp/admin.py
        new file:   firstproject/testapp/apps.py
        new file:   firstproject/testapp/migrations/__init__.py
        new file:   firstproject/testapp/migrations/__pycache__/__init__.cpython-36.pyc
        new file:   firstproject/testapp/models.py
        new file:   firstproject/testapp/tests.py
        new file:   firstproject/testapp/views.py
```

```
LENOVO@LENOVO-PC MINGW64 /e/my_codebase (master)
$ git commit -m 'firstcommit'
[master (root-commit) c95b9d9] firstcommit
 22 files changed, 213 insertions(+)
 create mode 100644 firstproject/db.sqlite3
 create mode 100644 firstproject/firstproject/__init__.py
 create mode 100644 firstproject/firstproject/__pycache__/__init__.cpython-36.pyc
 create mode 100644 firstproject/firstproject/__pycache__/settings.cpython-36.pyc
 create mode 100644 firstproject/firstproject/__pycache__/urls.cpython-36.pyc
 create mode 100644 firstproject/firstproject/__pycache__/wsgi.cpython-36.pyc
 create mode 100644 firstproject/firstproject/settings.py
 create mode 100644 firstproject/firstproject/urls.py
 create mode 100644 firstproject/firstproject/wsgi.py
 create mode 100644 firstproject/manage.py
 create mode 100644 firstproject/testapp/__init__.py
 create mode 100644 firstproject/testapp/__pycache__/__init__.cpython-36.pyc
 create mode 100644 firstproject/testapp/__pycache__/admin.cpython-36.pyc
 create mode 100644 firstproject/testapp/__pycache__/models.cpython-36.pyc
 create mode 100644 firstproject/testapp/__pycache__/views.cpython-36.pyc
 create mode 100644 firstproject/testapp/admin.py
 create mode 100644 firstproject/testapp/apps.py
 create mode 100644 firstproject/testapp/migrations/__init__.py
 create mode 100644 firstproject/testapp/migrations/__pycache__/__init__.cpython-
36.pyc
 create mode 100644 firstproject/testapp/models.py
 create mode 100644 firstproject/testapp/tests.py
 create mode 100644 firstproject/testapp/views.py

LENOVO@LENOVO-PC MINGW64 /e/my_codebase (master)
$ git status
On branch master
nothing to commit, working tree clean

LENOVO@LENOVO-PC MINGW64 /e/my_codebase (master)
$ git remote add origin https://github.com/djangodurga/django-deployment-first-
application.git

LENOVO@LENOVO-PC MINGW64 /e/my_codebase (master)
$ git push -u origin master
Enumerating objects: 29, done.
Counting objects: 100% (29/29), done.
Delta compression using up to 4 threads
Compressing objects: 100% (27/27), done.
```

```
Writing objects: 100% (29/29), 6.98 KiB | 376.00 KiB/s, done.
Total 29 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), done.
remote:
remote: Create a pull request for 'master' on GitHub by visiting:
remote:      https://github.com/djangodurga/django-deployment-first-
application/pull/new/master
remote:
To   https://github.com/djangodurga/django-deployment-first-application.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

Deployment on pythonanywhere.com:

---

pricing and signup-->beginner account(free)

username:durgasoftdurga
pwd:github12345

Open the console and Create Virtual Environment:

Console-->Bash-->

```
$mkvirtualevn --python=python3.6 myproj
$mkvirtualenv --python=python3.6 myproj
```

here myproj is the name of the virtual environment

```
(myproj) 09:40 ~ $
```

If we are seeing this means our virtual enviroment created and active.

Which package already installed in this virtual env:

```
(myproj) 09:40 ~ $ pip list
Package    Version
---------- -------
pip        18.1
setuptools 40.5.0
wheel      0.32.2
```

install django:

It is highly recommended to install the same version which is available on our local machine.
How to check our local machine django version:

```
C:\Users\LENOVO>python
Python 3.6.5 (v3.6.5:f59c093
Type "help", "copyright", "c
>>> import django
>>> django._version___
'1.11'
```

We can install django in virtualenv as follows.
pip install -U django==1.11

How to check whether django installed properly:

(myproj) 09:48 ~ $ pip list

```
Package     Version
--------    -------
Django      1.11
pip         18.1
pytz        2018.7
setuptools  40.5.0
wheel       0.32.2
```

(myproj) 09:49 ~ $ which django-admin
/home/durgasoftdjango/.virtualenvs/myproj/bin/djan
go-admin

Copy our application from github to our virtualenvironment (pythonanywhere):
---
https://github.com/djangodurga/django-second-deployment
clone or download
copy url: https://github.com/djangodurga/django-second-deployment.git

```
(myproj) 09:54 ~ $ git clone https://github.com/djangodurga/django-second-deployme
nt.git
Cloning into 'django-second-deployment'...
remote: Enumerating objects: 29, done.
remote: Counting objects: 100% (29/29), done.
remote: Compressing objects: 100% (25/25), done.
remote: Total 29 (delta 2), reused 29 (delta 2), pack-reused 0
Unpacking objects: 100% (29/29), done.
Checking connectivity ... done.
```

```
(myproj) 09:55 ~ $ ls
README.txt  django-second-deployment
(myproj) 09:56 ~ $ cd django-second-deployment/
(myproj) 09:56 ~/django-second-deployment (master)$ ls
secondproject
(myproj) 09:56 ~/django-second-deployment (master)$ cd secondproject/
(myproj) 09:56 ~/django-second-deployment/secondproject (master)$ python manage.py
makemigrations
No changes detected
(myproj) 09:57 ~/django-second-deployment/secondproject (master)$ python manage.py
migrate

(myproj) 09:58 ~/django-second-deployment/secondproject (master)$ pwd
/home/durgasoftdjango/django-second-deployment/secondproject
```

This is the sourcecode path

Configuration on the web tab:

Add a new web app--->Next-->

If we want to develop a fresh application we have to select Django.
But if we have application already then we have to select

Manual configuration (including virtualenvs)

Next-->Select Python Version-->Python 3.6-->Next

Source code:
/home/durgasoftdjango/django-second-deployment/secondproject

Virtualenv:
/home/durgasoftdjango/.virtualenvs/myproj

WSGI Configuration:

WSGI configuration file:/var/www/durgasoftdjango_pythonanywhere_com_wsgi.py

In the configuration file just remove Hello World related things

```
# ++++++++++++ DJANGO ++++++++++++
# To use your own django app use code like this:
import os
import sys
#
```


```
## assuming your django settings file is at
'/home/durgasoftdjango/mysite/mysite/settings.py'
## and your manage.py is is at '/home/durgasoftdjango/mysite/manage.py'
path = '/home/durgasoftdjango/django-second-deployment/secondproject'
if path not in sys.path:
   sys.path.append(path)
os.chdir(path)
os.environ.setdefault('DJANGO_SETTINGS_MODULE','secondproject.settings')

import  django
django.setup()
#
#os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'
#
## then:
from django.core.wsgi import get_wsgi_application
application = get_wsgi_application()
```

Add 'durgasoftdjango.pythonanywhere.com' to ALLOWED_HOSTS:

---

from bash shell

nano settings.py
move to the required position and add :
ALLOWED_HOSTS = ['durgasoftdjango.pythonanywhere.com']

ctrl+o to save
ctrl+x to close

Static Files:

By default in pythonanywhere deployment static files won't be considered. Hence while accessing our web application and django admin site look and feel will be changed. For that we have perform some configurations in the web tab of dashboard.

Static files related to admin site:

Static files:
URL: /static/admin
Path: /home/durgasoftdjango/.virtualenvs/myproj/lib/python3.6/site-packages/django/contrib/admin/static/admin

Note: After performing any configuration changes,compulsory we should reload our application

Static files related to our application:

To reflect css files,js files and images used in our application we have to perform the following configuration under Static files:

URL: /static
Path: /home/durgasoftdjango/django-second-deployment/secondproject/static

Note: It is highly recommended to disable DEBUG mode in production, because we should not display application level sensitive information to the end user. For this,in settings.py, we have to set
DEBUG=False