DJANGO

By

Mr. Durga

(Sun Certified & Real Time Expert)

DURGA
SOFTWARE SOLUTIONS®

SRI SAI BHAVANI XEROX

Ram Mirra Towers, Ground Floor
Beside Satyam Theatre
Ameerpet, HYD
Cell: 99 08 19 59 58

INDEX

Chapter-1: Introduction to Web development and Django 1

Chapter-2: Django & Atom Installation and Development of First Web Application 6

Chapter-3: Django Templates and Static Files 21

Chapter-4: Working with Models and Databases 34

Chapter-5: Working with Django Forms 50

Chapter-6: Working with Django Model Forms 66

Chapter-7: Working with Advanced Template Features 75

Chapter-8: Session Management 83

Chapter-9: User Authentication and Authorization 103

Chapter-10: Class Based Views and CRUD Operations by using both CBVs and FBVs 112

Chapter-11: Django ORM 138

Chapter-12: Working with Advanced Model Concepts 146

Chapter-13: Working with Django Middleware 155

Chapter-14: Deployment of our application in the Live Environment 163

Chapter-15: Real Time Project: Blog Application Development 173

Chapter-16: Introduction to Web application Development by using Flask 198

DETAILED INDEX

Chapter-1: Introduction to Web development and Django 1
Web Application 2
Front-End 2
HTML 2
CSS 2
JavaScript 2
Static Response vs Dynamic Response 3
Back-End 3
DJango 3
Top 5 Features of Django Framework 4

- Fast
- Fully loaded
- Security
- Scalability
- Versatile

Chapter-2: Django & Atom Installation and Development of First Web Application 6
How to install django 7
ATOM IDE/Editor 7
Speciality of ATOM IDE 7
How to configure Atom for Python 8
Django Project vs Django Application 8
How to create Django Project 9
How to run Django Development server 10
How to send first request 11
Role of Web Server 11
Creation of First web application 11
Activities required for Application 13
Http Request flow in Django Application 15
Summary of sequence of activities related to Django Project 15
How to change Django Server Port 16
Various Practice Applications 16
Defining urlpatterns at application level instead of project level 18

Chapter-3: Django Templates and Static Files 21
Django Templates 22
Python stuff required to develop Template Based Application 22
Steps to develop Template Based Application 23
Template Tags 24

Application to send date and time from views.py to template file 25
Application To display date ,time and student information 26
Application to wish end user based on time 27
Working with Static Files 28
Process to include static files inside template 28
How to include css file 30
DURGASOFT NEWS PORTAL-Mini Project 30

Chapter-4: Working with Models and Databases 34
Database configuration 35
How to check django database connection 35
Configuration of MySQL database 36
Configuration of Oracle database 36
Model Class 36
Converting Model class into database specific sql code 37
How to see corresponding sql code of migrations 38
How to execute generated SQL code (migrate command) 38
What is the advantage of creating tables with 'migrate' command 39
How to check created table in django admin interface 39
Creation of super user to login to admin interface 39
Difference between makemigrations and migrate 39
To display data in admin interface in browser 40
MVT Diagram 42
Important FAQs related to Model and database configurations 42
Faker Module 42
Working with MYSQL Database(studentinfo project) 43
Working with MYSQL Database and Faker Module(studentinfo project) 45

Chapter-5: Working with Django Forms 50
Django Forms 51
Advantages of Django Forms over HTML forms 51
Process to generate Django forms 51
CSRF(Cross Site Request Forgery) Token 53
How to process input data from the form inside views.py file 54
Student Name and Marks Submission Form Project(formproject1) 54
Student FeedBack Form Project 56
Form Validations 58
1. Explicitly by the programmer by using clean methods 58
2. By using Django inbuilt validators 59
How to implement custom validators by using the same validator parameter 60
Validation of total form directly by using a single clean method 60
How to check original pwd and reentered pwd are same or not in Signup Form 61
How to prevent requests from BOT 63

Chapter-6: Working with Django Model Forms 66
Model Forms(Forms based on Model) 67
How to develop Model based forms 67
How to save user's input data to database in Model based forms 68
Demo project-1(modelformproject) 68
Demo Project-2(movieproject) 69
How to add date widget 74

Chapter-7: Working with Advanced Template Features 75
Template Inheritance 76
How to implement template inheritane 76
Demo program: advtempproject 76
Advantages of Template Inheritance 77
How to add seperate css files to child templates 78
Tempalte Filters 79
Syntax of Template Filter 79
How to create our own filter 80
Template Tags for urls 82

Chapter-8: Session Management 83
Session Management 84
Session Management By using Cookies 84
How to test our browser supports Cookies or not 84
Application-1:Session Management by using Cookies(PageCount Application) 85
Application-2:Session Management by using Cookies 86
Application-3:Session Management by using Cookies 88
Limitations of Cookies 90
Temporary vs Permanent Cookies 90
Application-4:Session Management by using Cookies(Shopping Cart Application) 91
Session Management By using Session API(Django Session Framework) 93
Useful Methods for session Management 95
Application-1:Session Management by using Session API(PageCount Application) 95
Application-2:Session Management by using Session API(Profile Application) 96
Application-3:Session Management by using Session API(Shopping Cart Application) 99
Important Methods related to Session 101
Browser Length Sessions and Persistent Sessions 102

Chapter-9: User Authentication and Authorization 103
User Authentication and Authorization 104
Difference between Static and Media Folders 105
How to configure media folder in settings.py file 105
Authentication Application 106

Chapter-10: Class Based Views and CRUD Operations by using both CBVs and FBVs 112
Class Based Views(CBVs) 113
HelloWorld Application By using ClassBasedViews 113
Template Based Application by using Class Based Views 114
How to send context paramters 114
ListView 114
How to create template file for ListView 115
How to provide our own context object name 116
How to configure our own template file at project level 116
DetailView 116
Django CRUD Operations 119
CreateView class 120
UpdateView class 121
DeleteView class 122
Django Class Based Views Complete Example-1(cbvproject7) 124
Django Class Based Views Complete Example-2(cbvfinalproject) 127
CRUD Operations by using Function Based Views(FBVs) 131
Differences between CBVs and FBVs 137

Chapter-11: Django ORM 138
Django ORM 139
How to find Query associated with QuerySet 139
How to filter records based on some condition 139
Various possible Field Look ups 139
How to implement OR Queries in Django ORM 141
How to implement AND Queries in Django ORM 141
How to implement NOT Queries in Django ORM 142
How to perform Union operation for query sets of the same or different models 142
How to select only some columns in the queryset 142
Aggregate Functions 143
How to Create, Update, Delete Records 144
How to add multiple records at a time 144
How to Delete a Single Record 144
How to Delete Multiple Records 144
How to Delete all Records(Truncate Operation in SQL) 144
How to Update Field of a Particular Record 145
How to Order queryset in Sorting Order 145

Chapter-12: Working with Advanced Model Concepts 146
Model Inheritance 147
Abstract Base Class Model Inheritance 147
Multi table Inheritance 148
Multilevel Inheritance 149
Multiple Inheritance 149
Model Manager 150

How to define our own Custom Manager 150
Proxy Model Inheritance 152

Chapter-13: Working with Django Middleware 155
Middleware 156
Middleware Structure 157
Demo Application for Custom Middleware Execution Flow 158
Execution Process for a Single Middleware Class 158
Middleware application to show information saying application under maintenance 159
Middleware application to show meaningful response if view function raises any error 160
Configuration of multiple middleware classes 161

Chapter-14: Deployment of our application in the Live Environment 163
Deployment 164
Need of Version Control Systems 164
Git vs GitHub 164
How to install git 165
How to create account in github.com 165
Git Respository 165
Activtities related to Git Repository 165
Deployment on pythonanywhere.com 169
Open the console and Create Virtual Environment 169
Which package already installed in this virtual env 169
install django 169
How to check whether django installed properly 170
Copy our application from github to our virtualenvironment (pythonanywhere) 170
Configuration on the web tab 171
WSGI Configuration 171
Add 'durgasoftdjango.pythonanywhere.com' to ALLOWED_HOSTS 172
Static Files 172
1. Static files related to admin site 172
2. Static files related to our application 172

Chapter-15: Real Time Project: Blog Application Development 173
How to specify dropdown choice field in models.py 174
slug field 174
ForeignKey 174
DateTimeField 174
pytz module(python timezone module) 175
Customization of Admin Interface 175
Implementation of List View and Detail View By using FBVs 176
Django Pagination 180
How Pagination works 180

Class Based Views For Pagination 181
Mail Sending Functionality 182
Adding Comments Section to the Blog 185
Adding Tags Functionality To our application 189
How to display similar posts 192
Custom Template Tags 194
3 Utility Functions to define Custom Template Tags 194
How to create custom template tag 195
Usage of simple_tag to return number of posts published 195
Usage of inclusion_tag to display Latest Posts 196
Usage of assignment_tag to display Most Commented Posts 197

Chapter-16: Introduction to Web application Development by using Flask 198
How to install Flask 199
How to Develop Application 199
How to Run Development Server 199
How to send Request 199

Chapter-1

Introduction To Web Development And Django

Web Application:

The applications which will provide services over the web are called web applications.
Eg: gmail.com, facebook.com, durgasoftvideos.com etc

Every web application contains 2 main components

1) Front-End
2) Back-End

1) Front-End:

- It represents what user is seeing on the website
- We can develop Front-End content by using the following technologies:
- HTML, JS, CSS, JQuery and BootStrap

- JQuery and Bootstrap are advanced front-end technologies, which are developed by using HTML, CSS and JavaScript only.

HTML:

- HyperText Markup Language
- Every web application should contain HTML. i.e HTML is the mandatory technology for web development.It represents structure of web page

CSS: Cascading Style Sheets

- It is optional technology, still every web application contains CSS.
- The main objective of CSS is to add styles to the HTML Pages like colors, fonts, borders etc.

Java Script:

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

- The main objective of Java Script is to add functionality to the HTML Pages. ie to add dynamic nature to the HTML Pages.

- HTML → Meant for Static Responses
- HTML+JS → Meant for Dynamic Responses

- Eg 1: To display "Welcome to DURGASOFT" response to the end user only HTML is enough,b'z it is static response.

- Eg 2: To display current server date and time to the end user, only HTML is not enough we required to use some extra technology like JavaScript,JSP,ASP,PHP etc as it is dynamic response.

Static Response vs Dynamic Response:

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

Eg: Login Page of Gmail
Home Page of ICICI Bank

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

Eg: Inbox Page of Gmail
Balance Page of ICICI Bank

2) Back End:

- It is the technology used to decide what to show to the end user on the Front-End.
- ie Backend is responsible to generate required response to the end user,which is displayed by the Front-End.

- Back-End has 3 important components:
  1) The Language like Java,Python etc
  2) The Framework like DJango,Pyramid,Flask etc
  3) The Database like SQLite,Oralce,MySQL etc

- For the Backend language Python is the best choice b'z of the following reasons: Simple and easy to learn, libraries and concise code.
- For the Framework DJango is the best choice b'z it is Fast, Secure and Scalable. Django is the most popular web application framework for Python.
- DJango provides inbuilt database which is nothing but SQLite, which is the best choice for database.
- The following are various popular web applications which are developed by using Python and DJango
- YouTube, Dropbox, Quora, Instagram, Reddit, Yahoo Maps etc

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/

Chapter-2

Django & Atom Installation
And
Development of First Web Application

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

ATOM IDE/Editor:

Install ATOM IDE from the following link → https://atom.io/

Speciality of ATOM IDE:

- It is freeware.
- It is open source.
- It supports cross platform.
- It provides several auto completion short-cuts for easy development etc

How to Configure Atom for Python:

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:

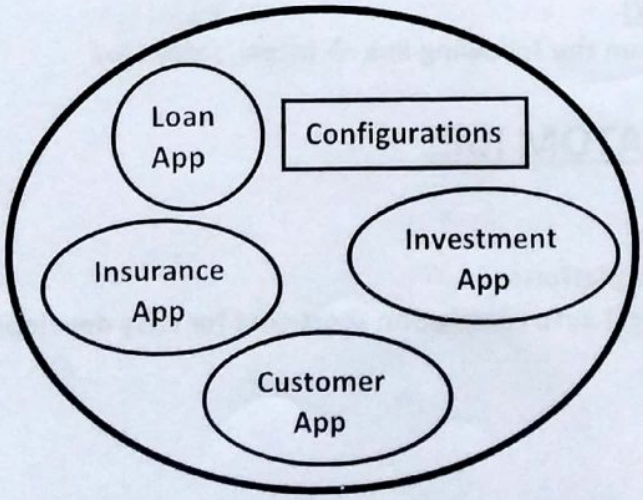
File → Settings → Install → In the searchbox just type terminal → platform-ide-terminal → settings → Shell Override

C:\Windows\System32\cmd.exe

Django 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



Bank Project

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 startserver
```

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 DURGASOFT 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.

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)  ]
```

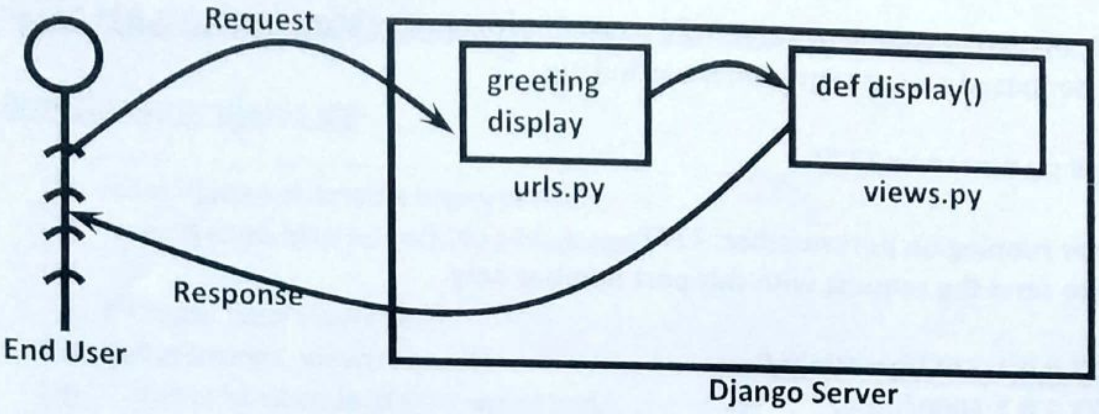
Whenever end user sending the request with urlpattern: greeting then disply() function will be executed and provide required response.

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:



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/

Various Practice Applications:

1. Write Django Application just to send Helloworld message as response.
2. Write Django application to send server time as response
3. Single application with multiple views

views.py:

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

# Create your views here.
def good_morning_view(request):
    return HttpResponse('<h1>Hello Friend Good Morning!!!</h1>')

def good_evening_view(request):
    return HttpResponse('<h1>Hello Friend Good Evening !!!</h1>')

def good_afternoon_view(request):
    return HttpResponse('<h1>Hello Friend Good Afternoon!!!</h1>')
```

urls.py

```
from django.conf.urls import url
from django.contrib import admin
from testapp import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^morning/', views.good_morning_view),
    url(r'^afternoon/', views.good_afternoon_view),
```

```
    url(r'^evening/', views.good_evening_view),
]
```

4. Single project with multiple applications:

greetingapp: views.py

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

# Create your views here.
def greetings_view(request):
    #total html code 1000 lines of code
    return HttpResponse('<h1>Hello Friends Good Morning...Have a Nice Day</h1')
```

timeapp:views.py

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

# Create your views here.
def time_info_view(request):
    time=datetime.datetime.now()
    s='<h1>Hello Current Date and Time is :'+str(time)+'</h1>'
    return HttpResponse(s)
```

Issue with urls.py

```
from greetingapp import views
from timeapp import views
```

Take special care while defining url patterns.we will get error b'z only one views.py is available. But we can solve this problem with any of the following 2 ways

First Approach

```
from greetingapp.views import greetings_view
from timeapp.views import time_info_view

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^greetings/',greetings_view),
    url(r'^time/',time_info_view),
```

```
8) ]
```

Second Approach

```
1) from greetingapp import views as v1
2) from timeapp import views as v2
3)
4) urlpatterns = [
5)     url(r'^admin/', admin.site.urls),
6)     url(r'^greetings/',v1.greetings_view),
7)     url(r'^time/',v2.time_info_view),
8) ]
```

Q. Is it possible to define mutlitple url patterns for the same view function?
Answer: Yes, by using the following approach

```
1) urlpatterns = [
2)     url(r'^admin/', admin.site.urls),
3)     url(r'^$', views.wish),
4)     url(r'^test/', views.wish)
5) ]
```

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

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 testapp

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

```
INSTALLED_APPS=[
    .........
        'testapp'
        ]
```

4. Define View Function in views.py

views.py

```
1) from django.shortcuts import render
2) from django.http import HttpResponse
3)
4) # Create your views here.
5) def first_view(request):
6)     return HttpResponse('<h1>Response from First View</h1>')
7)
8) def second_view(request):
9)     return HttpResponse('<h1>Response from Second View</h1>')
10)
11) def third_view(request):
12)     return HttpResponse('<h1>Response from third View</h1>')
13)
14) def fourth_view(request):
15)     return HttpResponse('<h1>Response from Fourth View</h1>')
16)
17) def fifth_view(request):
18)     return HttpResponse('<h1>Response from Fifth View</h1>')
```

5. Create a seperate urls.py file inside application

testapp/urls.py

```
1) from django.conf.urls import url
2) from testapp import views
```

```
3)
4) urlpatterns = [
5)     url(r'^first/', views.first_view),
6)     url(r'^second/', views.second_view),
7)     url(r'^third/', views.third_view),
8)     url(r'^fourth/', views.fourth_view),
9)     url(r'^fifth/', views.fifth_view),
10) ]
```

6. Include this application level urls.py inside project level urls.py file.
from django.conf.urls. import include

project level:urls.py

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

6. Run Server
py manage.py runserver

7. Send Request
http://127.0.0.1:8000/urlApp/test

Note: We can see reusability of application in other project just with only 2 lines addition

1. settings.py -->add application name
2. urls.py --->just add: url(r'^urlApp/',include('urlApp.urls'),

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

Chapter-3

Django Templates And Static Files

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:\durgaclasses\test.py

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

4) Inside D:\durgaclasses there 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:\durgaclasses\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:

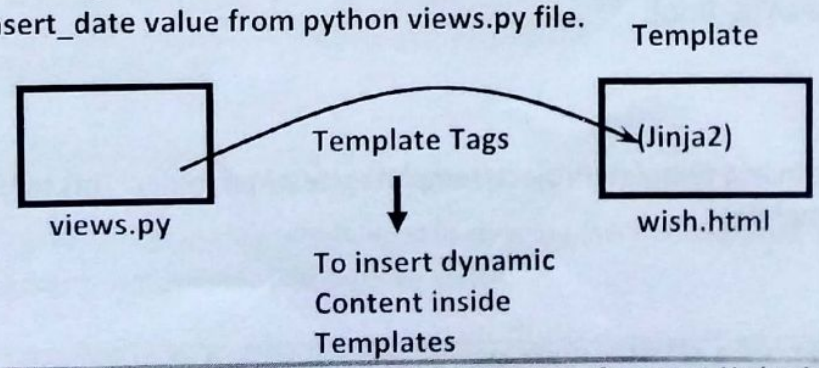
From Python views.py we can inject dynamic content to the template file by using template tags.

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.



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 display date, time and student information:

views.py

```
from django.shortcuts import render
import datetime

# Create your views here.
def template_view(request):
    dt=datetime.datetime.now()
    name='Durga'
    rollno=101
    marks=100
    my_dict={'date':dt,'name':name,'rollno':rollno,'marks':marks}
    return render(request,'testapp/results.html',my_dict)
```

results.html

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <title></title>
    <style>
      body{
        background: green;
        color:white;
      }
      h1{
        border:10px solid yellow;
      }
    </style>
  </head>
  <body>
    <h1>Hello this response from Template File.I will take care everything about pres
entation</h1><hr>
    <h2>The Server Date and Time is:{{date}}</h2>
    <ol>
      <li>Name:{{name}}</li>
      <li>Rollno:{{rollno}}</li>
      <li>Marks:{{marks}}</li>
    </ol>

  </body>
</html>
```

Application to wish End User based on Time

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)    #h11{
8)      color:red;
9)    }
10)   #h12{
11)     color:green;
12)   }
13)   </style>
14)  </head>
15)  <body>
16)   <h1 id=h11>{{insert_msg}}</h1>
17)   <h1 id=h12>Current Date and Time : {{insert_date}}</h1>
18)  </body>
19) </html>
```

views.py

```
1)  from django.shortcuts import render
2)  import datetime
3)
4)  # Create your views here.
5)  def wish(request):
6)     date=datetime.datetime.now()
7)     msg='Hello Guest !!!! Very Very Good '
8)     h=int(date.strftime('%H'))
9)     if h<12:
10)       msg +='Morning!!!'
11)    elif h<16:
12)       msg +='AfterNoon!!!'
13)    elif h<21:
14)       msg +='Evening!!!'
15)    else:
16)       msg='Hello Guest !!!! Very Very Good Night!!!'
17)    my_dict={'insert_date':date,'insert_msg':msg}
18)    return render(request,'testApp/wish.html',context=my_dict)
```

Templates Based Applications:

1. Application to Display Hello World
2. Application To display date and time
   (observe 3 ways of providing context dictionary)
3. Application To display date ,time and student information
4. Application to display wish message based on time like Good Morning,Good Evening etc

Note: Try to get Basic Idea about MVC and MVT wrt table and diagrams

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

```
1)  <!DOCTYPE html>
2)  {% load staticfiles %}
3)  <html lang="en" dir="ltr">
4)    <head>
5)      <meta charset="utf-8">
6)      <title>First Template Page</title>
7)      <style >
8)      #h11{
9)        color:red;
10)     }
11)     #h12{
12)       color:green;
13)     }
14)     </style>
15)   </head>
16)   <body>
17)     <h1 id=h11>{{insert_msg}}</h1>
18)     <h1 id=h12>Current Date and Time : {{insert_date}}</h1>
19)     <h1>This climate preferable image is:</h1>
20)     <img src="{% static "images/divine3.jpg" %}" alt="">
21)   </body>
22) </html>
```

views.py

```
1)  from django.shortcuts import render
2)  import datetime
3)
4)  # Create your views here.
5)  def wish(request):
6)      date=datetime.datetime.now()
7)      msg=None
8)      h=int(date.strftime('%H'))
9)      if h<12:
10)         msg='Hello Guest !!!! Very Very Good Morning!!!'
11)     elif h<16:
12)         msg='Hello Guest !!!! Very Very Good AfterNoon!!!'
13)     elif h<21:
14)         msg='Hello Guest !!!! Very Very Good Evening!!!'
15)     else:
16)         msg='Hello Guest !!!! Very Very Good Night!!!'
17)     my_dict={'insert_date':date,'insert_msg':msg}
18)     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

```
1) img{
2)   height: 500px;
3)   width: 500px;
4)   border: 10px red groove;
5)   margin:0% 20%;
6) }
7) h1{
8)   color:blue;
9)   text-align: center;
10) }
```

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" %}" >
```

DURGASOFT NEWS PORTAL:

index.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)   <body>
10)    <h1>Welcome to DURGASOFT NEWS PORTAL</h1>
11)    <ul>
12)      <li> <a href="/movies">Movies Information</a> </li>
13)      <li> <a href="/sports">Sports Information</a> </li>
14)      <li> <a href="/politics">Politics Information</a> </li>
15)    </ul>
16)  </body>
17) </html>
```

news.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)  <body>
10)    <h1>{{head_msg}}</h1>
11)    <ul>
12)     <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',
```

```
            'sub_msg2':'Rupee Value now 1$:70Rs',
            'sub_msg3':'In India Single Paisa Black money No more',
            }
    return render(request,'news.html',context=my_dict)
def index(request):
    return render(request,'index.html')
```

settings.py

```
import os

BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
TEMPLATES_DIR=os.path.join(BASE_DIR,'templates')
STATIC_DIR=os.path.join(BASE_DIR,'static')

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

TEMPLATES = [
    {
        '
        'DIRS': [TEMPLATES_DIR,],

            ],
        },
    },
]

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

urls.py

```
from django.conf.urls import url
from django.contrib import admin
from newsApp import views
urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^movies/', views.moviesInfo),
    url(r'^sports/', views.sportsInfo),
    url(r'^politics/', views.politicsInfo),
    url(r'^$', views.index),
]
```

Chapter-4

Working with Models And Databases

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',
5)         'USER':'scott',
6)         'PASSWORD':'tiger'
7)     }
8) }
```

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

```
1) from django.db import models
2)
3) # Create your models here.
4) class Employee(models.Model):
5)     eno=models.IntegerField()
6)     ename=models.CharField(max_length=30)
7)     esal=models.FloatField()
8)     eaddr=models.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)  def __str__(self):
11)      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)
```

Note:

We should do this registration in a single line otherwise we are getting error.

admin.site.register(Employee)

admin.site.register(EmployeeAdmin)

Practice Activity:

Create the following tables and populate with some sample data

1. Job(posting date,location,offeredsalary,qualification)
2. Book(number,title,author,publisheddate)
3. Customer(name,ano,mailid,phonenumber,age)

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

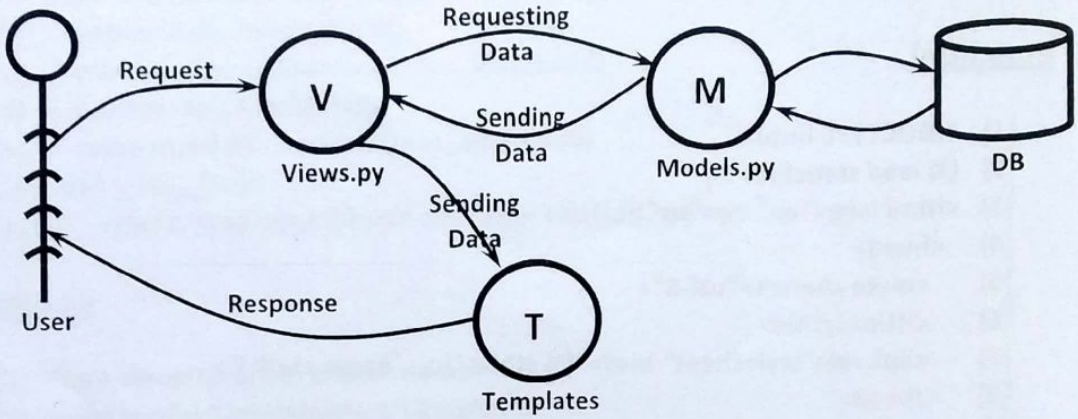
```
1) from django.shortcuts import render
2) from testApp.models import Employee
3)
4) # Create your views here.
5) def empdata(request):
6)     emp_list=Employee.objects.all()
7)     my_dict={'emp_list':emp_list}
8)     return render(request, 'testApp/emp.html', context=my_dict)
```

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>
29)     {% endfor %}
30)
31)   </table>
32)   {%else%}
```

```
33)    <p> No records found </p>
34)    {% endif %}
35)
36)  </body>
37) </html>
```

MVT Diagram



Important FAQs Related to Model and database configurations:

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.

```
1) from faker import Faker
2) from random import *
3) fakegen=Faker()
4) name=fakegen.name()
5) 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')))
```

Working with MYSQL Database (studentinfo project):

settings.py

```
TEMPLATE_DIR=os.path.join(BASE_DIR,'templates')
INSTALLED_APPS = [
    ....,
    'testapp'
]
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [TEMPLATE_DIR],
    ....
    }
]
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'studentdb',
        'USER':'root',
        'PASSWORD':'root'
    }
}
```

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()
```

admin.py

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

views.py

```
1) from django.shortcuts import render
2) from testapp.models import Student
3)
4) # Create your views here.
5) def studentview(request):
6)     student_list=Student.objects.order_by('marks')
7)     my_dict={'student_list':student_list}
8)     return render(request,'testapp/students.html',context=my_dict)
```

students.html

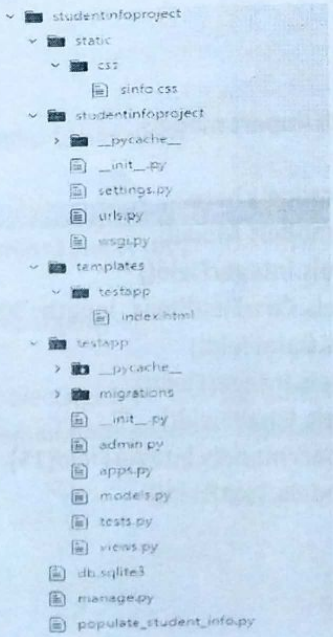
```
1) <!DOCTYPE html>
2) <html lang="en" dir="ltr">
3)   <head>
4)     <meta charset="utf-8">
5)     <title></title>
6)   </head>
7)   <body>
8)     {%if student_list %}
9)     <table border="1" margin='auto'>
10)      <thead>
11)        <th>Name</th>
12)        <th>Marks</th>
```

```
13)     </thead>
14)     {% for student in student_list %}
15)     <tr>
16)       <td>{{student.name}}</td>
17)       <td>{{student.marks}}</td>
18)     </tr>
19)     {% endfor %}
20)   </table>
21)   {% else %}
22)   <p>No Studetns found</p>
23)  {% endif %}
24)  </body>
25) </html>
```

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.studentview),
8)  ]
```

Working with MYSQL Database and Faker Module (studentinfo Project):



settings.py

```
TEMPLATE_DIR=os.path.join(BASE_DIR,'templates')
STATIC_DIR=os.path.join(BASE_DIR,'static')
INSTALLED_APPS = [
    ....,
    'testapp'
]
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [TEMPLATE_DIR],
    ....
    }
]
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'studentdb',
        'USER':'root',
        'PASSWORD':'root'
    }
}
STATIC_URL = '/static/'
STATICFILES_DIRS=[
STATIC_DIR,
]
```

models.py

```
from django.db import models

# Create your models here.
class Student(models.Model):
    rollno=models.IntegerField()
    name=models.CharField(max_length=30)
    dob=models.DateField()
    marks=models.IntegerField()
    email=models.EmailField()
    phonenumber=models.IntegerField(15)
    address=models.TextField()
```

admin.py

```
from django.contrib import admin
from testapp.models import Student

# Register your models here.
class StudentAdmin(admin.ModelAdmin):
    list_display=['rollno','name','dob','marks','email','phonenumber','address']

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

populate_student_info.py

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

import django
django.setup()

from testapp.models import Student
from faker import Faker
from random import *
fake=Faker()
def phonenumbergen():
    d1=randint(6,9)
    num=''+str(d1)
    for i in range(9):
        num=num+str(randint(0,9))
    return int(num)
def populate(n):
    for i in range(n):
        frollno=fake.random_int(min=1,max=999)
        fname=fake.name()
        fdob=fake.date()
        fmarks=fake.random_int(min=1,max=100)
        femail=fake.email()
        fphonenumber=phonenumbergen()
        faddress=fake.address()
        student_record=Student.objects.get_or_create(rollno=frollno,name=fname,dob=fdob,marks=fmarks,email=femail,phonenumber=fphonenumber,address=faddress)
populate(30)
```

views.py

```
1)  from django.shortcuts import render
2)  from testapp.models import Student
3)
4)  # Create your views here.
5)  def home_page_view(request):
6)      students=Student.objects.all()
7)      #students=Student.objects.filter(marks__lt=35)
8)      #students=Student.objects.filter(name__startswith='A')
9)      #students=Student.objects.all().order_by('marks')
10)     #students=Student.objects.all().order_by('-marks')
11)     return render(request,'testapp/index.html',{'students':students})
```

index.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="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7
    /css/bootstrap.min.css" integrity="sha384-
    BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" c
    rossorigin="anonymous">
8)      <link rel="stylesheet" href="{%static "css/sinfo.css"%}">
9)    </head>
10)   <body>
11)     <div class="container" >
12)       <h1>Student Information</h1><hr>
13)       {% if students%}
14)         {%for student in students%}
15)           <h2>{{student.name}} Information</h2><hr>
16)           <ul>
17)             <li>Student Rollno:{{student.rollno}}</li>
18)             <li>Student DOB:{{student.dob}}</li>
19)             <li>Student Marks:{{student.marks}}</li>
20)             <li>Student Email:{{student.email}}</li>
21)             <li>Student Phone Number:{{student.phonenumber}}</li>
22)             <li>Student Address:{{student.address}}</li>
23)           </ul>
24)           <br>
25)         {%endfor%}
26)       {%else%}
```

```
27)       <p>Student Records Are Not Available</p>
28)     {%endif%}
29)   </div>
30)  </body>
31) </html>
```

sinfo.css

```
1)  body .container{
2)    background: red;
3)    color:white;
4)  }
5)  h1{
6)    text-align: center;
7)  }
8)  h2{
9)    color:yellow;
10) }
```

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.home_page_view),
8)  ]
```

Chapter-5

Working with Django Forms

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:

```
1) from django import forms
2) class StudentForm(forms.Form):
3)     name=forms.CharField()
4)     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()
```

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

Alternative Short Way:

```
def studentinputview(request):
    form=forms.StudentForm()
    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

```
<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>
```

input.html:

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

```
15)         <input type="submit" class="btn btn-primary" name="" value="Submit">
16)     </form>
17)   </div>
18)  </body>
19) </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.

CSRF (Cross Site Request Forgery) Token:

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

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

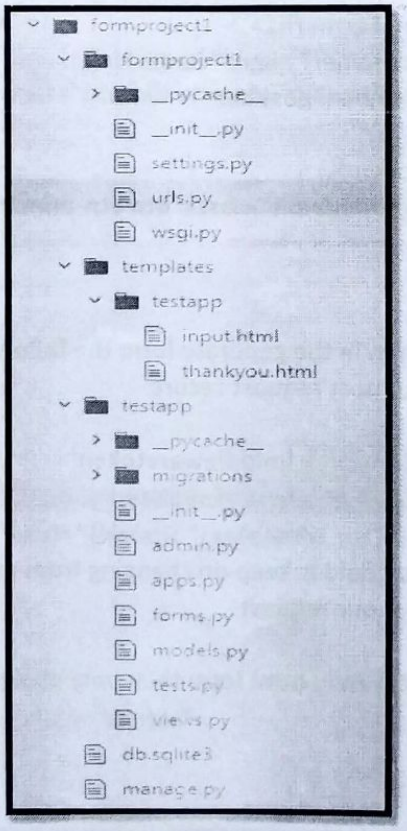
How to process Input Data from the form inside views.py File:

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})
```

Student Name and Marks Submission Form Project (formproject1):



forms.py

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

views.py

```
from django.shortcuts import render
from testapp.forms import StudentForm
# Create your views here.
def student_input_view(request):
    sent=False
    if request.method=='POST':
        form=StudentForm(request.POST)
        if form.is_valid():
            print('Form Validation Success and printing data')
            print('Name:',form.cleaned_data['name'])
            print('Marks:',form.cleaned_data['marks'])
            sent=True
    form=StudentForm()
    return render(request,'testapp/input.html',{'form':form,'sent':sent})
```

input.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/cs
s/bootstrap.min.css" integrity="sha384-
BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" c
rossorigin="anonymous">
  </head>
  <body>
    <div class="container" align='center'>
      {% if sent %}
        <h1>Thanks for Providing Information</h1>
        <h2>Enter Next Student Marks</h2>
      {%else%}
        <h1>Student Marks Submit Form</h1>
      {%endif%}<hr>
```

```
17)     <form method='POST'>
18)       {{form.as_p}}
19)       {%csrf_token%}
20)       <input type="submit" name="" class='btn btn-lg btn-
   primary' value="Submit Marks">
21)     </form>
22)   </div>
23)  </body>
24) </html>
```

thankyou.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>Thanks for providing your Name and Marks</h1>
9)    </body>
10) </html>
```

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'^input/', views.student_input_view),
8)  ]
```

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"
    crossorigin="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()
```

```
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.MaxLengthValidator(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 in Signup 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()
7)      email=forms.EmailField()
8)      feedback=forms.CharField(widget=forms.Textarea)
9)
10)     def clean(self):
11)         print('Total Form Validation by using single clean method...')
12)         total_cleaned_data=super().clean()
13)         inputname=total_cleaned_data['name']
14)         if inputname[0].lower() !='d':
15)             raise forms.ValidationError('Name should starts with d|D')
16)         inputrollno=total_cleaned_data['rollno']
17)         if inputrollno<=0:
18)             raise forms.ValidationError('Rollno should be >0')
```

```
class SignupForm(forms.Form):
    name=forms.CharField()
    password=forms.CharField(widget=forms.PasswordInput)
    rpassword=forms.CharField(label='Reenter Password',widget=forms.PasswordInput)
    email=forms.EmailField()

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

views.py

```
from django.shortcuts import render
from testapp.forms import FeedbackForm,SignupForm

# Create your views here.
def feedback_form_view(request):
    form=FeedbackForm()
    # feedback_submitted=False
    if request.method=='POST':
        form=FeedbackForm(request.POST)
        if form.is_valid():
            print('Basic Validation Completed and Printing Data')
            print('Name:',form.cleaned_data['name'])
            print('RollNo:',form.cleaned_data['rollno'])
            print('Email:',form.cleaned_data['email'])
            print('Feedback:',form.cleaned_data['feedback']) # feedback_submitted=True
    return render(request,'testapp/feedback.html',{'form':form})

def signup_form_view(request):
    form=SignupForm()
    if request.method=='POST':
        form=SignupForm(request.POST)
        if form.is_valid():
            print('Basic Validation Completed and Printing Data')
            print('Name:',form.cleaned_data['name'])
            print('Password:',form.cleaned_data['password'])
            print('Email:',form.cleaned_data['email'])
    return render(request,'testapp/signup.html',{'form':form})
```

signup.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-
   BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" c
   rossorigin="anonymous">
8)   </head>
9)   <body>
10)   <div class="container" align='center'>
11)       <h1>Student Signup Form</h1><hr>
12)       <form method="post">
13)         {{form.as_p}}
14)         {%csrf_token%}
15)         <input type="submit" class='btn btn-lg btn-warning' value="Signup">
16)       </form>
17)   </div>
18) </body>
19) </html>
```

Note: If we want to perform validations by comparing multiple input fields then we should go for single clean() method.

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.

```
from django import forms
from django.core import validators

class FeedBackForm(forms.Form):
    name=forms.CharField()
    password=forms.CharField(widget=forms.PasswordInput)
    rpassword=forms.CharField(label='Re Enter Password',
widget=forms.PasswordInput)
    rollno=forms.IntegerField()
    email=forms.EmailField()
    feedback=forms.CharField(widget=forms.Textarea)
    bot_handler=forms.CharField(required=False,widget=forms.HiddenInput)

    def clean(self):
        print('validating passwords match...')
        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

Chapter-6

Working with Django Model Forms

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:

```
from django.db import models

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

forms.py:

```
from django import forms
from testapp.models import Student
class StudentForm(forms.ModelForm):
    #fields with validations
    class Meta:
```

```
    model=Student
    fields='__all__'
```

views.py:

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

# Create your views here.
def student_view(request):
    form=forms.StudentForm
    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

models.py

```
from django.db import models

# Create your models here.
class Movie(models.Model):
    rdate=models.DateField()
    moviename=models.CharField(max_length=30)
    hero=models.CharField(max_length=30)
    heroine=models.CharField(max_length=30)
    rating=models.IntegerField()
```

admin.py

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

forms.py

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

views.py

```
1) from django.shortcuts import render
2) from testapp.forms import MovieForm
3) from testapp.models import Movie
4)
5) # Create your views here.
6) def index_view(request):
7)     return render(request,'testapp/index.html')
8)
9) def add_movie_view(request):
10)     form=MovieForm()
11)     if request.method=='POST':
12)         form=MovieForm(request.POST)
13)         if form.is_valid():
14)             form.save()
15)         return index_view(request)
16)     return render(request,'testapp/addmovie.html',{'form':form})
17)
18) def list_movie_view(request):
19)     movies_list=Movie.objects.all().order_by('-rating')
20)     return render(request,'testapp/listmovie.html',{'movies_list':movies_list})
```

index.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="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7
/css/bootstrap.min.css" integrity="sha384-
BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" c
rossorigin="anonymous">
8)     <link rel="stylesheet" href="{% static "css/movie.css"%}">
```

```
9)    <link href="https://fonts.googleapis.com/css?family=Merriweather" rel="stylesh
   eet">
10)  </head>
11)  <body>
12)   <div class="container" align="center">
13)    <div class="jumbotron">
14)       <h1>DURGA Movies...</h1>
15)       <p>up to date movies information</p>
16)       <a class="btn btn-primary btn-lg"
   href="/addmovie" role="button">Add Movie</a>
17)       <a class="btn btn-primary btn-lg"
   href="/listmovies" role="button">List Movies</a>
18)    </div>
19)   </div>
20)  </body>
21) </html>
```

listmovie.html

```
1)  <!DOCTYPE html>
2)  {% load cust_filters%}
3)  {%load staticfiles%}
4)  <html lang="en" dir="ltr">
5)   <head>
6)    <meta charset="utf-8">
7)    <title></title>
8)    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7
   /css/bootstrap.min.css" integrity="sha384-
   BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" c
   rossorigin="anonymous">
9)    <link rel="stylesheet" href="{% static "css/movie.css"%}">
10)   <link href="https://fonts.googleapis.com/css?family=Merriweather"
   rel="stylesheet">
11)  </head>
12)  <body>
13)   <h1>Movies Information</h1>
14)   {% if movies_list %}
15)    <table>
16)     <thead>
17)       <th>Released Date</th>
18)       <th>Movie Name</th>
19)       <th>Hero</th>
20)       <th>Heroine</th>
21)       <th>Rating</th>
22)     </thead>
```

```
23)      {% for movie in movies_list%}
24)        <tr>
25)          <td>{{movie.rdate}}</td>
26)          <td>{{movie.moviename}}</td>
27)          <td>{{movie.hero}}</td>
28)          <td>{{movie.heroine}}</td>
29)          <td>{{movie.rating}}</td>
30)        </tr>
31)       {% endfor %}
32)     </table>
33)    {% else %}
34)     <p>No Movies Found</p>
35)    {%endif%}
36)  </body>
37) </html>
```

addmovie.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="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7
    /css/bootstrap.min.css" integrity="sha384-
    BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
    crossorigin="anonymous">
8)      <link rel="stylesheet" href="{% static "css/movie.css"%}">
9)      <link href="https://fonts.googleapis.com/css?family=Merriweather"
    rel="stylesheet">
10)   </head>
11)   <body>
12)    <div class="container" align='center'>
13)      <h1>Add Movie Information</h1><hr>
14)      <form  method="post">
15)        {{form.as_p}}
16)        {%csrf_token%}
17)        <input type="submit" class='btn btn-primary' value="Add Movie">
18)      </form>
19)    </div>
20)   </body>
21) </html>
```

movie.css

```
1) body{
2)   background: url(https://images.unsplash.com/photo-1513366208864-
     87536b8bd7b4?ixlib=rb-
     0.3.5&ixid=eyJhcHBfaWQiOjEyMDd9&s=b04402d9c23edcc4b321db9fd9be60a7&au
     to=format&fit=crop&w=334&q=80);
3)   /* background-repeat: no-repeat;
4)   background-size: cover; */
5) }
6) .container .jumbotron{
7)   margin-top:200px;
8)   background: orange;
9)   color:white;
10)  font-family: 'Merriweather', serif;
11) }
12) table,tr,td,thead,th{
13)  border:2px solid white;
14)  font-size: 30px;
15)  color:white;
16)  background:red;
17)  margin:auto;
18) }
19) thead,th{
20)  border:2px solid red;
21)  font-size: 30px;
22)  color:blue;
23)  margin:auto;
24)  background:yellow;
25) }
26) h1{
27)  text-align: center;
28) }
```

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.index_view),
8)     url(r'^addmovie/', views.add_movie_view),
9)     url(r'^listmovies/', views.list_movie_view),
```

How to Add Date Widget:

write a DateInput class in forms.py as:

```
class DateInput(forms.DateInput):
  input_type = 'date'
```

--> Now define ModelForm class as:

```
class MoviesForm(forms.ModelForm):
  movie_name=forms.CharField(widget=forms.TextInput(attrs={'class':'form-control'}))
  release_date=forms.DateField(widget=DateInput(attrs={'class':'form-control'}))
  hero_name=forms.CharField(widget=forms.TextInput(attrs={'class':'form-control'}))
  heroine_name=forms.CharField(widget=forms.TextInput(attrs={'class':'form-control'}))
  director_name=forms.CharField(widget=forms.TextInput(attrs={'class':'form-control'}))
  movie_ratings=forms.FloatField(widget=forms.TextInput(attrs={'class':'form-control'}))

  class Meta:
      model=MoviesModel

 fields=['movie_name','release_date','hero_name','heroine_name','director_name','movie
_ratings']
```

--> Here 'class':'form-control' is a bootstrap class used to format the date field

--> link for DateInput class https://docs.djangoproject.com/en/2.1/ref/forms/widgets/#dateinp

Chapter-7

Working with Advanced Template Features

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

```
1)  <!DOCTYPE html>
2)  html,css,bootstrap links
3)  <body>
4)    common code required for every child tempalte
5)    {% block child_block%}
6)      Anything outside of this block available to child tag.
7)      in child template the specific code should be in this block
8)    {%endblock%}
9)  </body>
10) </html>
```

child.html

```
1)  <!DOCTYPE html>
2)  {%extends 'testapp/base.html'%}
3)    {% block child_block %}
4)      child specific extra code
5)    {%endblock%}
```

```
body{
  background: url(https://images.unsplash.com/photo-1460650671472-ba449b1a6f10?ixlib=rb-0.3.5&ixid=eyJhcHBfaWQiOjEyMDd9&s=daeb797a01124d5e37bdb0f543314f90&auto=format&fit=crop&w=666&q=80);
  background-repeat: no-repeat;
  background-size: cover;

}
.navbar{
  background:orange;
}
.navbar a{
  color:white;
}
img{
  height: 200px;
  width: 30%;
  margin:1.5%;
  border:2px orange solid;
}
#home{
  font-size: 100px;
  font-weight: 900;
  color:white;
  text-align: center;
}
h1{
  font-size: 40px;
  font-weight: 900;
  color:white;
  text-align: center;
}
li{
  color:yellow;
  font-size: 30px;
}
```

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.

How to Add Seperate CSS Files to Child Templates?

base.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="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7
/css/bootstrap.min.css" integrity="sha384-
BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
crossorigin="anonymous">
8)   <link rel="stylesheet" href="{% static "css/advtempot.css"%}">
9)   {% block xyz %}{% endblock %}
10) </head>
11) <body>
12)  <nav class="navbar navbar-default navbar-inverse">
13) <div class="container">
14)  <!-- Brand and toggle get grouped for better mobile display -->
15)  <div class="navbar-header">
16)   <a class="navbar-brand" href="/">DURGA NEWS</a>
17)  </div>
18)   <ul class="nav navbar-nav">
19)    <li><a href="/">Home</a></li>
20)    <li><a href="/movies">Movies</a></li>
21)    <li><a href="/sports">Sports</a></li>
22)    <li><a href="/politics">Politics</a></li>
23)   </ul>
24) </div><!-- /.container-fluid -->
25) </nav>
26) {% block body_block %}
27) {% endblock %}
28) </body>
29) </html>
```

child.html

```
1) <!DOCTYPE html>
2) {% extends 'testapp/base.html'%}
3) {%load staticfiles%}
4) {% block xyz %}
5) <link rel="stylesheet" href="{%static "css/new.css" %}">
6) {%endblock%}
7)   {% block body_block %}
8)    <h1>Welcome to DURGA NEWS PORTAL</h1>
9)   {% endblock %}
```

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.

Example-2

cust_filters.py

```
1)  from django import template
2)  register=template.Library()
3)
4)  @register.filter(name='truncate5')
5)  def truncate5(value):
6)     result=value[0:5]
7)     return result
8)
9)  @register.filter(name='t_n')
10) def truncate_n(value,n):
11)    result=value[0:n]
12)    return 'durga'
```

lower.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)      {%load cust_filters%}
9)      {% for r in data_list%}
10)     <ul>
11)       <li>{{r.name|t_n:6}}</li>
12)       <li>{{r.subject|t_n:3}}</li>
13)       <li>{{r.dept|t_n:2}}</li>
14)       <li>{{r.date|timesince}}</li>
15)       <li>{{r.date|date:"d-m-Y"}}</li>
16)     </ul><hr>
17)     {%endfor%}
18)   </body>
19) </html>
```

3) Template Tags for URLs:

- <a href='testapp1/thankyou'>Thank You </a>
- <a href="{% url 'thankyou' %}">Thank You </a>
- <a href="{% url 'testapp.views.thankyou' %}">Thank You </a>
- <a href="{% url 'testapp:thankyou' %}">Thank You </a>

Chapter-8

Session Management

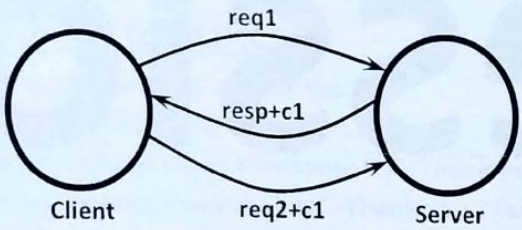
Session Management:

- 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.



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.

Application-1 Session Management by using Cookies (PageCount Application)

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>
```

Application-2 Session Management by using Cookies

```
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>
```

```
    <h1>Welcome to DURGASOFT</h1>
    <form  action="/second">
      {{form.as_p}}
      {%csrf_token%}
      <input type="submit" name="" value="Enter Name">
    </form>
  </body>
</html>
```

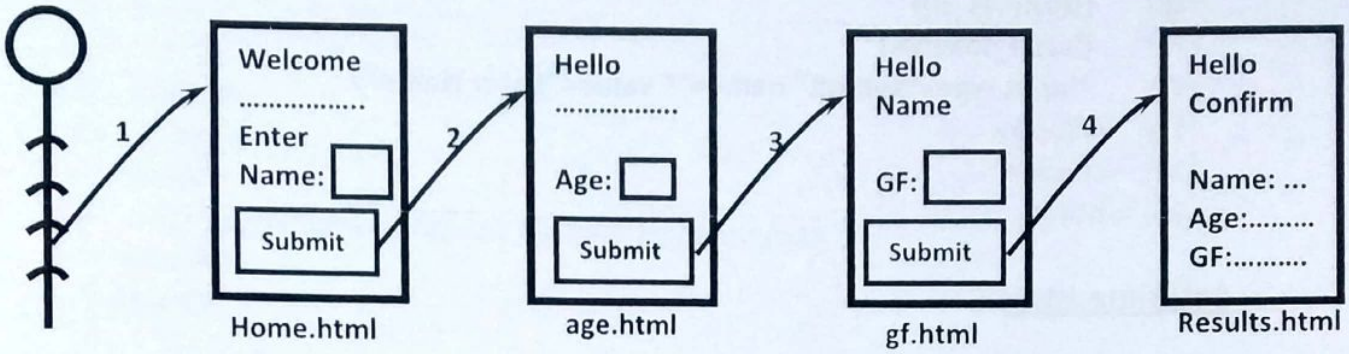
datetime.html

```
<!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>
```

Application-3 Session Management by using Cookies



views.py

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

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>Welcome to DURGASOFT</h1>
9)      <form action='/age'>
10)       Enter Name: <input type="text" name="name" value=""><br><br>
11)       <input type="submit" name="" value="Submit Name">
12)     </form>
13)   </body>
14) </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)        <form action='/results'>
```

```
    Enter Girl Friend Name: <input type="text" name="gfname" value=""><br><br>
     <input type="submit" name="" value="Submit GFName">
    </form>
  </body>
</html>
```

results.html

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h1>Hello {{name}} Thanks for providing info</h1>
    <h2>Please cross check your data and confirm</h2><hr>
    <ul>
      <li>Name:{{name}}</li>
      <li>Age:{{age}}</li>
      <li>Girl Friend Name:{{gfname}}</li>
    </ul>
  </body>
</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.

Application-4 Session Management by using Cookies (Shopping Cart Application)

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)   </div>
17)  </body>
18) </html>
```

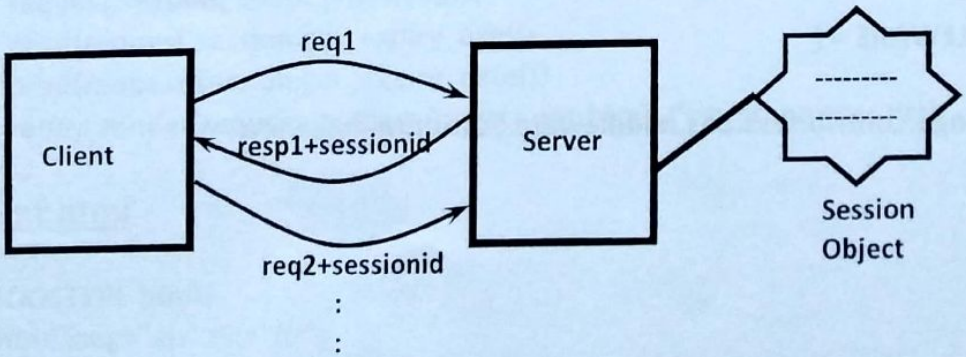
additem.html

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

Session Management By using Session API (Django Session Framework):



- 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

Application-1 Session Management by using Session API (PageCount Application)

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

Application-2 Session Management by using Session API (Profile Application)

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):
```

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

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

name.html

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h1>Name Registration Form</h1><hr>
    <form  action="/age" >
      {{form}}
      {%csrf_token%}<br><br>
      <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

```
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>Girl Friend Registration Form</h1><hr>
9)      <form  action="/results" >
10)       {{form}}
11)       {%csrf_token%}<br><br>
12)       <input type="submit" name="" value="Submit GF Info">
13)     </form>
14)   </body>
15) </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)      {%if request.session %}
9)      <h1>Thanks for providing information..Plz confirm once</h1>
10)     <ul>
11)       {%for key,value in request.session.items %}
12)       <li> <h2>{{key|upper}}:    {{value|title}}</h2> </li>
13)       {%endfor%}
14)     </ul>
15)     {%else%}
16)     <p>No Information available</p>
17)     {%endif%}
18) </body>
19) </html>
```

Application-3 Session Management by using Session API (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">
```

```
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>
```

displayitems.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>Your Shopping Cart Information:</h1>
9)    {%if request.session %}
10)   <table border=2>
11)    <thead>
12)     <th>Item Name</th>
13)     <th>Quantity</th>
14)    </thead>
15)    {% for key,value in request.session.items %}
16)    <tr>
17)     <td>{{key}}</td>
18)     <td>{{value}}</td>
19)    </tr>
20)    {% endfor %}
21)    </table>
22)   {%else%}
23)    <p>No Items in your shopping cart</p>
```

```
24)   {%endif%}
25) </body>
26) </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

```
1) if request.method=='POST':
2)       name=request.POST['name']
3)       quantity=request.POST['quantity']
4)       request.session[name]=quantity
5)       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.

Chapter-9

User Authentication And Authorization

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.

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

Authentication Application:

views.py

```
1)  from django.shortcuts import render
2)  from django.contrib.auth.decorators import login_required
3)
4)  # Create your views here.
5)  def home_page_view(request):
6)      return render(request,'testapp/home.html')
7)
8)  @login_required
9)  def java_exams_view(request):
10)     return render(request,'testapp/javaexams.html')
11) @login_required
12) def python_exams_view(request):
13)     return render(request,'testapp/pythonexams.html')
14) @login_required
15) def aptitude_exams_view(request):
16)     return render(request,'testapp/aptitudeexams.html')
17)
18) def logout_view(request):
19)     return render(request,'testapp/logout.html')
```

urls.py

```
1)  from django.conf.urls import url,include
2)  from django.contrib import admin
3)  from testapp import views
4)
5)  urlpatterns = [
6)      url(r'^admin/', admin.site.urls),
7)      url('accounts/',include('django.contrib.auth.urls')),
8)      url(r'^$', views.home_page_view),
9)      url(r'^python/', views.python_exams_view),
10)     url(r'^java/', views.java_exams_view),
```

```
11)   url(r'^aptitude/', views.aptitude_exams_view),
12)   url(r'^logout/', views.logout_view),
13) ]
```

auth123.css

```
1)  body{
2)    background: #efb917;
3)    color:blue;
4)  }
5)  .jumbotron{
6)    background: green;
7)    color:white;
8)  }
```

base.html

```
1)  <!DOCTYPE html>
2)  {%load staticfiles%}
3)  <html lang="en" dir="ltr">
4)    <head>
5)      <meta charset="utf-8">
6)      <title></title>
7)      <!-- Latest compiled and minified CSS -->
8)  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/cs
    s/bootstrap.min.css" integrity="sha384-
    BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
    crossorigin="anonymous">
9)  <link rel="stylesheet" href="{%static "css/auth123.css"%}">
10) </head>
11) <body>
12)   <nav class='navbar navbar-default navbar-inverse'>
13)     <div class="container">
14)       <div class="navbar-header">
15)         <a class="navbar-brand" href="/">DURGAEXAMS</a>
16)       </div>
17)       <ul class='nav navbar-nav'>
18)         <li> <a href="/java">Java Exams</a> </li>
19)         <li> <a href="/python">Python Exams</a> </li>
20)         <li> <a href="/aptitude">Aptitude Exams</a> </li>
21)       </ul>
22)       <ul class='nav navbar-nav navbar-right'>
23)         <li> <a href="#">Signup</a> </li>
24)         <li> <a href="/accounts/login/">Login</a> </li>
25)         <li> <a href="/accounts/logout/">Logout</a> </li>
```

```
26)      </ul>
27)     </div>
28)    </nav>
29)    {%block body_block%}
30)    {%endblock%}
31)  </body>
32) </html>
```

home.html

```
1)  <!DOCTYPE html>
2)  {%extends 'testapp/base.html'%}
3)      {%block body_block%}
4)      <div class="jumbotron" >
5)        <div class="container">
6)          <h1>Welcome to Durga Exams</h1>
7)          <h2>Rules:</h2>
8)          <ul>
9)            <li>Rule-1:You should write only one exam per day</li>
10)           <li>Rule-1:You should write only one exam per day</li>
11)           <li>Rule-1:You should write only one exam per day</li>
12)           <li>Rule-1:You should write only one exam per day</li>
13)           <li>Rule-1:You should write only one exam per day</li>
14)           <li>Rule-1:You should write only one exam per day</li>
15)           <li>Rule-1:You should write only one exam per day</li>
16)         </ul>
17)       </div>
18)     </div>
19)     {%endblock%}
```

javaexams.html

```
1)  <!DOCTYPE html>
2)  {%extends 'testapp/base.html'%}
3)      {%block body_block%}
4)      <div class="jumbotron" >
5)        <div class="container" align='center'>
6)          <h1>Welcome to Java Exams</h1>
7)        </div>
8)      </div>
9)      {%endblock%}
```

pythonexams.html

```
1) <!DOCTYPE html>
2) {%extends 'testapp/base.html'%}
3)     {%block body_block%}
4)     <div class="jumbotron" >
5)       <div class="container" align='center'>
6)         <h1>Welcome to Python Exams</h1>
7)       </div>
8)     </div>
9)     {%endblock%}
```

aptitudeexams.html

```
1) <!DOCTYPE html>
2) {%extends 'testapp/base.html'%}
3)     {%block body_block%}
4)     <div class="jumbotron" >
5)       <div class="container" align='center'>
6)         <h1>Welcome to Aptitude Exams</h1>
7)       </div>
8)     </div>
9)     {%endblock%}
```

registration/login.html

```
1) <!DOCTYPE html>
2) {%extends 'testapp/base.html'%}
3)     {%block body_block%}
4)      <div class="container" align='center'>
5)        <h1>Login to write Exams...</h1>
6)        <form  method="post">
7)          {{form.as_p}}
8)          {%csrf_token%}
9)          <button type="submit">Login</button>
10)       </form>
11)     </div>
12)     {%endblock%}
```

testapp/logout.html

```
1) <!DOCTYPE html>
2) {%extends 'testapp/base.html'%}
3)     {%block body_block%}
4)     <div class="jumbotron" >
```

```
5)      <div class="container">
6)        <h1>Thanks for visiting Durgaexams</h1>
7)        <p>Anyway we are feeling very sad b'z you logout atleast 24 hours per day
    you have to spend with our website like facebook</p>
8)        <h2>Please Login Again:</h2>
9)        <a href="/accounts/login" class="btn btn-primary btn-lg btn-success">
    Login</a>
10)     </div>
11)    </div>
12)    {%endblock%}
```

settings.py

```
1) LOGIN_REDIRECT_URL='/'
2) LOGOUT_REDIRECT_URL='/logout'
```

Configure Signup Form:

forms.py

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

views.py

```
1) def signup_view(request):
2)     form=SignUpForm()
3)     if request.method=='POST':
4)         form=SignUpForm(request.POST)
5)         user=form.save()
6)         user.set_password(user.password)
7)         user.save()
8)         return HttpResponseRedirect('/accounts/login')
9)     return render(request,'testapp/signup.html',{'form':form})
```

urls.py

```
1) urlpatterns = [
2)     ....
3)     url(r'^signup/', views.signup_view),
```

```
4) ]
```

Password Hashers in settings.py

```
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) ]
```

Future Assignments:

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

Chapter-10

Class Based Views And CRUD Operations By Using Both CBVs And FBVs

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

```
from django.views.generic import View
from django.http import HttpResponse

# Create your views here.
class HelloWorldView(View):
    def get(self,request):
        return HttpResponse('<h1>This is from ClassBasedView</h1>')
```

urls.py

```
from testapp import views

urlpatterns = [
    ...
    url(r'^$', views.HelloWorldView.as_view()),
]
```

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>
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-
   BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" c
   rossorigin="anonymous">
8)   </head>
9)   <body>
10)   <div class="container" >
11)     {%block body_block%}
12)     {%endblock %}
13)   </div>
14)  </body>
15) </html>
```

company_list.html

```
1) <!DOCTYPE html>
2) {%extends 'testapp/base.html'%}
3)       {%block body_block%}
4)         <h1>List of All Companies</h1><hr>
5)         <ol>
```

```
6)          {%for company in company_list%}
7)        <h2><li> <a href="/{{company.id}}">{{company.name|upper}}</a> </li></h2>
8)          {%endfor%}
9)         </ol>
10)       {%endblock %}
```

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)       {%endblock %}
```

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()),
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):
```

```
    return self.name

class Employee(models.Model):
    eno=models.IntegerField()
    name=models.CharField(max_length=128)
    salary=models.FloatField()
    company=models.ForeignKey(Company,related_name='employees')
```

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>
        <h2>Employee Information</h2>
        {%for emp in company.employees.all%}
         <ul>
          <li>Employee Number:{{emp.eno}}</li>
          <li>Employee Name:{{emp.name}}</li>
          <li>Employee Salary:{{emp.salary}}</li> <hr>
         </ul>
         {%endfor%}
      </ol>
{%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) def get_absolute_url(self):
12)        return reverse('detail',kwargs={'pk':self.pk})
```

UpdateView class:

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 class:

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 %}
```

Django Class Based Views Complete Example-1 (cbvproject7)

models.py

```
from django.db import models
from django.core.urlresolvers import reverse

# Create your models here.
class Beer(models.Model):
    name=models.CharField(max_length=128)
    taste=models.CharField(max_length=128)
    color=models.CharField(max_length=128)
    price=models.FloatField()
    def __str__(self):
        return self.name

    def get_absolute_url(self):
        return reverse('detail',kwargs={'pk':self.pk})
```

admin.py

```
from django.contrib import admin
from testapp.models import Company

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

views.py

```
from django.shortcuts import render
from testapp.models import Company
from django.core.urlresolvers import reverse_lazy
from django.views.generic import ListView,DetailView,CreateView,UpdateView,
DeleteView

# Create your views here.
class CompanyListView(ListView):
    model=Company
    #default template_name is company_list.html
    #defult context_object_name is company_list
class CompanyDetailView(DetailView):
```

```
    model=Company
    #default template_name is company_detail.html
    #defult context_object_name is company or object
class CompanyCreateView(CreateView):
    model=Company
    fields=('name','location','ceo')

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

class CompanyDeleteView(DeleteView):
    model=Company
    success_url=reverse_lazy('companies')
```

urls.py

```
from django.conf.urls import url
from django.contrib import admin
from testapp import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^companies/', views.CompanyListView.as_view(),name='companies'),
    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/cs
s/bootstrap.min.css" integrity="sha384-
BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
crossorigin="anonymous">
  </head>
<body>
```

```
11)   <div class="container" >
12)     {%block body_block%}
13)     {%endblock %}
14)   </div>
15)  </body>
16) </html>
```

company_list.html

```
1)  <!DOCTYPE html>
2)  {%extends 'testapp/base.html'%}
3)        {%block body_block%}
4)          <h1>List of All Companies</h1><hr>
5)          <ol>
6)            {%for company in company_list%}
7)            <h2><li> <a href="/{{company.id}}">{{company.name|upper}}</a> </li></h2>
8)            {%endfor%}
9)          </ol>
10)       {%endblock %}
```

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)         <a href="/delete/{{company.id}}" class='btn btn-danger'>Delete</a>
12)       {%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">
```

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

company_confirm_delete.html

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

Django Class Based Views Complete Example (cbvfinalproject)

models.py

```
from django.db import models
from django.core.urlresolvers import reverse

# Create your models here.
class Beer(models.Model):
    name=models.CharField(max_length=128)
    taste=models.CharField(max_length=128)
    color=models.CharField(max_length=128)
    price=models.FloatField()

    def __str__(self):
        return self.name

    def get_absolute_url(self):
        return reverse('detail',kwargs={'pk':self.pk})
```

admin.py

```
from django.contrib import admin
from testapp.models import Beer

# Register your models here.
class BeerAdmin(admin.ModelAdmin):
    list_display=['name','taste','color','price']
```

```
7) admin.site.register(Beer,BeerAdmin)
```

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

```
from django.shortcuts import render,redirect
from testapp.forms import EmployeeForm
from testapp.models import Employee

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

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

models.py

```
from django.db import models

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

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) # Create your models here.
3) class Employee(models.Model):
4)     eno=models.IntegerField()
5)     ename=models.CharField(max_length=64)
6)     esal=models.FloatField()
7)     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()
```

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

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

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

urls.py

```
from django.conf.urls import url
from django.contrib import admin
from testapp import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^$', views.show_view),
    url(r'^insert/', views.insert_view),
    url(r'^delete/(?P<id>\d+)/$', views.delete_view),
    url(r'^update/(?P<id>\d+)/$', views.update_view),
]
```

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-
```

```
8)  BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" c
    rossorigin="anonymous">
9)   </head>
10)  <body>
11)   <div class="container" align='center'>
12)    {%block body_block%}
13)    {%endblock%}
14)   </div>
15)  </body>
16) </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)     {%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.ename}}"
    ><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.eaddr}}"
    > <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.

Chapter-11

Django ORM

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:

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) <u>contains → Case-sensitive Containment Test</u>

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

SQL equivalent:
SELECT ... WHERE headline LIKE '%Lennon%';

4) <u>icontains → Case-insensitive Containment Test</u>

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

5) <u>in</u>

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

<u>Examples:</u>
Entry.objects.filter(id__in=[1, 3, 4])
Entry.objects.filter(headline__in='abc')

6) <u>gt</u>

Greater than
<u>Example:</u> Entry.objects.filter(id__gt=4)

7) <u>gte</u>

Greater than or equal to.

8) <u>lt</u>

Less than.

9) <u>lte</u>

Less than or equal to.

10) <u>startswith</u>

Case-sensitive starts-with.
<u>Example:</u> Entry.objects.filter(headline__startswith='Lennon')

11) <u>istartswith</u>

Case-insensitive starts-with.
<u>Example:</u> Entry.objects.filter(headline__istartswith='Lennon')

12) <u>endswith</u>

Case-sensitive ends-with.
<u>Example:</u> Entry.objects.filter(headline__endswith='Lennon')

13) iendswith

Case-insensitive ends-with.

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

14) range

Range test (inclusive)

```
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

```
from django.shortcuts import render
from django.db.models import Q
from django.db.models import Avg,Sum,Max,Min,Count
from testapp.models import Employee

# Create your views here.
def display_view(request):
    avg1=Employee.objects.all().aggregate(Avg('esal'))
    max=Employee.objects.all().aggregate(Max('esal'))
    min=Employee.objects.all().aggregate(Min('esal'))
    sum=Employee.objects.all().aggregate(Sum('esal'))
    count=Employee.objects.all().aggregate(Count('esal'))
    my_dict={'avg':avg1,'max':max,'min':min,'sum':sum,'count':count}
    return render(request,'testapp/aggregate.html',my_dict)
```

aggregate.html

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

Chapter-12

Working with Advanced Model Concepts

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

```
class Student(models.Model):
    name=models.CharField(max_length=64)
    email=models.EmailField()
    address=models.CharField(max_length=256)
    rollno=models.IntegerField()
    marks=models.IntegerField()

class Teacher(models.Model):
    name=models.CharField(max_length=64)
    email=models.EmailField()
    address=models.CharField(max_length=256)
    subject=models.CharField(max_length=64)
    salary=models.FloatField()
```

With Inheritance

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

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)
```

3) Multi Level Inheritance:

Inheritance at multiple levels.

```
1)  class Person(models.Model):
2)     name=models.CharField(max_length=64)
3)     age=models.IntegerField()
4)
5)  class Employee(Person):
6)     eno=models.IntegerField()
7)     esal=models.FloatField()
8)
9)  class Manager(Employee):
10)    exp=models.IntegerField()
11)    team_size=models.IntegerField()
```

Note: Multilevel inheritance internally multitable inheritance only.

4) Multiple Inheritance:

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

```
1)  class Parent1(models.Model):
2)     f1=models.CharField(max_length=64)
3)     f2=models.CharField(max_length=64)
4)
5)  class Parent2(models.Model):
6)     f3=models.CharField(max_length=64)
7)     f4=models.CharField(max_length=64)
8)
9)  class Child(Parent1,Parent2):
10)    f5=models.CharField(max_length=64)
11)    f6=models.CharField(max_length=64)
```

Note:

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

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

```
from django.db import models
class CustomManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().order_by('eno')
# Create your models here.

class Employee(models.Model):
```

```
eno=models.IntegerField()
ename=models.CharField(max_length=64)
esal=models.FloatField()
eaddr=models.CharField(max_length=256)
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) <u>To Customize all() Method Behaviour, which Method we have to override inside Custom Manager Class?</u> get_queryset() Method

Q) <u>In Custom Manager Class, is it Possible to define New Methods?</u>
Yes

views.py

```
from django.shortcuts import render
from testapp.models import Employee

# Create your views here.
def display_view(request):
  #employees=Employee.objects.get_emp_sal_range(12000,20000)
employees=Employee.objects.get_employees_sorted_by('esal')
  my_dict={'employees':employees}
  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%}
```

5) Proxy Model Inheritance:

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

```
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()
20)     eaddr=models.CharField(max_length=256)
21)     objects=CustomManager1()
22)
23) class ProxyEmployee(Employee):
24)     objects=CustomManager2()
25)     class Meta:
26)         proxy=True
27)
28) class ProxyEmployee2(Employee):
29)     objects=CustomManager3()
30)     class Meta:
31)         proxy=True
```

admin.py

```
1)  from django.contrib import admin
2)  from testapp.models import Employee,ProxyEmployee,ProxyEmployee2
3)
4)  # Register your models here.
5)  class EmployeeAdmin(admin.ModelAdmin):
6)      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%}
```

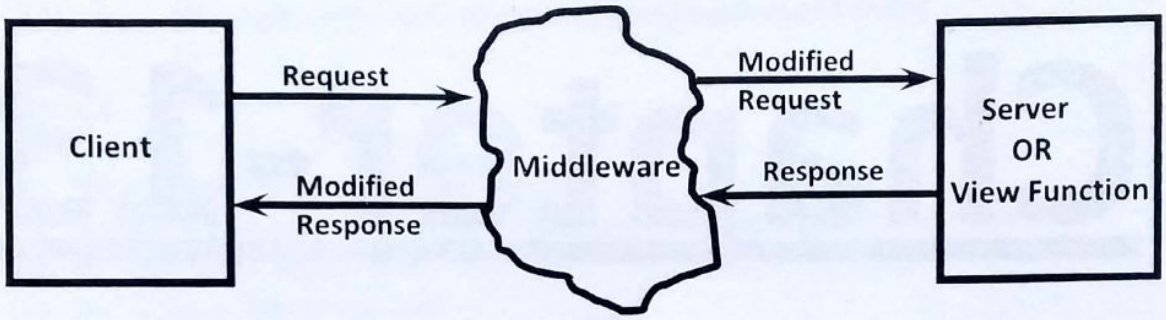
Chapter-13

Working with Django Middleware

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.



Client → Request → Middleware → modifiedrequest
Server → Response → Middleware → modifiedresponse

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

```
1) MIDDLEWARE = [
2)     'django.middleware.security.SecurityMiddleware',
3)     'django.contrib.sessions.middleware.SessionMiddleware',
4)     'django.middleware.common.CommonMiddleware',
5)     'django.middleware.csrf.CsrfViewMiddleware',
6)     'django.contrib.auth.middleware.AuthenticationMiddleware',
7)     'django.contrib.messages.middleware.MessageMiddleware',
8)     'django.middleware.clickjacking.XFrameOptionsMiddleware',
9) ]
```

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 after 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
   plz try after some time!!!</h1>')
```

settings.py

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

views.py

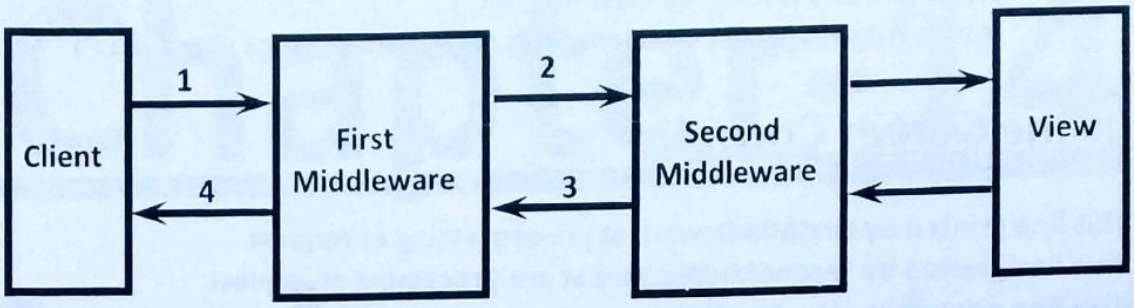
```
1) from django.http import HttpResponse
2)
3) # Create your views here.
4) def home_page_view(request):
5)     print(10/0)
6)     return HttpResponse('<h1>Hello This is from home page view</h1>')
```

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



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

Chapter-14

Deployment of Our Application in the Live Environment

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

Just login to github.com and create FREE account

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:

- Create a folder named with my_repo which acts as Git Repository.
- Copy the required files to this folder for tracking purposes.
- Initialize Git by using the following command → git init
- 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.
- 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'
- 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 Virtual Environment (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

Chapter-15

Real Time Project

Blog Application Development

How to specify dropdown choice field in models.py

```
1) class Post(models.Model):
2)    STATUS_CHOICES=(('draft','Draft'),
3)                    ('published','Published'))
4) status=models.CharField(max_length=10,choices=STATUS_CHOICES,default='draft')
```

slug field:

slug = models.SlugField(max_length=256,unique_for_date='publish')

- slug field can be used in urls.
- A slug is a short label containing only alphabet symbols,numbers,underscore symbol and hyphens.
- We can use slug field to build human understandable and SEO friendly URLs.
- unique_for_date specifies that slug is unique per publish date.

ForeignKey:

author = models.ForeignKey(User,related_name='blog-posts')

The value of author must be one of Users in default auth application.Multiple posts can be published by same author (same User). Hence it represents MANY-TO-ONE relationship. We can specify reverse relationship from User to Post with the related_name attribute.

DateTimeField:

publish = models.DateTimeField(default=timezone.now)

We have to use the following import ➔ from django.utils import timezone

models.py

```
1) from django.db import models
2) from django.contrib.auth.models import User
3) from django.utils import timezone
4) # Create your models here.
5) class Post(models.Model):
6)    STATUS_CHOICES=(('draft','Draft'),('published','Published'))
7)    title=models.CharField(max_length=264)
8)    slug=models.SlugField(max_length=264,unique_for_date='publish')
9)    author=models.ForeignKey(User,related_name='blog_posts')
10)   body=models.TextField()
11)   publish=models.DateTimeField(default=timezone.now)
12) created=models.DateTimeField(auto_now_add=True)#datetime of create() action
```

```
updated=models.DateTimeField(auto_now=True)#datetme of save() action
status=models.CharField(max_length=10,choices=STATUS_CHOICES,default='draft')

class Meta:
    ordering=('-publish',)
def __str__(self):
    return self.title
```

pytz Module (Python Timezone Module):

Timezone definitions are required for python and used by SQLite to work with datetimes. pytz module provides timezone definitions. Hence we required to install explicitly

pip install pytz

Note: Python 3.X alerady contains this module and we are not required to install.

Note: Django provides inbuilt support for timezones. In settings.py the following attribute already defined.

USE_TZ = True

Customization of Admin Interface:

We can customize admin interface for our model by defining ModelAdmin class.
In this class we can specify the following properties based on our requirement.

1) list_display:

Here we have to specify the field names which have to display.

2) list_filter

To filter records based on our provided fields
list_filter=('status','created','publish','author')
As the result, on the Right hand side side bar will be appeared for filtering purpose.

3) search_fields

search_fields=('title','body')
A search bar appeared on the page

4) prepopulated_fields

prepopulated_fields={'slug':('title',)}
If we type title then the same value will be considered automatically for the slug field also.

5) raw_id_fields

raw_id_fields=('author',)

author field is displayed with lookup widget,that can scale better than dropdown list. This is helpful if huge number of values are available.

6) date_hierarchy

date_hierarchy='publish'

Below search button,we can see a bar to navigate quickly based on published date.

7) ordering

ordering=['status','publish']

To display records according to specified order

admin.py

```
1)  from django.contrib import admin
2)  from blog.models import Post
3)
4)  # Register your models here.
5)  class PostAdmin(admin.ModelAdmin):
6)      list_display=['title','slug','author','publish','created','updated','status']
7)      list_filter=('status','created','publish','author')
8)      search_fields=('title','body')
9)      prepopulated_fields={'slug':('title',)}
10)     raw_id_fields=('author',)
11)     ordering=['status','publish']
12)
13) admin.site.register(Post,PostAdmin)
```

Implementation of List View and Detail View By using FBVs:

views.py

```
1)  from django.shortcuts import render,get_object_or_404
2)  from blog.models import Post
3)
4)  # Create your views here.
5)  def post_list_view(request):
6)      post_list=Post.objects.all()
7)      return render(request,'blog/post_list.html',{'post_list':post_list})
8)
9)  def post_detail_view(request,year,month,day,post):
```

```
10)   post=get_object_or_404(Post,slug=post,
11)                          status='published',
12)                          publish__year=year,
13)                          publish__month=month,
14)                          publish__day=day)
15)
16)   return render(request,'blog/post_detail.html',{'post':post})
```

urls.py

```
1)  from django.conf.urls import url
2)  from django.contrib import admin
3)  from blog import views
4)
5)  urlpatterns = [
6)      url(r'^admin/', admin.site.urls),
7)      url(r'^$', views.post_list_view),
8)      url(r'^(?P<year>\d{4})/(?P<month>\d{2})/(?P<day>\d{2})/(?P<post>[-
    \w]+)/$', views.post_detail_view,name='post_detail'),
9)  ]
```

models.py

```
1)  from django.db import models
2)  from django.contrib.auth.models import User
3)  from django.utils import timezone
4)  from django.core.urlresolvers import reverse
5)  # Create your models here.
6)  class CustomManager(models.Manager):
7)      def get_queryset(self):
8)          return super().get_queryset().filter(status='published')
9)
10) class Post(models.Model):
11)     STATUS_CHOICES=(('draft','Draft'),('published','Published'))
12)     title=models.CharField(max_length=264)
13)     slug=models.SlugField(max_length=264,unique_for_date='publish')
14)     author=models.ForeignKey(User,related_name='blog_posts')
15)     body=models.TextField()
16)     publish=models.DateTimeField(default=timezone.now)
17)     created=models.DateTimeField(auto_now_add=True)#datetime of create() action
18)     updated=models.DateTimeField(auto_now=True)#datetme of save() action
19) status=models.CharField(max_length=10,choices=STATUS_CHOICES,default='draft')
20)     objects=CustomManager()
21)
22)     class Meta:
```

```
23)         ordering=('-publish',)
24)     def __str__(self):
25)         return self.title
26)
27)     def get_absolute_url(self):
28)         return reverse('post_detail',args=[self.publish.year,
29)         self.publish.strftime('%m'),self.publish.strftime('%d'),self.slug])
```

Note: By using reverse() method we will get canonical URLs.

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"
crossorigin="anonymous">
7)   <link rel="stylesheet" href="{% static "css/blog.css"%}">
8)   <title>{%block title %}{%endblock%}</title>
9)  </head>
10) <body>
11)  <div class="content">
12)   {%block content%}{%endblock%}
13)  </div>
14)  <div class="sidebar">
15)   <h2>DURGA'S BLOG</h2>
16)   <p>Here You can see My Python Updations...</p>
17)  </div>
18) </body>
19) </html>
```

post_list.html

```
1) <!DOCTYPE html>
2) {%extends 'blog/base.html'%}
3) {%block title %}DURGA'S BLOG{%endblock%}
4) {%block content%}
5)  <h1>DURGA's Blog</h1>
6)  {%for post in post_list %}
7)   <h2> <a href="{{post.get_absolute_url}}">{{post.title}}</a> </h2>
8)   <p id='date'>Published on {{post.publish}} by {{post.author|title}}</p>
```

```
    {{post.body|truncatewords:30|linebreaks}}
{%endfor%}
{%endblock%}
```

post_detail.html

```
<!DOCTYPE html>
{%extends 'blog/base.html'%}
{%block title %}{{post.title}}{%endblock%}
{%block content%}
    <h1>{{post.title}}</h1>
    <p id='date'>Published on {{post.publish}} by {{post.author|title}}</p>
    {{post.body|linebreaks}}
{%endblock%}
```

blog.css

```
.content{
  float:left;
  width: 60%;
  padding:0 0 0 30px;
}
.sidebar{
  float:right;
  width:30%;
  padding:10px;
  background: green;
  height: 100%;
}
h1{
  text-align: center;
  border-bottom: 1px solid #bbb;
  padding: 0 0 10px 0;
}
#date{
  color:yellow;
  font-size:12px;
  font-style: italic;
}
body{
  background: red;
  color:white;
}
```

Django Pagination:

- If more number of posts (records) are available then it is highly recommended to display all records across multiple pages, so that end user can feel more comfortable. This concept is called Pagination (ie. Display records across multiple pages is called pagination)
- Django can provide inbuilt support for Pagination.It contains paginator module and Paginator class.

views.py

```
1)  from django.shortcuts import render,get_object_or_404
2)  from django.core.paginator import Paginator,EmptyPage,PageNotAnInteger
3)  from blog.models import Post
4)
5)  # Create your views here.
6)  def post_list_view(request):
7)      post_list=Post.objects.all()
8)      paginator=Paginator(post_list,2)
9)      page_number=request.GET.get('page')
10)     try:
11)         post_list=paginator.page(page_number)
12) except PageNotAnInteger:
13)         post_list=paginator.page(1)
14)     except EmptyPage:
15)         post_list=paginator.page(paginator.num_pages)
16)     return render(request,'blog/post_list.html',{'post_list':post_list})
```

How Pagination Works:

1) We have to create Paginator object with list of records and number of records per page.

```
paginator = Paginator(post_list,2)
```

2) We have to get list of records related to current page as follows.

```
page_number = request.GET.get('page')
post_list = paginator.page(page_number)
```

3) If the page_number is not an integer, means that we are not passing any page number in the query string with page attribute,it means it is first page,then we have to get first page records as follows

```
post_list=paginator.page(1)
```

4) If the parameter is higher than last page number then we will get EmptyPage.Instead of displaying EmptyPage we have to display last page records as follows

```
post_list = paginator.page(paginator.num_pages)
```

page_list.html

We have to add only one include tag to include paginator where ever pagination is required.

```
1) <!DOCTYPE html>
2) {%extends 'blog/base.html'%}
3) {%block title %}DURGA'S BLOG{%endblock%}
4) {%block content%}
5)  <h1>DURGA's Blog</h1>
6)  {%for post in post_list %}
7)    <h2> <a href="{{post.get_absolute_url}}">{{post.title}}</a> </h2>
8)    <p id='date'>Published on {{post.publish}} by {{post.author|title}}</p>
9)    {{post.body|truncatewords:30|linebreaks}}
10) {%endfor%}
11) {% include 'blog/pagination.html' with page=post_list%}
12) {%endblock%}
```

pagination.html

```
1)  <div class="paginator">
2)    <span>
3)      {%if page.has_previous%}
4)        <a href="?page={{page.previous_page_number}}">Previous</a>
5)      {%endif%}
6)    </span>
7)    <span class='current'>
8)      page {{page.number}} of {{page.paginator.num_pages}}
9)    </span>
10)   <span>
11)     {%if page.has_next%}
12)       <a href="?page={{page.next_page_number}}">Next</a>
13)     {%endif%}
14)  </span>
15) </div>
```

Note: We can reuse this pagination template anywhere pagination is required.

Class Based Views For Pagination:

We have to use paginate_by property to specify the number of records per page. ListView will always send 'page_obj' object which contains the current page records

```
1) from django.views.generic import ListView
2) class PostListView(ListView):
3)     model=Post
```

```
4)    paginate_by=1
```

post_list.html

```
1)  <!DOCTYPE html>
2)  {%extends 'blog/base.html'%}
3)  {%block title %}DURGA'S BLOG{%endblock%}
4)  {%block content%}
5)   <h1>DURGA's Blog</h1>
6)   {%for post in post_list %}
7)     <h2> <a href="{{post.get_absolute_url}}">{{post.title}}</a> </h2>
8)     <p id='date'>Published on {{post.publish}} by {{post.author|title}}</p>
9)     {{post.body|truncatewords:30|linebreaks}}
10) {%endfor%}
11) {% include 'blog/pagination.html' with page=page_obj%}
12) {%endblock%}
```

pagination.html

```
1)  same code
2)  <div class="paginator">
3)    <span>
4)      {%if page.has_previous%}
5)        <a href="?page={{page.previous_page_number}}">Previous</a>
6)      {%endif%}
7)    </span>
8)    <span class='current'>
9)      page {{page.number}} of {{page.paginator.num_pages}}
10)   </span>
11)   <span>
12)     {%if page.has_next%}
13)       <a href="?page={{page.next_page_number}}">Next</a>
14)     {%endif%}
15) </span>
16) </div>
```

Mail Sending Functionality:

In the web application sending mail is very common requirement. Django provides in built support for mail sending functionality.
Django provides mail module & send_mail() function.

To send email compulsory SMTP Server must be required.
SMTP → Simple Mail Transfer Protocol

We have to configure SMTP Server information in the settings.py file.
EMAIL_HOST: The SMTP Server Host. The default value is localhost
EMAIL_PORT: The SMTP Server Port. The default value is 25
EMAIL_HOST_USER: Username for SMTP Server
EMAIL_HOST_PASSWORD:Password for SMTP Server
EMAIL_USE_TLS: whether to use a TLS Secure Connection
(TLS: Transport Layer Security)

For gmail SMTP Server:

EMAIL_HOST= 'smtp.gmail.com'
EMAIL_PORT= 587
EMAIL_HOST_USER= Username for SMTP Server
EMAIL_HOST_PASSWORD=Password for SMTP Server
EMAIL_USE_TLS= True

We can send mail by using send_mail() function
send_mail(subject,msg,sender,list of receivers,fail_silently=False)

fail_silently is optional parameter. We are telling to raise exception if email not sending properly.

```
1) from django.core.mail import send_mail
2) send_mail('Hello','Very imp msg','python6656@gmail.com',[
   'xxx@gmail.com',yyy@gmail.com'])
```

If we are getting 1 then the mail sent successfully.

forms.py

```
1) from django import forms
2) class EmailSendForm(forms.Form):
3)     name=forms.CharField()
4)     email=forms.EmailField()
5)     to=forms.EmailField()
6)     comments=forms.CharField(required=False,widget=forms.Textarea)
```

views.py

```
1) from django.core.mail import send_mail
2) from blog.forms import EmailSendForm
3) ...
4) def mail_send_view(request,id):
5)     post=get_object_or_404(Post,id=id,status='published')
6)     sent=False
7)     if request.method=='POST':
```

```
8)        form=EmailSendForm(request.POST)
9)        if form.is_valid():
10)           cd=form.cleaned_data
11)           post_url=request.build_absolute_uri(post.get_absolute_url())
12)           subject='{}({}) recommends you to read "{}"'.format(cd['name'],cd['email'],
    post.title)
13)           message='Read Post At: \n {}\n\n{}\' Comments:\n{}'.format(post_url,cd
    ['name'],cd['comments'])
14)           send_mail(subject,message,'durga@blog.com',[cd['to']])
15)           sent=True
16)    else:
17)       form=EmailSendForm()
18) return render(request,'blog/sharebymail.html',{'post':post,'form':form,'sent':sent})
```

sharebymail.html

```
1)  <!DOCTYPE html>
2)  {%extends 'blog/base.html'%}
3)  {%block title %}Share Post By Email{%endblock%}
4)  {%block content%}
5)    {%if sent%}
6)     <h1>Email Successfully Sent !!!</h1>
7)      <p>"{{post.title}}" was sent by email</p>
8)    {%else%}
9)        <h1>Share {{post.title}}" by Email!!!"</h1>
10)      <form method="post">
11)       {{form.as_p}}
12)       {%csrf_token%}
13)       <input type="submit" name="" class='btn btn-lg btn-
    success' value="Send Mail">
14)      </form>
15)    {%endif %}
16) {%endblock%}
```

urls.py

```
1)  urlpatterns = [
2)      ...
3)      url(r'^(?P<id>\d+)/share/$', views.mail_send_view),
4)  ]
```

Add Send Mail Button in Details Page:

post_detail.html

```
1) <!DOCTYPE html>
2) {%extends 'blog/base.html'%}
3) {%block title %}{{post.title}}{%endblock%}
4) {%block content%}
5)     <h1>{{post.title}}</h1>
6)     <p id='date'>Published on {{post.publish}} by {{post.author|title}}</p>
7)     {{post.body|linebreaks}}
8)     <a href="/{{post.id}}/share" class='btn btn-lg btn-
   primary'>Share Post By Email</a>
9) {%endblock%}
```

Adding Comments Section to the Blog:

Activities:

1) We required to create a Model to save comments
2) We required to create a model based form to submit comments
3) We required to create a view function to process comments and save to the database
4) We have to edit post_detail.html to display list of comments and add a form to submit comments

models.py

Comments are always associated with Post and without Post there is no chance of existing comments.Each Post can have multiple comments.Hence it is Many-to-One relationship, which can be specified by Foreign Key.

```
class Comment(models.Model):
  post=models.ForeignKey(Post,related_name='comments')
```

We can use related_name to access all comments associated with a particular post.
post.comments.all()

If we are not using related_name then the default related name is:
modelclassname_set
Eg: comment_set

models.py

```
...
class Comment(models.Model):
    post=models.ForeignKey(Post,related_name='comments')
    name=models.CharField(max_length=32)
    email=models.EmailField()
    body=models.TextField()
    created=models.DateTimeField(auto_now_add=True)
    updated=models.DateTimeField(auto_now=True)
    active=models.BooleanField(default=True)
    class Meta:
        ordering=('created',)
    def __str__(self):
        return 'Commented By {} on {}'.format(self.name,self.post)

py manage.py  makemigrations
py manage.py  migrate
```

admin.py

```
from django.contrib import admin
from blog.models import Post,Comment

# Register your models here.
class PostAdmin(admin.ModelAdmin):
    list_display=['title','slug','author','publish','created','updated','status']
    list_filter=('status','created','publish','author')
    search_fields=('title','body')
    prepopulated_fields={'slug':('title',)}
    raw_id_fields=('author',)
    ordering=['status','publish']
class CommentAdmin(admin.ModelAdmin):
    list_display=('name','email','post','body','created','updated','active')
    list_filter=('active','created','updated')
    search_fields=('name','email','body')

admin.site.register(Post,PostAdmin)
admin.site.register(Comment,CommentAdmin)
```

Creation of Model Based form for comments

```
1) from blog.models import Comment
2) class CommentForm(forms.ModelForm):
3)    class Meta:
4)       model=Comment
5)       fields=('name','email','body')
```

Display and Processing Form in views.py

```
1) from blog.models import Comment
2) from blog.forms import CommentForm
3)
4) def post_detail_view(request,year,month,day,post):
5)    post=get_object_or_404(Post,slug=post,
6)                           status='published',
7)                           publish__year=year,
8)                           publish__month=month,
9)                           publish__day=day)
10)   comments=post.comments.filter(active=True)
11)   csubmit=False
12)   if request.method=='POST':
13)      form=CommentForm(data=request.POST)
14)      if form.is_valid():
15)         new_comment=form.save(commit=False)
16)         new_comment.post=post
17)         new_comment.save()
18)         csubmit=True
19)   else:
20)      form=CommentForm()
21)   return render(request,'blog/post_detail.html',{'post':post,'form':form,'comment
s':comments,'csubmit':csubmit})
```

Adding Comments to the Detail Page:

We have to display

1) Number of comments
2) For every comment we have to add head part
3) Comment body
4) Add Comment form to submit new comment
5) If already submitted then display success message

```
1) <!DOCTYPE html>
2) {%extends 'blog/base.html'%}
3) {%block title %}{{post.title}}{%endblock%}
4) {%block content%}
```

```
5)      <h1>{{post.title}}</h1>
6)      <p id='date'>Published on {{post.publish}} by {{post.author|title}}</p>
7)      {{post.body|linebreaks}}
8)      <a href="/{{post.id}}/share" class='btn btn-lg btn-primary'>
   Share Post By Email</a>
9)      {% with comments.count as comments_count %}
10)     <h2>{{comments_count}} Comment{{comments_count|pluralize}}</h2>
11)     {% endwith%}
12)     {%if comments %}
13)       {%for comment in comments %}
14)         <p id='ch'> comment {{forloop.counter}} by {{comment.name}} on
   {{comment.created}}</p>
15)         <div class="cb">{{comment.body|linebreaks}}</div>
16)         <hr>
17)       {%endfor%}
18)       {%else%}
19)       <p>There are NO Comments Yet !!!</p>
20)     {%endif%}
21)     {%if csubmit %}
22)       <h2>Your Comment Added Succefully</h2>
23)       {%else%}
24)       <form method="post">
25)         {{form.as_p}}
26)         {%csrf_token%}
27)         <input type="submit" name="" value="Submit Comment">
28)       </form>
29)     {%endif%}
30) {%endblock%}
```

Note:

```
{% with comments.count as comments_count %}
    <h2>{{comments_count}} Comment{{comments_count|pluralize}}</h2>
{% endwith%}
```

{%with%} can be used to assign a value to new variable. We can use that variable any number of times based on our requirement. Most of the times it can be used to avoid hitting database multiple times.

Note:

pluralize can be used to add 's' if it is required

Comment{{comments_count|pluralize}}

Eg: 0 Comments
1 Comment
2 Comments

Adding Tags Functionality To our Application:

Advantages of Tagging

1) For Search Engine Optimization
2) For easy navigation to the end user

By using 3rd party application 'taggit' we can implement tagging in our application.

How to install taggit Application

pip install django-taggit

After installation add this application in the settings.py

```
1) INSTALLED_APPS = [
2)     ....,
3)     'blog',
4)     'taggit'
5) ]
```

Add TaggableManager to our Post Model

```
1) from taggit.managers import TaggableManager
2) class Post(models.Model):
3)     ....
4)     objects=CustomManager()
5)     tags=TaggableManager()
6)     ...
```

To get all Posts:

Post.objects.all()

To get all tags associated with a post:

post.tags.all()

makemigrations and migrate:

to reflect changes made to Post model.

Operations related to taggit:

```
D:\djangoprojects\blogproject>py manage.py shell
>>> from blog.models import Post
>>> post=Post.objects.get(id=2)
>>> post.tags.add('python','django','datascience','dhoni')
>>> post.tags.all()
<QuerySet [<Tag: dhoni>, <Tag: django>, <Tag: python>, <Tag: datascience>]>
```

```
>>> post.tags.remove('datascience')
>>> post.tags.all()
<QuerySet [<Tag: dhoni>, <Tag: django>, <Tag: python>]>
```

Add Tags to the List Page:

To display tags associated with current post,we have to write code as follows.

```
{%for post in post_list %}
  ...
  <p>Tags:{{post.tags.all|join:','}}</p>
  ..
{%endfor%}
```

post_list.html

```
1)  <!DOCTYPE html>
2)  {%extends 'blog/base.html'%}
3)  {%block title %}DURGA'S BLOG{%endblock%}
4)  {%block content%}
5)   <h1>DURGA's Blog</h1>
6)   {%for post in post_list %}
7)     <h2> <a href="{{post.get_absolute_url}}">{{post.title}}</a> </h2>
8)     <p>Tags:{{post.tags.all|join:','}}</p>
9)     <p id='date'>Published on {{post.publish}} by {{post.author|title}}</p>
10)    {{post.body|truncatewords:30|linebreaks}}
11) {%endfor%}
12) {% include 'blog/pagination.html' with page=post_list%}
13) {%endblock%}
```

Editing post_list_view() for Tags

If we click any tag name then to display all posts related to that tag we have to edit view function.

```
1)  from taggit.models import Tag
2)  def post_list_view(request,tag_slug=None):
3)      post_list=Post.objects.all()
4)      tag=None
5)      if tag_slug:
6)          tag=get_object_or_404(Tag,slug=tag_slug)
7)          post_list=post_list.filter(tags__in=[tag])
8)
9)      paginator=Paginator(post_list,2)
10)     page_number=request.GET.get('page')
11)     try:
12)         post_list=paginator.page(page_number)
```

```
13)    except PageNotAnInteger:
14)        post_list=paginator.page(1)
15)    except EmptyPage:
16)        post_list=paginator.page(paginator.num_pages)
17)    return render(request,'blog/post_list.html',{'post_list':post_list,'tag':tag})
```

urls.py

```
1)  from django.conf.urls import url
2)  from django.contrib import admin
3)  from blog import views
4)
5)  urlpatterns = [
6)      url(r'^admin/', admin.site.urls),
7)      url(r'^mail/', views.mail_send_view),
8)      url(r'^$', views.post_list_view),
9)      url(r'^tag/(?P<tag_slug>[-
    \w]+)/$', views.post_list_view, name='post_list_by_tag_name'),
10)     # url(r'^$', views.PostListView.as_view()),
11)     url(r'^(?P<year>\d{4})/(?P<month>\d{2})/(?P<day>\d{2})/(?P<post>[-
    \w]+)/$', views.post_detail_view,name='post_detail'),
12)     url(r'^(?P<id>\d+)/share/$', views.mail_send_view),
13) ]
```

post_list.html

```
1)  <!DOCTYPE html>
2)  {%extends 'blog/base.html'%}
3)  {%block title %}DURGA'S BLOG{%endblock%}
4)  {%block content%}
5)   <h1>DURGA's Blog</h1>
6)   {%if tag%}
7)   <h2>Posts tagged with "{{tag.name}}"</h2>
8)   {%endif%}
9)   {%for post in post_list %}
10)   <h2> <a href="{{post.get_absolute_url}}">{{post.title}}</a> </h2>
11)   <p>Tags:
12)    {%for tag in post.tags.all %}
13)    <a href="{%url "post_list_by_tag_name" tag.slug %}">{{tag.name}}</a>
14)    {%if not forloop.last%}, {%endif%}
15)    {%endfor%}
16)   </p>
17)   <p id='date'>Published on {{post.publish}} by {{post.author|title}}</p>
18)   {{post.body|truncatewords:30|linebreaks}}
19) {%endfor%}
```

```
20) {% include 'blog/pagination.html' with page=post_list%}
21) {%endblock%}
```

How to display similar posts:

1) Retrieve all tags for the current post
2) Get all posts that are tagged with any of those tags
3) Exclude current post from the list to avoid recommended the same post
4) Order the results by number of tags shared with the current post
5) If two or more posts with the same number of tags, recommend the most recent post
6) Limit the query to the number of posts we have to recommend

```
1) from django.db.models import Count
2)
3) post_tags_ids=post.tags.values_list('id',flat=True)
4) similar_posts=Post.objects.filter(tags__in=post_tags_ids).exclude(id=post.id)
5) similar_posts=similar_posts.annotate(same_tags=Count('tags')).order_by('-
   same_tags','publish')[:4]
```

Note: By using annotate() function, we can add fields to the query. In the above code, Count() function returned value we are assigning to same_tags variable and this variable can be used in next level.

views.py

```
1) ....
2) from blog.models import Comment
3) from blog.forms import CommentForm
4) from django.db.models import Count
5)
6) def post_detail_view(request,year,month,day,post):
7)     post=get_object_or_404(Post,slug=post,
8)                            status='published',
9)                            publish__year=year,
10)                           publish__month=month,
11)                           publish__day=day)
12)    post_tags_ids=post.tags.values_list('id',flat=True)
13)    similar_posts=Post.objects.filter(tags__in=post_tags_ids).exclude(id=post.id)
14)    similar_posts=similar_posts.annotate(same_tags=Count('tags')).order_by('-
   same_tags','publish')[:4]
15)
16)    comments=post.comments.filter(active=True)
17)    csubmit=False
18)    if request.method=='POST':
19)        form=CommentForm(data=request.POST)
20)        if form.is_valid():
```

```
        new_comment=form.save(commit=False)
        new_comment.post=post
        new_comment.save()
        csubmit=True
    else:
      form=CommentForm()
    return render(request,'blog/post_detail.html',{'post':post,'form':form,'comment
s':comments,'csubmit':csubmit,'similar_posts':similar_posts})
```

post_detail.html

In this html page we have to display similar posts

```
<h2>Similar Posts</h2>
{%if similar_posts%}
   {%for post in similar_posts%}
     <h3> <a href="{{post.get_absolute_url}}">{{post.title}}</a> </h3>
   {%endfor%}
    {%else%}
        <p>No Similar Posts yet</p>
{%endif%}
```

complete file

```
<!DOCTYPE html>
{%extends 'blog/base.html'%}
{%block title %}{{post.title}}{%endblock%}
{%block content%}
     <h1>{{post.title}}</h1>
     <p id='date'>Published on {{post.publish}} by {{post.author|title}}</p>
     {{post.body|linebreaks}}
     <a href="/{{post.id}}/share" class='btn btn-lg btn-primary'>
Share Post By Email</a>
     <h2>Similar Posts</h2>
     {%if similar_posts%}
        {%for post in similar_posts%}
         <h3> <a href="{{post.get_absolute_url}}">{{post.title}}</a> </h3>
        {%endfor%}
        {%else%}
        <p>No Similar Posts yet</p>
     {%endif%}
     {% with comments.count as comments_count %}
     <h2>{{comments_count}} Comment{{comments_count|pluralize}}</h2>
     {% endwith%}
     {%if comments %}
        {%for comment in comments %}
```

```
22)         <div class="commentdata">
23)           <p id='ch'> comment {{forloop.counter}} by {{comment.name}} on
   {{comment.created}}</p>
24)           {{comment.body|linebreaks}}
25)         </div>
26)       {%endfor%}
27)       {%else%}
28)       <p>There are NO Comments Yet !!!</p>
29)     {%endif%}
30)     {%if csubmit %}
31)       <h2>Your Comment Added Succefully</h2>
32)       {%else%}
33)       <form method="post">
34)         {{form.as_p}}
35)         {%csrf_token%}
36)         <input type="submit" name="" value="Submit Comment">
37)       </form>
38)     {%endif%}
39) {%endblock%}
```

Custom Template Tags:

Django provide a big list of template tags like

{{post.title}}
{% csrf_token %}
{% if post_list %} {%endif %}
{%for post in post_list %} ..{%endfor %}
etc

Sometimes these predefined tags may not fulfill our requirement. In this case we can define our own template tag based on our required functionality,such type of tags are called Custom Template Tags.

3 Utility Functions to define Custom Template Tags:

Django provides the following 3 helper functions for creating our own template tags.

1) **simple_tag:** Perform some processing and returns a string
2) **inclusion_tag:** Perform some processing and returns a rendered template
3) **assignment_tag:** Perform some processing and assign the result to a variable in the context.

How to Create Custom Template Tag:

1) Inside our application folder ceate a folder named with 'templatetags'
2) In that 'templatetags' folder,create an empty file __init__.py,so that this folder is considered as python package
3) In that 'templatetags' folder, create a python file(blog_tags.py) to hold custom template tags.

```
...
 blog
  |--.....
  |-templatetags
    |- __init__.py
    |- blog_tags.py
```

Note: custom template tags should be placed in application folder but not in project folder

Usage of simple_tag to return number of posts published:

blog_tags.py

```
1) from django import template
2) from blog.models import Post
3) register=template.Library()
4)
5) @register.simple_tag
6) def total_posts():
7)     return Post.objects.count()
```

Note:

1) By using register variable we can register template tags with Library.
2) By default function name (total_posts) will become tag name.But we can specify our own name as follows

@register.simple_tag(name='my_tag')

How to use Custom Template Tag

1) We have to load custom tag functionality as follows → {% load blog_tags %}
2) We can call template tag as follows → {% total_posts %}

base.html

```
1) <!DOCTYPE html>
2) {%load blog_tags %}
3) {%load staticfiles%}
4) <html lang="en" dir="ltr">
```

```
5)  <head>
6)    <meta charset="utf-8">
7)    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7
/css/bootstrap.min.css" integrity="sha384-
BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" c
rossorigin="anonymous">
8)    <link rel="stylesheet" href="{% static "css/blog.css"%}">
9)    <title>{%block title %}{%endblock%}</title>
10) </head>
11) <body>
12)   <div class="content">
13)     {%block content%}{%endblock%}
14)   </div>
15)   <div class="sidebar">
16)     <h2>DURGA'S BLOG</h2>
17)     <p>The total number of published posts upto today: {%total_posts%}</p>
18)   </div>
19) </body>
20) </html>
```

Usage of inclusion_tag to display Latest Posts

The most commonly used and powerful utility function is inclusion_tag.

blog_tags.py

```
1) ...
2) @register.inclusion_tag('blog/latest_posts123.html')
3) def show_latest_posts(count=5):
4)     latest_posts=Post.objects.order_by('-publish')[:count]
5)     return {'latest_posts':latest_posts}
```

latest_posts123.html

```
1) <ul>
2)   {% for post in latest_posts  %}
3)   <li><a href="{{post.get_absolute_url}}">{{post.title}}</a></li>
4)   {% endfor %}
5) </ul>
```

base.html

```
1) ...
2) <div class="sidebar">
3)     <h2>DURGA'S BLOG</h2>
4)     <p>The total number of published posts upto today:{%total_posts %}</p>
```

```
5)      <h3>Latest Posts:</h3>{% show_latest_posts 100 %}
6)  </div>
```

Usage of assignment_tag to display Most Commented Posts:

Assignment_tag is exactly same as simple_tag except that it stores the result in the given variable.

blog_tags.py

```
1)  from django.db.models import Count
2)
3)  @register.assignment_tag
4)  def get_most_commented_posts(count=5):
5)     return Post.objects.annotate(total_comments=Count('comments')).order_by('-
    total_comments')[:count]
```

base.html

```
1)  <div class="sidebar">
2)      <h2>DURGA'S BLOG</h2>
3)      <p>The total number of published posts upto today: <span id='pcount'>{%total
    _posts %}</span></p>
4)      <h3>Latest Posts:</h3>{% show_latest_posts 100 %}
5)      <h3>Most Commented Posts:</h3>
6)      {% get_most_commented_posts as most_commented_posts%}
7)      <ul>
8)       {% for post in most_commented_posts  %}
9)       <li><a href="{{post.get_absolute_url}}" id='la'>{{post.title}}</a></li>
10)      {% endfor %}
11)     </ul>
12) </div>
```

Chapter-16

Introduction To Web Application Development by using Flask

How to develop web application by using Flask Framework:

How to install Flask:

pip install flask

How to Develop Application:

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)