

# 01-Framework

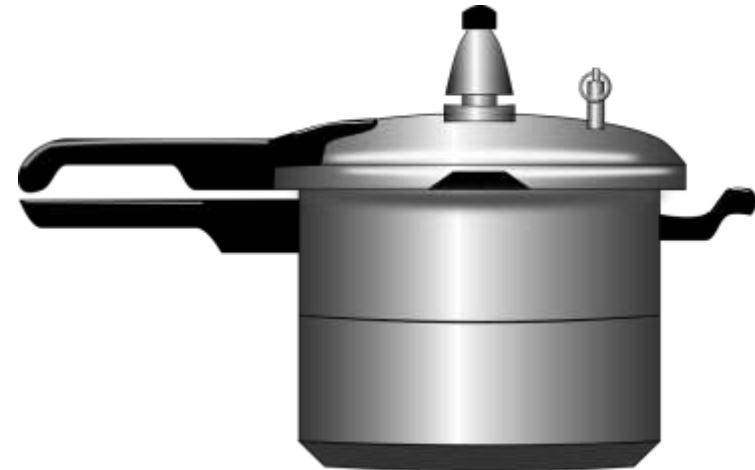
A Framework is a set of conceptual structure and guidelines, used to build something useful.

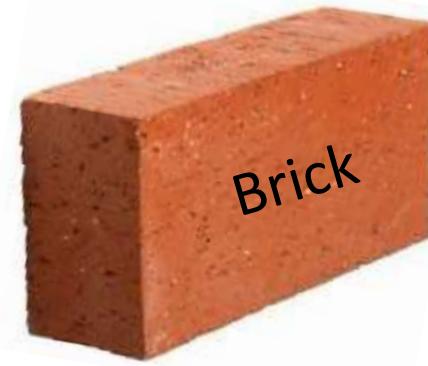


**khichdi**

- Chawal
- Daal
- Aalu
- Gobi
- Namak
- Haldi
- Mirchi

2 Katori Chawal  
1 Katori Daal  
2 Aalu  
4 Gobi  
1 Chamach Namak  
 $\frac{1}{2}$  Chamach Haldi  
1 Mirchi





Brick

# **Framework**

A framework may include predefined classes and functions that can be used to process input, manage hardware devices, and interact with system software.

The purpose of the framework is to allow developers to focus on building a unique feature for their Projects rather than writing code from scratch.

# Why use Framework

- Collection of tools
- No need to start from scratch
- Save Time
- Improve Productivity
- Clean Code
- Reusable Code
- Testing
- Debugging

## **02-Web Framework**

A Web Framework (WF) or Web Application Framework (WAF) which helps to build Web Applications.

Web frameworks provide tools and libraries to simplify common web development operations. This can include web services, APIs, and other resources.

Web frameworks help with a variety of tasks, from templating and database access to session management and code reuse.

More than 80% of all web app frameworks rely on the Model View Controller architecture.

# Some Web Framework

- Laravel
- Codeigniter
- Zend
- Django
- Spring

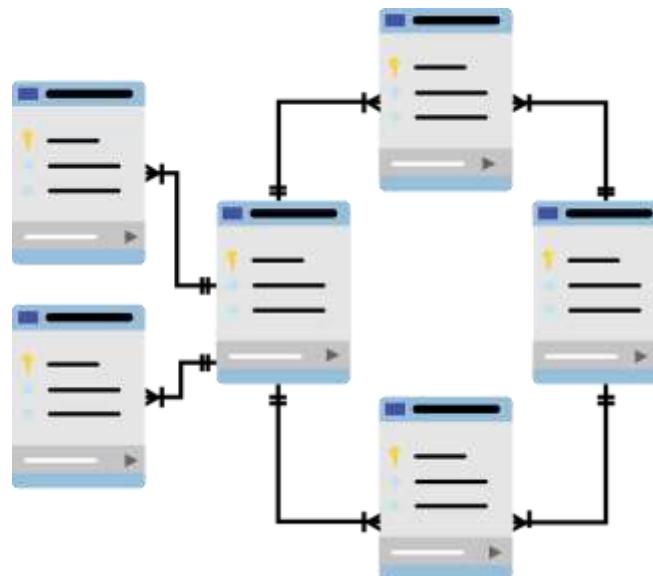
# 03-Model View Template (MVT)

The MVT is an design pattern that separates an application into three main logical components *Model*, *View*, and *Template*.

Each of these component has their own role in a Project.

# Model

The Model responsible to handle database. It is a data access layer which handles the data.



# View

The user can send request by interacting with template, the view handles these requests and sends to Model then get appropriate response from the Model, sends response to template.

It may also have required logics.

It works as a mediator between Template and Model.

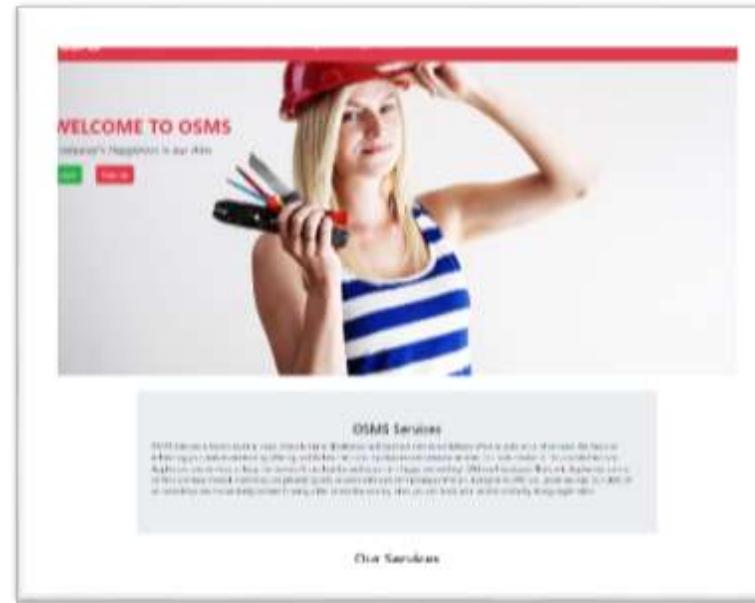


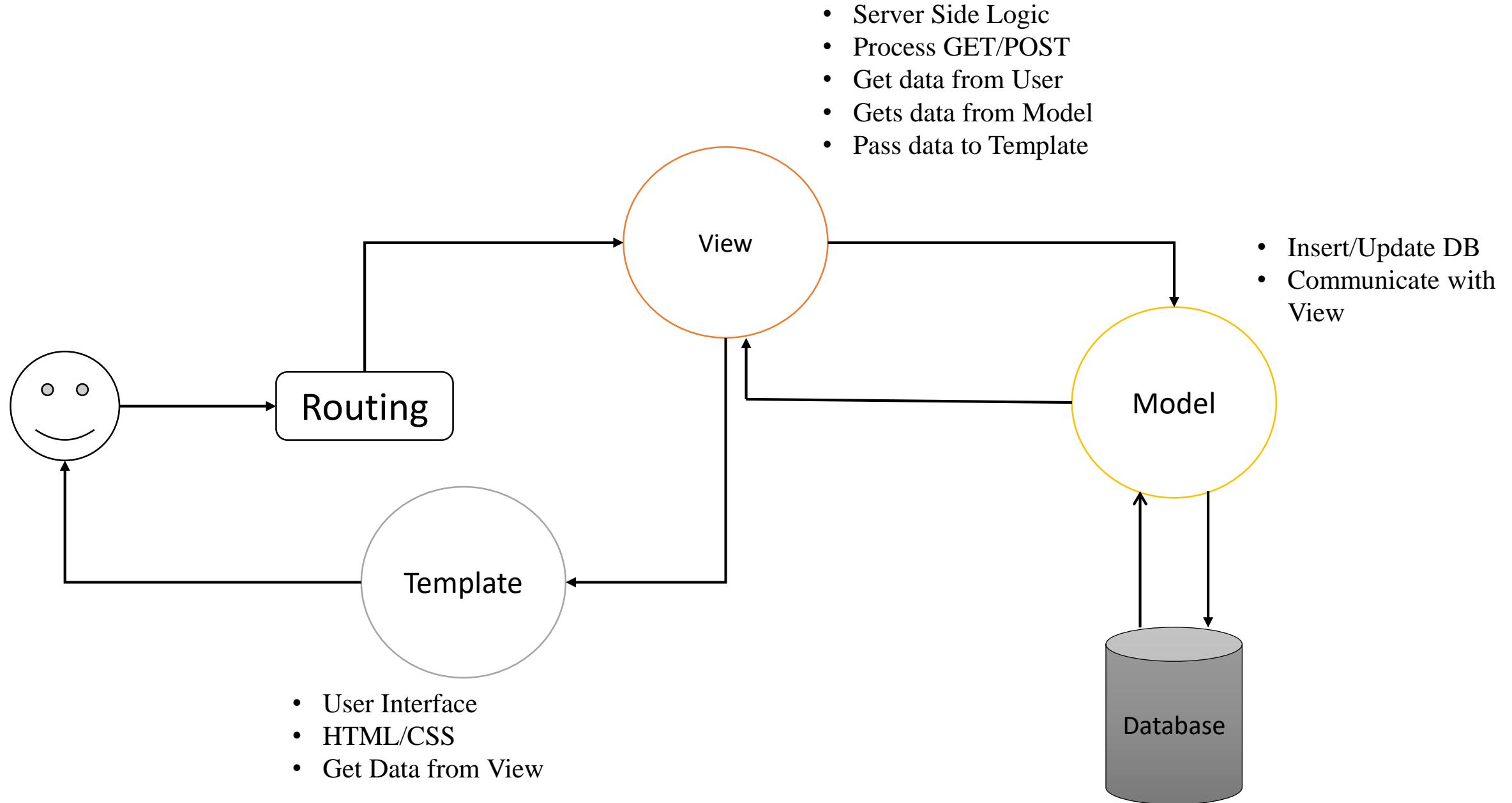
# Template

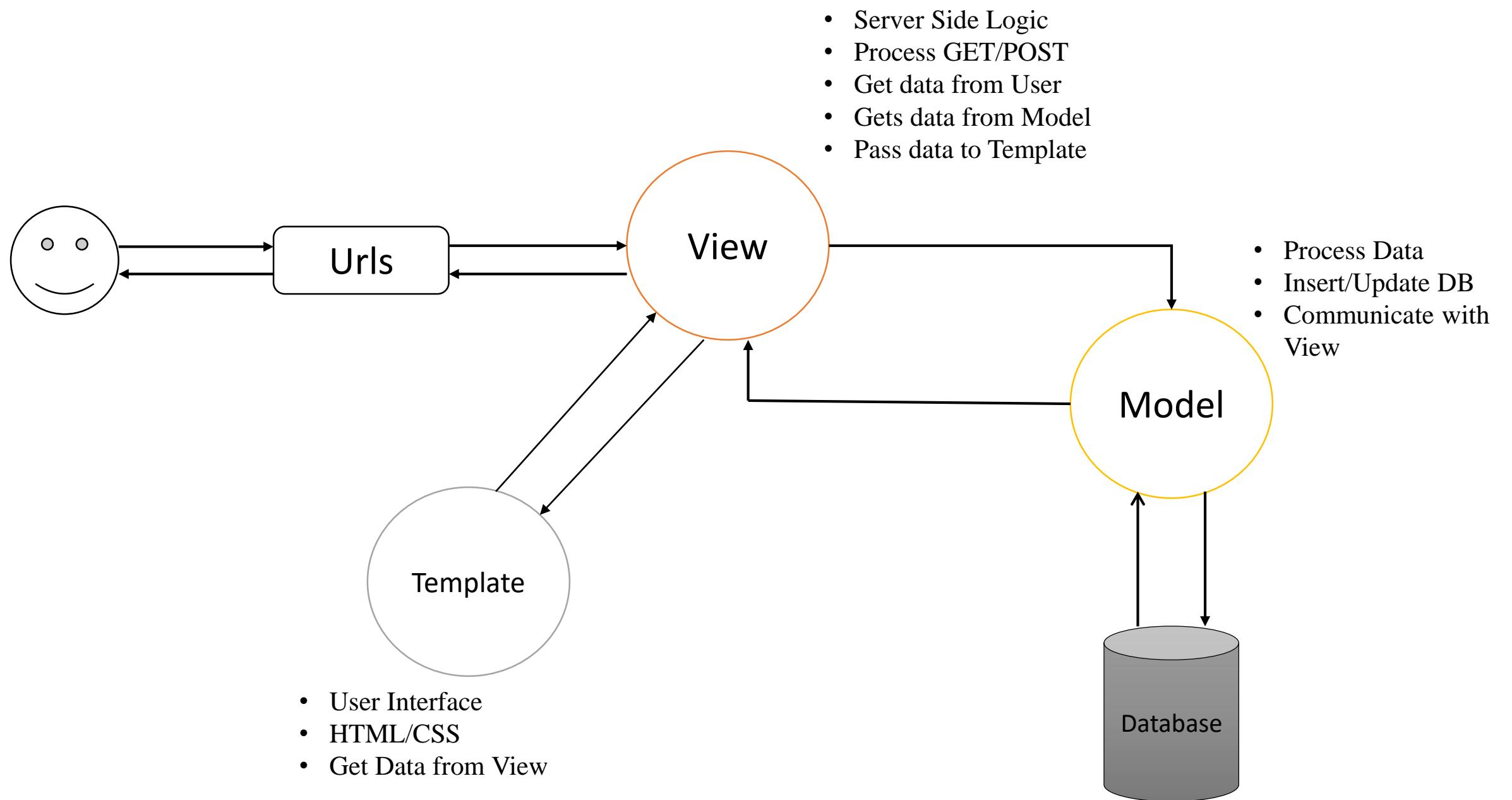
It represents how data should be presented to the application user. User can read or write the data from template.

Basically it is responsible for showing end user content, we can say it is user interface.

It may consists of HTML, CSS, JS mixed with Django Template Language.







# Why use MVT

- Organized Code
- Independent Block
- Reduces the complexity of web applications
- Easy to Maintain
- Easy to modify

# Basic Structure



# 04-Django

Django is a free, open-source Python based **High-Level Web Framework**.

It follows the **Model View Template** (MVT) architectural pattern.

It was originally created by **Adrian Holovaty** and **Simon Willison**.

It was created on 2003 at **Lawrence Journal World Newspaper**.

It was released publicly under a **BSD** license in July 2005.

In June 2008, it was announced that **Django Software Foundation** (DSF) would maintain Django in the future.

# What we can build with Django

We can build High End Web Applications.

It encourages rapid development and clean, pragmatic design.

- Youtube
- Instagram
- Bitbucket
- NASA
- Spotify

# Advantages of Django

- Open Source
- Fast
- Secure
- Scalable
- Authentication
- Versatile
- Provides Development Web Server by Default.
- Provides SQLite Database by Default.

# 05-Do You Know ?

- HTML
- CSS
- JavaScript
- SQL
- Python
- MVT
- PIP
- Bootstrap

# Django Requirements

- Python 3.0 or Higher
- PIP
- Text/Code Editor/IDE – Notepad++, VS Code, ATOM, Brackets, PyCharm
- Web Browser – Google Chrome, Mozilla Firefox, Edge

# 06-Django Requirements

- Python 3.0 or Higher  
`python --version`
- PIP  
`pip --version`
- Text/Code Editor/IDE – Notepad++, VS Code, ATOM, Brackets, PyCharm
- Web Browser – Google Chrome, Mozilla Firefox, Edge

# Check Django is installed or Not

*django-admin --version*

# How to install Django

- Separate Virtual Environment
- Globally

# Install Django in Separate Environment

## Install Virtual Environment Wrapper

*pip install virtualenvwrapper-win* – This is used to install Virtual environment wrapper.

## Create Virtual Environment (VE)

*mkvirtualenv envname* – This is used to create virtual environment. It will automatically activate environment.

## Active VE

*workon envname* – This is used to activate environment.

## Install Django in Created VE

*pip install django* – First activate environment then run the command to install django within active environment. You can also specify version *pip install django==2.0*

# Django Project

A Django Project may contain multiple Project Application, which means a group of Application and files is called as Django Project.

An Application is a Part of Django Project.

## SchoolProject

- Registration App
- Fees App
- Exam App
- Attendance App
- Result App

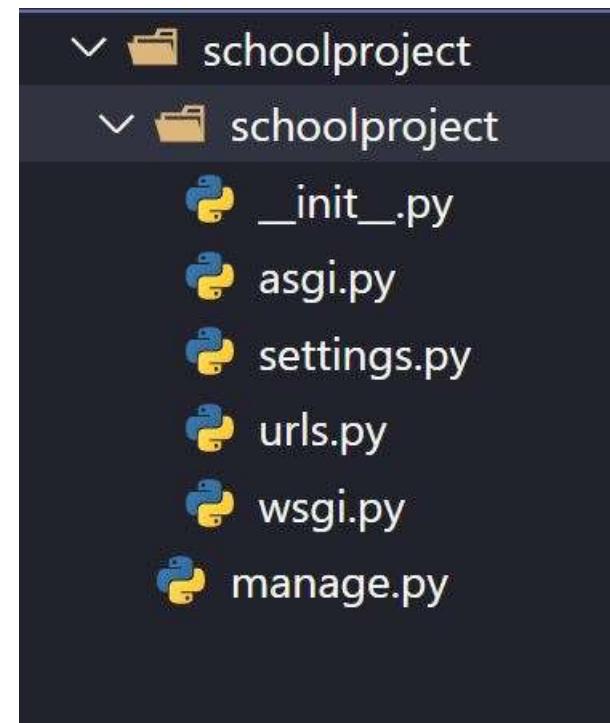
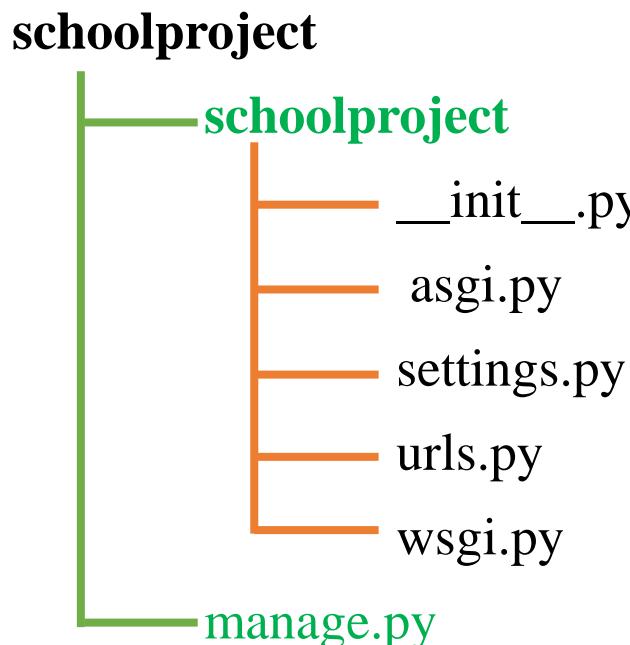
# Create Django Project

Syntax: -

*django-admin startproject **projectname***

Example:-

*django-admin startproject **schoolproject***



# Uninstall Django from Separate Environment

## Active Virtual Environment (VE)

*workon envname* – This is used to activate environment.

## Uninstall Django from VE

*pip uninstall django* – This is used to uninstall django.

## Remove Virtual Environment

*rmvirtualenv envname* – This is used to remove virtual environment.

## Uninstall Virtual Environment Wrapper

*pip uninstall virtualenvwrapper-win* – This is used to uninstall Virtual environment wrapper.

# How to install Django

- Separate Virtual Environment
- Globally

# Django Requirements

- Python 3.0 or Higher  
`python --version`
- PIP  
`pip --version`
- Text/Code Editor/IDE – Notepad++, VS Code, ATOM, Brackets, PyCharm
- Web Browser – Google Chrome, Mozilla Firefox, Edge

# Check Django is installed or Not

*django-admin --version*

# How to Install Django

*pip install django* – This is used to install Django.

## For Version Specific

*pip install django==2.0*

# How to Uninstall Django

*pip uninstall django*

# Django Project

A Django Project may contain multiple Project Application, which means a group of Application and files is called as Django Project.

An Application is a Part of Django Project.

## SchoolProject

- Registration App
- Fees App
- Exam App
- Attendance App
- Result App

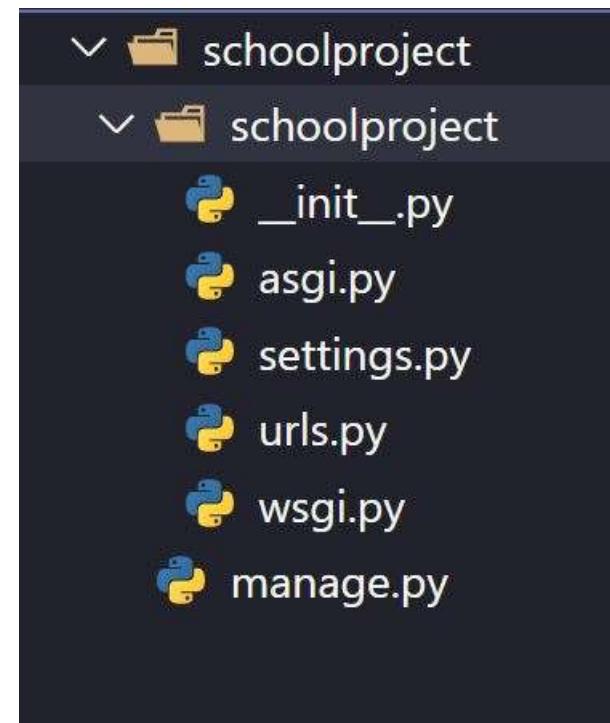
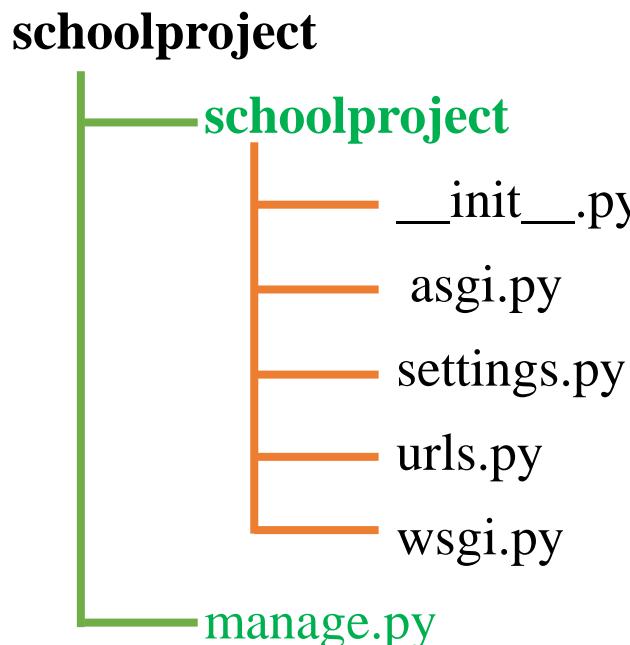
# Create Django Project

Syntax: -

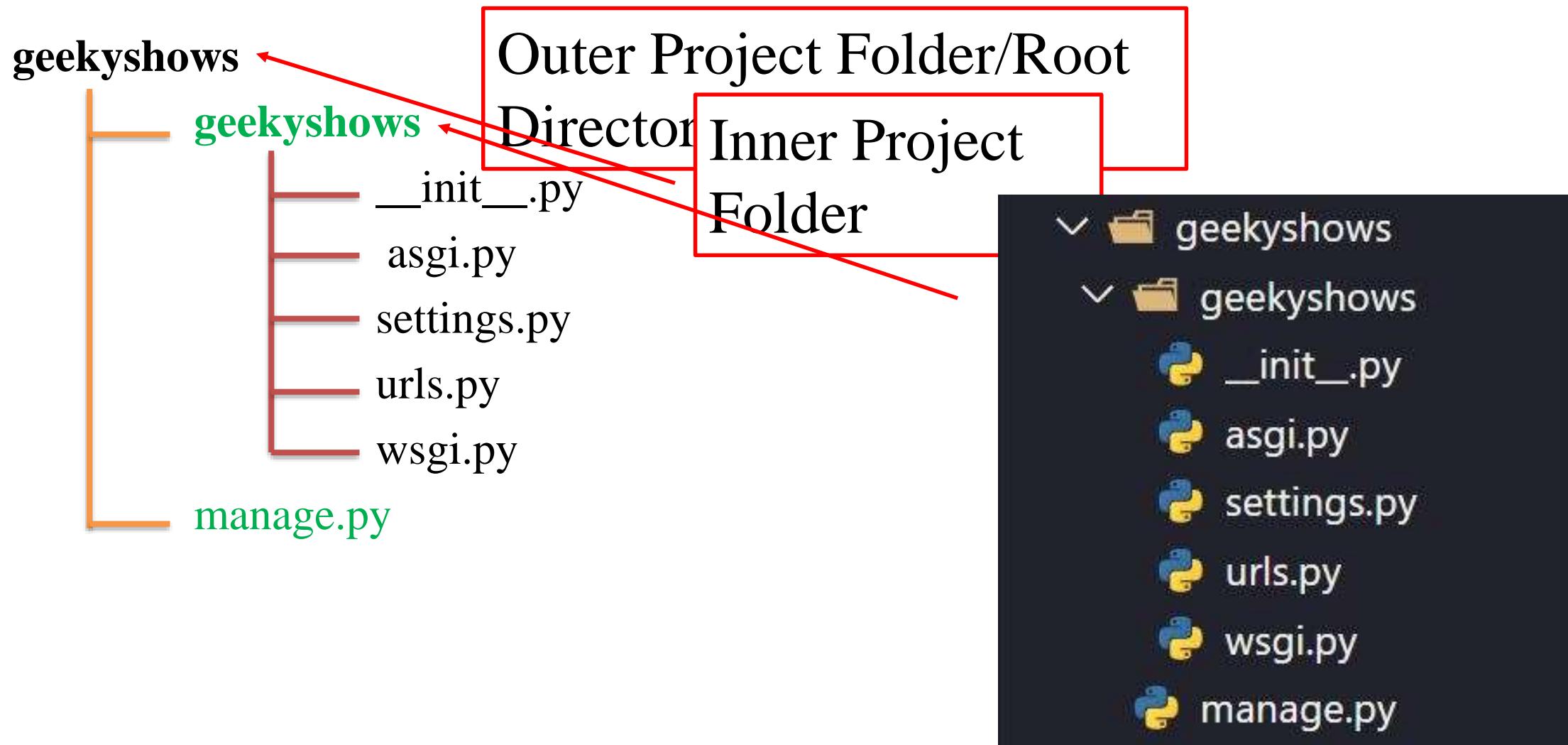
*django-admin startproject **projectname***

Example:-

*django-admin startproject **schoolproject***



# 07-Django Project Directory Structure



# Django Project Directory Structure

**\_\_init\_\_.py** – The folder which contains `__init__.py` file is considered as python package.

**wsgi.py** – WSGI (Web Server Gateway Interface) is a specification that describes how a web server communicates with web applications, and how web applications can be chained together to process one request. WSGI provided a standard for synchronous Python apps.

**asgi.py** – ASGI (Asynchronous Server Gateway Interface) is a spiritual successor to WSGI, intended to provide a standard interface between async-capable Python web servers, frameworks, and applications. ASGI provides standard for both asynchronous and synchronous apps.

**settings.py** – This file contains all the information or data about project settings.

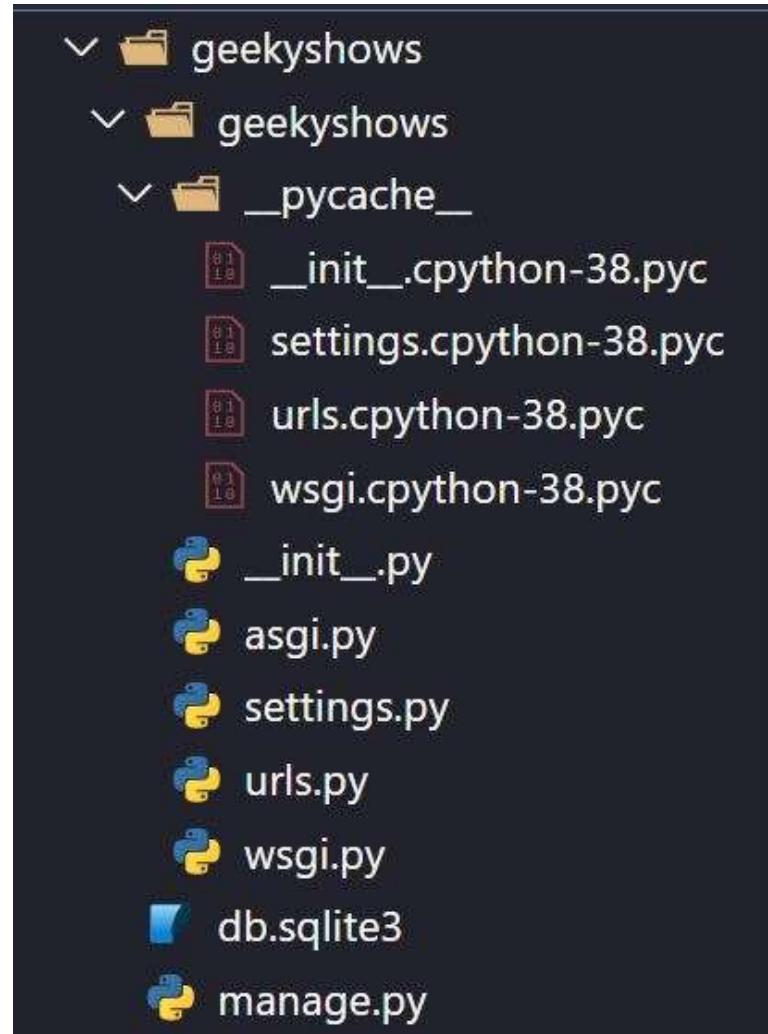
E.g.: Database Config information, Template, Installed Application, Validators etc.

**urls.py** – This file contains information of url attached with application.



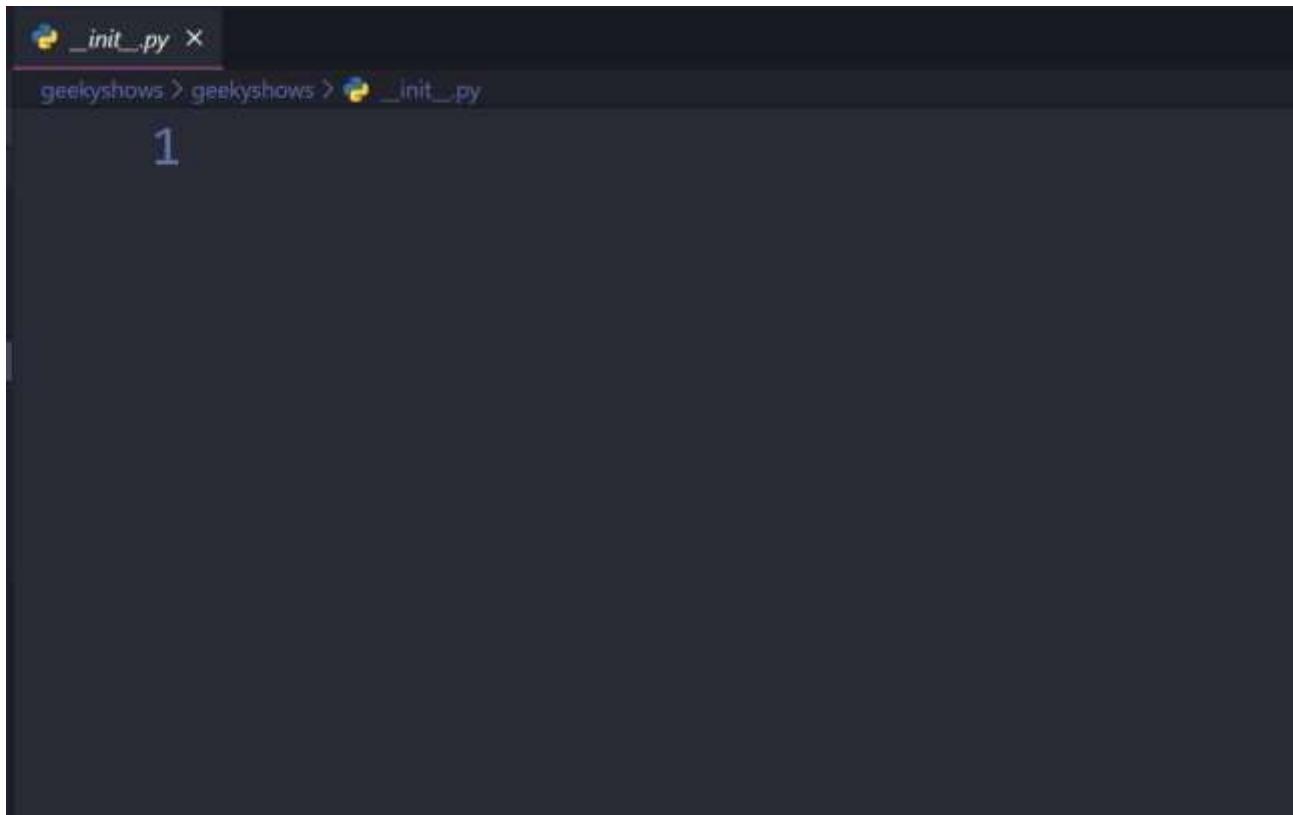
**manage.py** – `manage.py` is automatically created in each Django project. It is Django's command-line utility also sets the `DJANGO_SETTINGS_MODULE` environment variable so that it points to your project's `settings.py` file. Generally, when working on a single Django project, it's easier to use `manage.py` than `django-admin`.

# Django Project Directory Structure

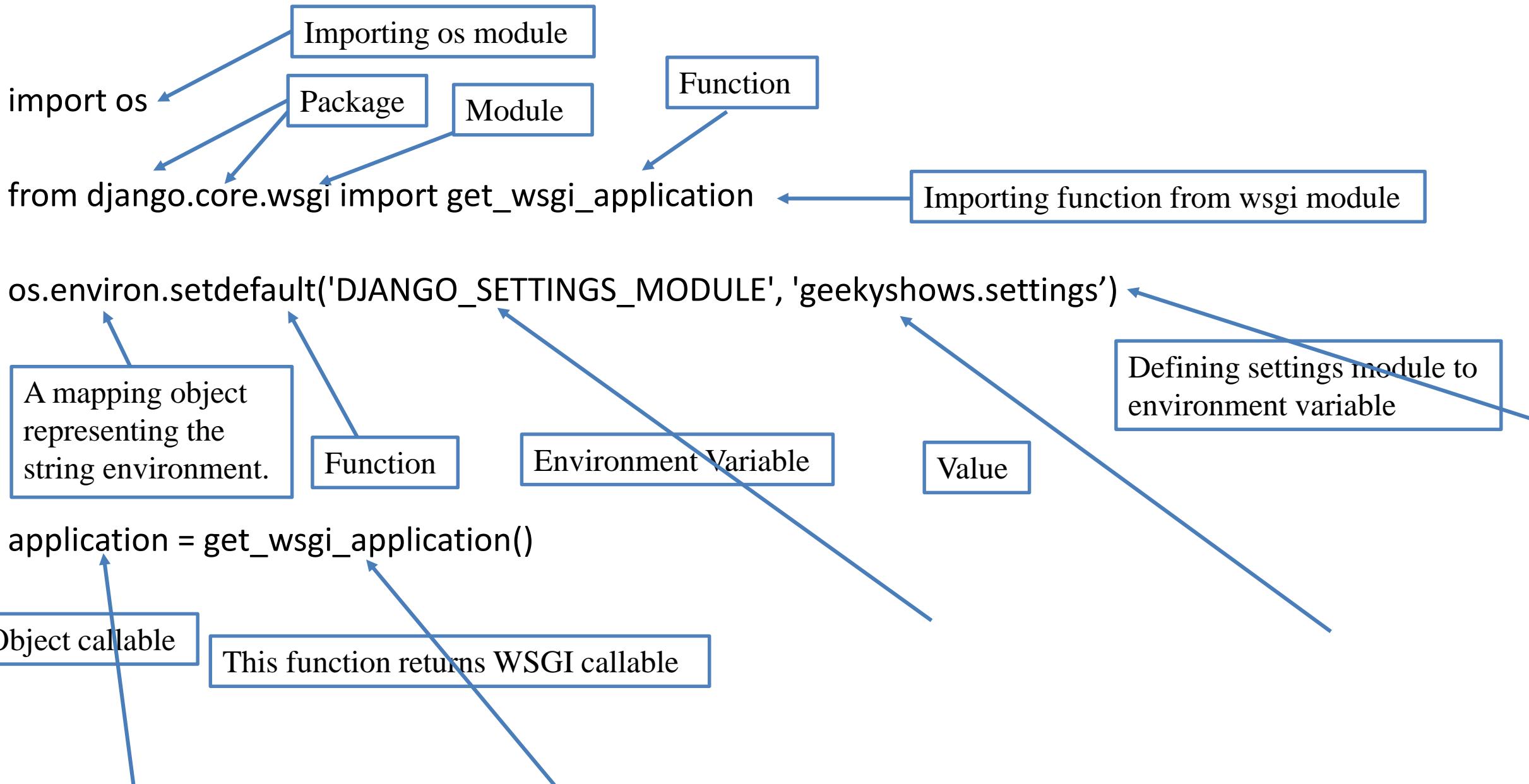


# `__init__.py`

The folder which contains `__init__.py` file is considered as python package.



# wsgi.py



# wsgi.py

```
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'geekyshows.settings')
```

When the WSGI server loads your application, Django needs to import the settings module, that's where your entire application is defined.

Django uses the DJANGO\_SETTINGS\_MODULE environment variable to locate the appropriate settings module.

It must contain the dotted path to the settings module.

You can use a different value for development and production; it all depends on how you organize your settings. If this variable isn't set, the default wsgi.py sets it to mysite.settings, where mysite is the name of your project. That's how runserver discovers the default settings file by default.

# asgi.py

```
import os

from django.core.asgi import get_asgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'geekyshows.settings')

application = get_asgi_application()
```

# settings.py

*import os*

Importing os module

*BASE\_DIR = os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))*

BASE\_DIR is a variable which contains abspath of base directory/project folder

e.g. - C:\AllDjango\geekyshows

# settings.py

```
SECRET_KEY = '9f=3m6^04@*z1dn*!utxe^yn!3vpkjtbbg0&t^+_cxaigm*7p'
```

This is used to provide cryptographic signing, and should be set to a unique, unpredictable value. django-admin startproject automatically adds a randomly-generated SECRET\_KEY to each new project.

Django will refuse to start if SECRET\_KEY is not set.

## The secret key is used for:

- All sessions if you are using any other session backend than django.contrib.sessions.backends.cache, or are using the default get\_session\_auth\_hash().
- All messages if you are using CookieStorage or FallbackStorage.
- All PasswordResetView tokens.
- Any usage of cryptographic signing, unless a different key is provided.

# settings.py

## *DEBUG = True*

A Boolean (True/False) that turns on/off debug mode.

Never deploy a site into production with DEBUG turned on.

One of the main features of debug mode is the display of detailed error pages.

If DEBUG is *False*, you also need to properly set the ALLOWED\_HOSTS setting.

Failing to do so will result in all requests being returned as “Bad Request (400)”.

# settings.py

## *ALLOWED\_HOSTS = [ ]*

A list of strings representing the host/domain names that this Django site can serve. Values in this list can be fully qualified names (e.g. 'www.example.com'), in which case they will be matched against the request's Host header exactly (case-insensitive, not including port).

A value beginning with a period can be used as a subdomain wildcard: '.example.com' will match example.com, www.example.com, and any other subdomain of example.com.

A value of '\*' will match anything; in this case you are responsible to provide your own validation of the Host header.

# settings.py

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

Built-in Applications

A list of strings designating all applications that are enabled in this Django installation. Each string should be a dotted Python path to an application configuration class (preferred) or a package containing an application.

Application names and labels must be unique in INSTALLED\_APPS.

Application names – The dotted Python path to the application package must be unique. There is no way to include the same application twice, short of duplicating its code under another name.

Application labels – By default the final part of the name must be unique too. For example, you can't include both django.contrib.auth and myproject.auth.

# settings.py

```
MIDDLEWARE = [  
    'django.middleware.security.SecurityMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    'django.middleware.csrf.CsrfViewMiddleware',  
    'django.contrib.auth.middleware.AuthenticationMiddleware',  
    'django.contrib.messages.middleware.MessageMiddleware',  
    'django.middleware.clickjacking.XFrameOptionsMiddleware',  
]
```

A list of middleware to use.

# settings.py

*ROOT\_URLCONF = 'geekyshows.urls'*

A string representing the full Python import path to your root URLconf, for example "mydjangoapps.urls" can be overridden on a per-request basis by setting the attribute urlconf on the incoming HttpRequest object.

# settings.py

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

The template backend to use.

Directories where the engine should look for template source files, in search order.

Whether the engine should look for template source files inside installed applications.

Extra parameters to pass to the template backend. Available parameters vary depending on the template backend.

A list containing the settings for all template engines to be used with Django. Each item of the list is a dictionary containing the options for an individual engine.

# settings.py

`WSGI_APPLICATION = 'geekyshows.wsgi.application'`

The full Python path of the WSGI application object that Django's built-in servers (e.g. runserver) will use.

The django-admin startproject management command will create a standard `wsgi.py` file with an application callable in it, and point this setting to that application.

If not set, the return value of `django.core.wsgi.get_wsgi_application()` will be used.

# settings.py

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.sqlite3',  
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),  
    }  
}
```

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.postgresql',  
        'NAME': 'mydatabase',  
        'USER': 'mydatabaseuser',  
        'PASSWORD': 'mypassword',  
        'HOST': '127.0.0.1',  
        'PORT': '5432',  
    }  
}
```

A dictionary containing the settings for all databases to be used with Django.

It is a nested dictionary whose contents map a database alias to a dictionary containing the options for an individual database.

The DATABASES setting must configure a default database; any number of additional databases may also be specified.

# settings.py

```
AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME': 'django.contrib.auth.password_validation.UserAttributeSimilarityValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.CommonPasswordValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.NumericPasswordValidator',
    },
]
```

The list of validators that are used to check the strength of user's passwords.

# settings.py

*LANGUAGE\_CODE = 'en-us'*

A string representing the language code for this installation. This should be in standard language ID format. For example, U.S. English is "en-us".

*TIME\_ZONE = 'UTC'*

A string representing the time zone for this installation.

*USE\_I18N = True*

A boolean that specifies whether Django's translation system should be enabled. This provides a way to turn it off, for performance. If this is set to False, Django will make some optimizations so as not to load the translation machinery.

*USE\_L10N = True*

A boolean that specifies if localized formatting of data will be enabled by default or not. If this is set to True, e.g. Django will display numbers and dates using the format of the current locale.

# settings.py

*USE\_TZ = True*

A boolean that specifies if datetimes will be timezone-aware by default or not. If this is set to True, Django will use timezone-aware datetimes internally. Otherwise, Django will use naive datetimes in local time.

*STATIC\_URL = '/static/'*

URL to use when referring to static files located in STATIC\_ROOT.

Example: "/static/" or "http://static.example.com/"

If not None, this will be used as the base path for asset definitions (the Media class) and the staticfiles app.

It must end in a slash if set to a non-empty value.

You may need to configure these files to be served in development and will definitely need to do so in production.

# urls.py

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

# manage.py

```
import os
import sys
def main():
    os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'geekyshows.settings')
    try:
        from django.core.management import execute_from_command_line
    except ImportError as exc:
        raise ImportError(
            "Couldn't import Django. Are you sure it's installed and "
            "available on your PYTHONPATH environment variable? Did you "
            "forget to activate a virtual environment?"
        ) from exc
    execute_from_command_line(sys.argv)
if __name__ == '__main__':
    main()
```

# 08-How to run server

Django provides built-in server which we can use to run our project.

**runserver** – This command is used to run built-in server of Django.

Steps:-

- Go to Project Folder
- Then run command ***python manage.py runserver***
- Server Started
- Visit ***http://127.0.0.1:8000*** or ***http://localhost:8000***
- You can specify Port number ***python manage.py runserver 5555***
- Visit ***http://127.0.0.1:5555*** or ***http://localhost:5555***

# How to Stop server

*ctrl + c* is used to stop Server

Note – Sometime when you make changes in your project you may need to restart the server.

# 09-How to Start/Create Application

A Django project contains one or more applications which means we create application inside Project Folder.

Syntax:- `python manage.py startapp appname`

## Creating One Application inside Project:-

- Go to Project Folder
- Run Command *python manage.py startapp course*

## Creating Multiple Applications inside Project:-

- Go to Project Folder
- *python manage.py startapp course*
- *python manage.py startapp fees*
- *python manage.py startapp result*

# How to Install Application in Our Project

As we know a Django project can contain multiple application so just creating application inside a project is not enough we also have to install application in our project.

We install application in our project using settings.py file.

settings.py file is available at Project Level which means we can install all the application of project.

This is compulsory otherwise Application won't be recognized by Django.

Open **settings.py** file

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'application_name1',  
    'application_name2',  
]
```

Save settings.py File

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'course',  
]
```

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'course',  
    'fees',  
    'result',  
]
```

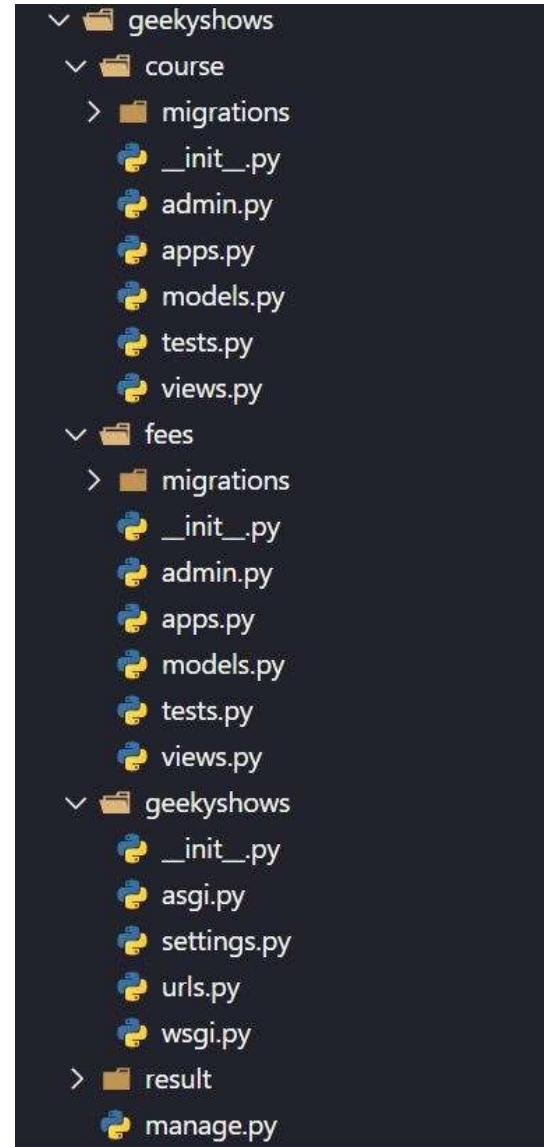
# Geeky Steps

- Create Django Project: *django-admin startproject geekyshows*
- Change Directory to Django Project: *cd geekyshows*
- Create Django Application: *python manage.py startapp course*
- Add/Install Application to Django Project (course to geekyshows)
  - Open *settings.py*
  - Add *course*  
 $INSTALLED_APPS = [\text{'django.contrib.admin'}, \text{'course'}, ]$
  - Save *settings.py*



# Geeky Steps

- Create Django Project: *django-admin startproject geekyshows*
- Change Directory to Django Project: *cd geekyshows*
- Create Django Application1: *python manage.py startapp course*
- Create Django Application2: *python manage.py startapp fees*
- Create Django Application3: *python manage.py startapp result*
- Add/Install Applications to Django Project
  - Open *settings.py*
  - Add *course, fees, result*  
*INSTALLED\_APPS = ['django.contrib.admin', 'course', 'fees', 'result', ]*
  - Save *settings.py*



# 10-Application Directory Structure

- Go to Project Folder
- Run Command *python manage.py startapp course*

**course**

**migrations**

`__init__.py`

`__init__.py`

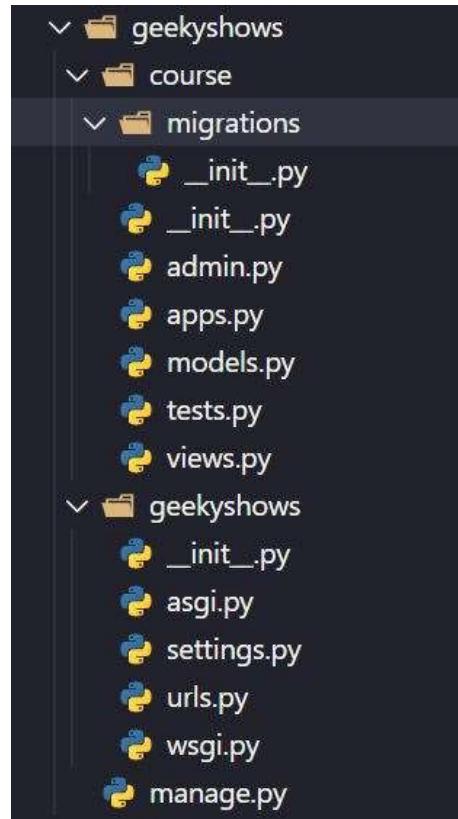
`admin.py`

`apps.py`

`models.py`

`tests.py`

`views.py`



# Application Directory Structure

**migrations** – This folder contains `__init__.py` file which means it's a python package. It also contains all files which are created after running `makemigration` command .

**`__init__.py`** – The folder which contains `__init__.py` file is considered as Python Package.

**`admin.py`** – This file is used to register sql tables so we could perform CRUD operation from Admin Application. Admin Application is provided by Django to perform CRUD operation.

**`apps.py`** – This file is used to config app.

**`models.py`** – This file is used to create our own model class later these classes will be converted into database table by Django for our application.

**`tests.py`** – This is files is used to create tests.

**`views.py`** – This file is used to create view. We write all the business logic related code in this file.

# Django

M – Model

V – View

T – Template

# 11-View

- Function Based View
- Class Based View

# Function Based View

A function based view, is a Python function that takes a Web request and returns a Web response.

This response can be the HTML contents of a Web page, or a redirect, or a 404 error, or an XML document, or an image or anything.

Each view function takes an `HttpRequest` object as its first parameter.

The view returns an `HttpResponse` object that contains the generated response. Each view function is responsible for returning an `HttpResponse` object.

We will call these functions as *view function* or *view function of application or view*.

Syntax:-

```
def function_name (request):  
    return HttpResponse('html/variable/text')
```

It's a Class

HttpResponse Object

HttpRequest Object

# Function Based Views

We use **views.py** file of the application to write functions which may contain business logic of application, later it required to define url name for this function in the **urls.py** file of the project.

Syntax:-

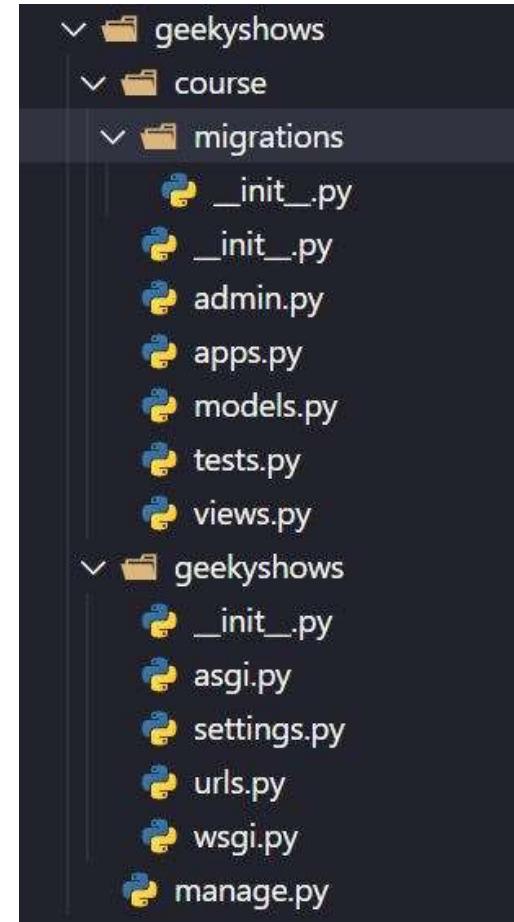
**views.py**

```
def function_name1 (request):  
    return HttpResponse('html/variable/text')
```

HttpRequest Object

HttpResponse Object

```
def function_name2 (request):  
    return HttpResponse('html/variable/text')
```



# Function Based Views

Syntax:-

```
def function_name (request):  
    return HttpResponse('html/variable/text')
```

HttpRequest Object

HttpResponse Object

Where **HttpResponse** is class which is in **django.http** module so we have to import it before using **HttpResponse**.

## views.py

```
from django.http import HttpResponse  
  
def learn_django(request):  
    return HttpResponse('Hello Django')  
  
def learn_python(request):  
    return HttpResponse('<h1>Hello Python</h1>')
```

```
def learn_var(request):  
    a = '<h1>Hello Variable</h1>'  
    return HttpResponse(a)
```

```
def learn_math(request):  
    a = 10 + 10  
    return HttpResponse(a)
```

# Function Based Views

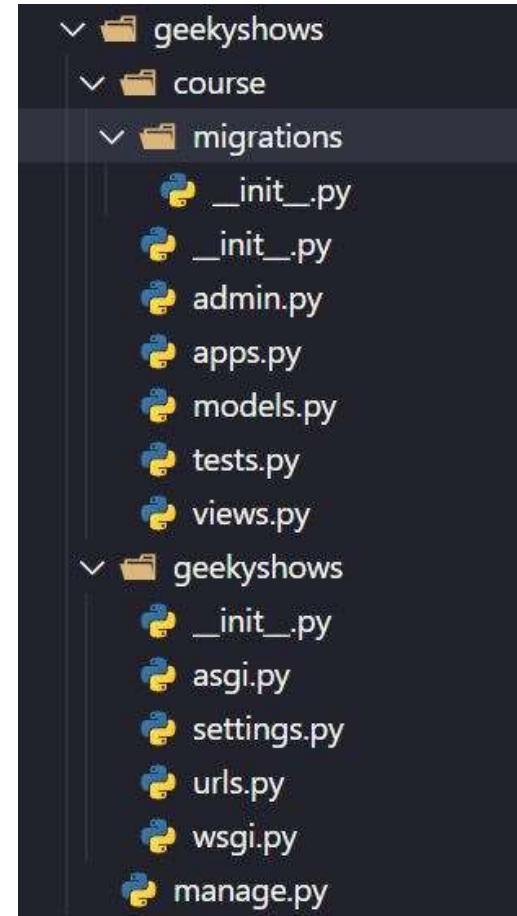
## Single Application with Single view function

### views.py

```
from django.http import HttpResponse  
  
def learn_django(request):  
    return HttpResponse('Hello Django')
```

### urls.py

```
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('learndj/', views.learn_django),  
]
```



# Function Based Views

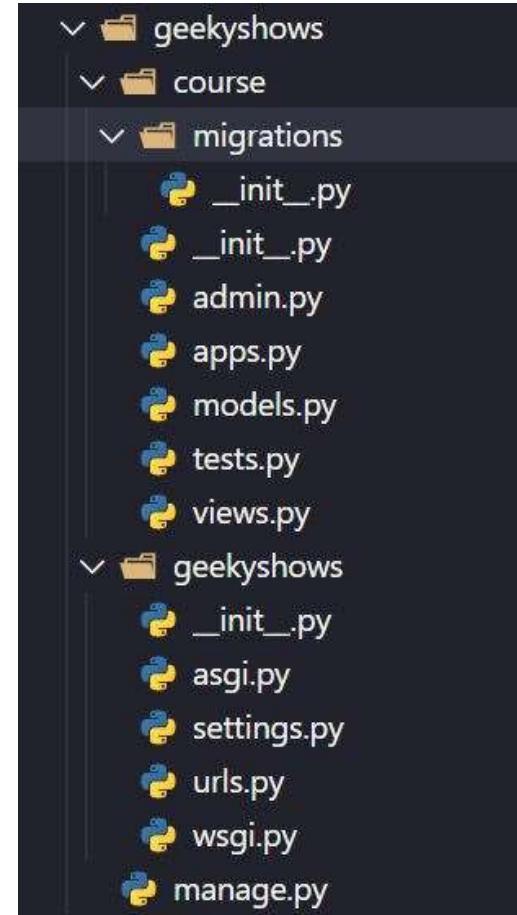
## Single Application with multiple view functions.

### views.py

```
from django.http import HttpResponse  
  
def learn_django(request):  
    return HttpResponse('Hello Django')  
  
def learn_python(request):  
    return HttpResponse('<h1>Hello Python</h1>')
```

### urls.py

```
urlpatterns = [  
    path('learndj/', views.learn_django),  
    path('learnpy/', views.learn_python),  
]
```



# Geeky Steps

- Create Django Project: *django-admin startproject geekyshows*
- Change Directory to Django Project: *cd geekyshows*
- Create Django Application: *python manage.py startapp course*
- Add/Install Application to Django Project (course to geekyshows) using *settings.py* file  
*INSTALLED\_APPS = ['django.contrib.admin', 'course', ]*
- Write View Function inside **views.py**
  - Open **views.py**
  - Import `HttpResponse` class from `django.http` module  
*from django.http import HttpResponse*
  - Write view Function  
*def learn\_django(request):*  
*return HttpResponse('Hello Django')*
  - Save *views.py*



# 12-URL Dispatcher

To design URLs for app, you create a Python module informally named urls.py This module is pure Python code and is a mapping between URL path expressions to view functions.

This mapping can be as short or as long as needed.

It can reference other mappings.

It's pure Python code so it can be constructed dynamically.

## urls.py

```
urlpatterns = [  
    path(route, view, kwargs=None, name=None)  
]
```

## urls.py

```
urlpatterns = [  
    path('learndj/', views.learn_django),  
]
```

# path()

path(route, view, kwargs=None, name=None) - It returns an element for inclusion in urlpatterns.

Where,

- The route argument should be a string or gettext\_lazy() that contains a URL pattern. The string may contain angle brackets e.g. <username> to capture part of the URL and send it as a keyword argument to the view. The angle brackets may include a converter specification like the int part of <int:id> which limits the characters matched and may also change the type of the variable passed to the view. For example, <int:id> matches a string of decimal digits and converts the value to an int.
- The view argument is a view function or the result of as\_view() for class-based views. It can also be an django.urls.include().
- The kwargs argument allows you to pass additional arguments to the view function or method. It should be a dictionary.
- name is used to perform URL reversing.

# path ()

## urls.py

```
urlpatterns = [  
    path(route, view, kwargs=None, name=None)  
]
```

## urls.py

```
urlpatterns = [  
    path('learndj/', views.learn_Django, {'check': 'OK'}, name='learn_django'),  
]
```

# URL Pattern inside Project

*urls.py* file is used to define url pattern attached with application or view of application or view function of application.

*urls.py* file is located inside *inner project folder* not inside *application folder* which means we define url at project level for applications. Defined url name will be used by application user to get response from the application or view function of application.

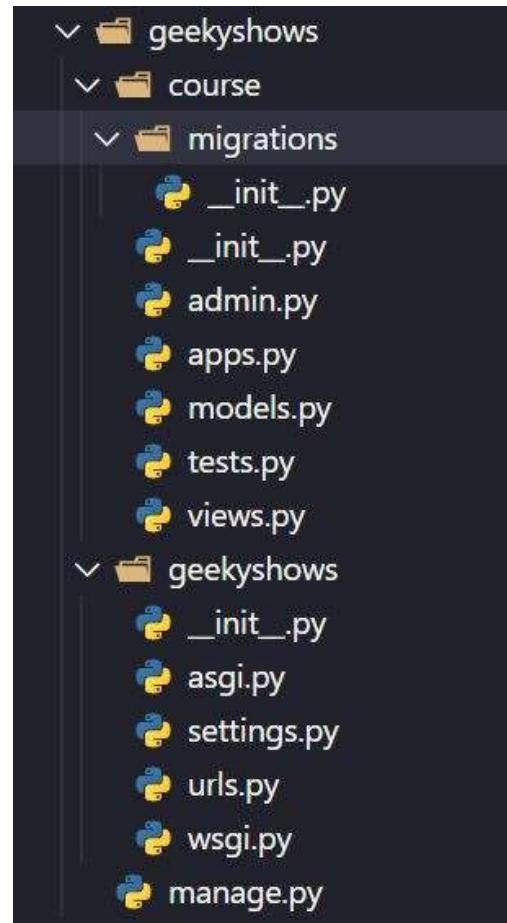
Steps:-

- Open *urls.py*
- Import Module (Python file) of the application
- Write URL Name and Map it with function

```
urlpatterns = [  
    path(route, view, kwargs=None, name=None)  
]  
  
from course import views  
  
urlpatterns = [  
    path('learndj/', views.learn_django),  
]
```

*learndj* is mapped with *learn\_django* function which is inside *views.py* file.

<http://127.0.0.1:8000/learndj>



# URL Pattern inside Project

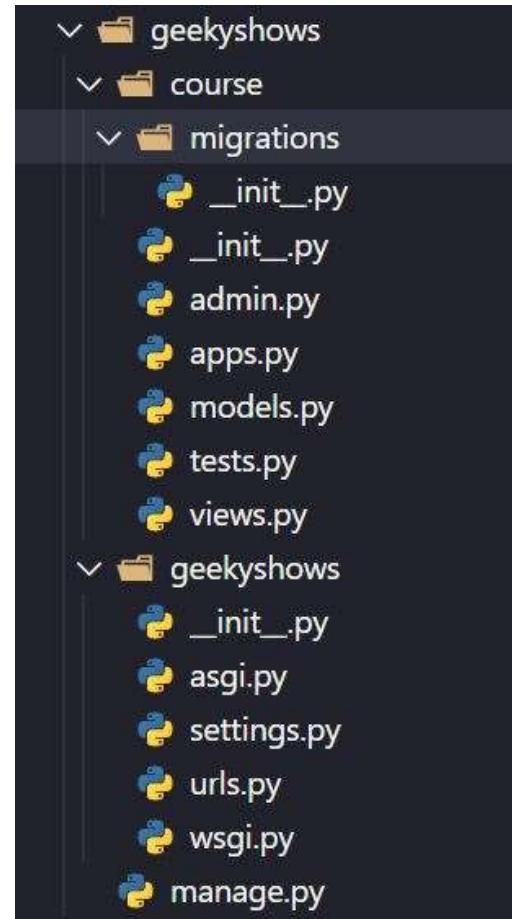
## Single Application with Single function.

### views.py

```
from django.http import HttpResponse  
  
def learn_django(request):  
  
    return HttpResponse('Hello Django ')
```

### urls.py

```
from course import views  
  
urlpatterns = [  
    path('learndj/', views.learn_django),  
]  
  
http://127.0.0.1:8000/learndj
```



# URL Pattern inside Project

We can define multiple url for one view function. Which means we can access same view function with multiple urls.

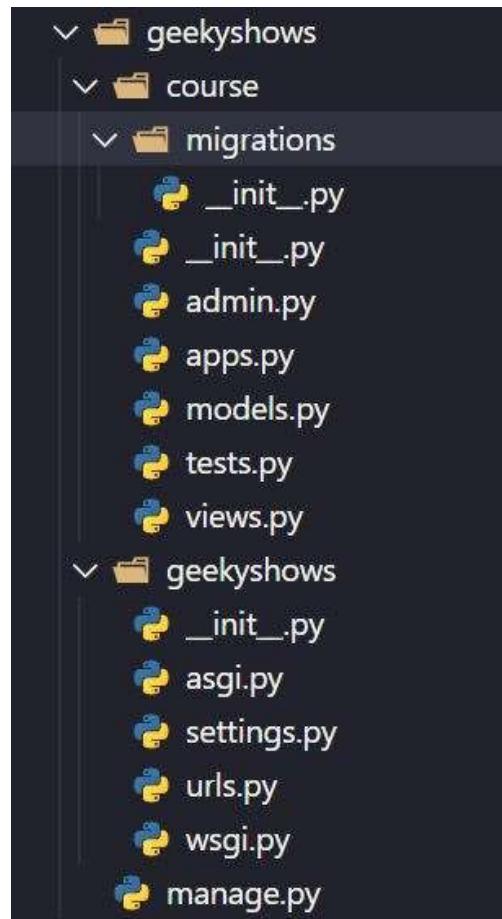
## urls.py

```
from course import views
```

```
urlpatterns = [
    path('learndj/', views.learn_django),
    path('altlearndj/', views.learn_django),
]
```

<http://127.0.0.1:8000/learndj>

<http://127.0.0.1:8000/altlearndj>



# URL Pattern inside Project

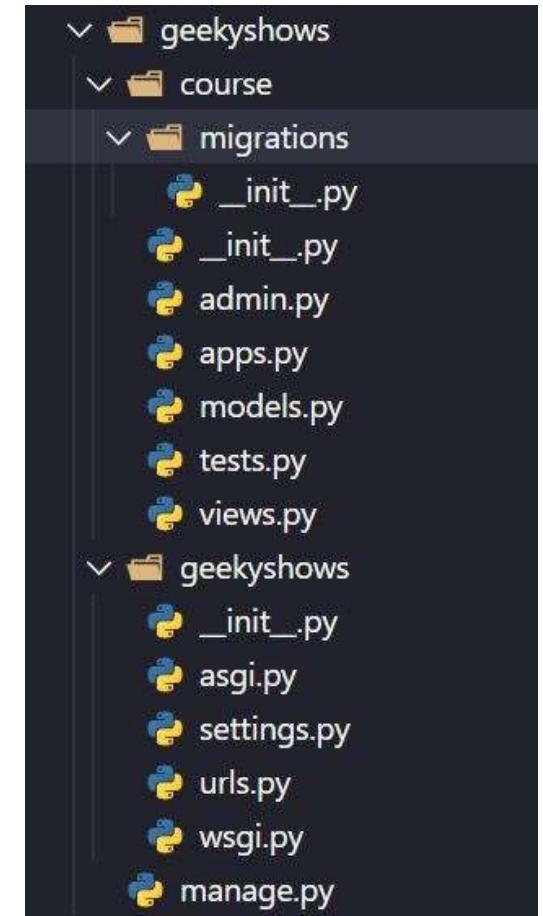
## Single Application with multiple functions.

### views.py

```
from django.http import HttpResponseRedirect  
  
def learn_django(request):  
    return HttpResponseRedirect('Hello Django')  
  
def learn_python(request):  
    return HttpResponseRedirect('<h1>Hello Python</h1>')
```

### urls.py

```
from course import views  
  
urlpatterns = [  
    path('learndj/', views.learn_django),  
    path('learnpy/', views.learn_python),  
]
```

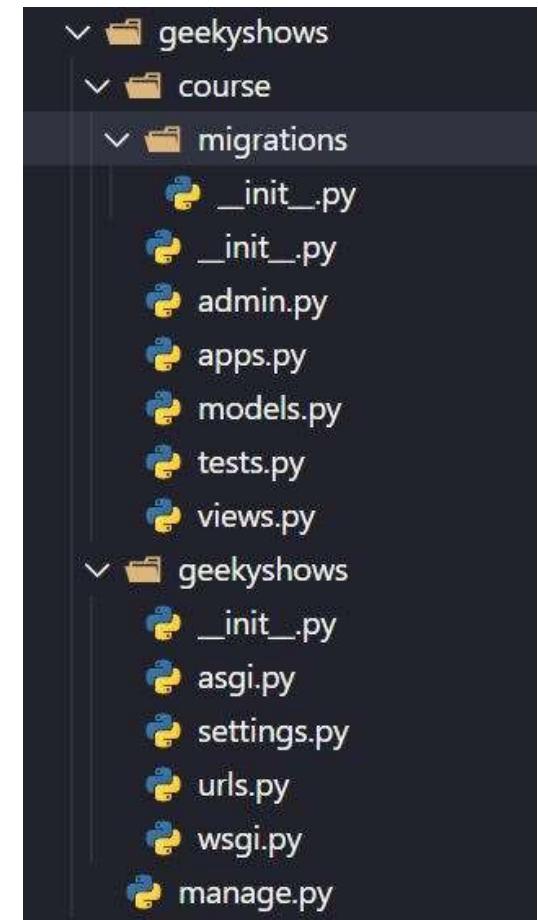


http://127.0.0.1:8000/learndj

http://127.0.0.1:8000/learnpy

# Geeky Steps

- Create Django Project: *django-admin startproject geekyshows*
- Change Directory to Django Project: *cd geekyshows*
- Create Django Application: *python manage.py startapp course*
- Add/Install Application to Django Project (course to geekyshows) using settings.py file INSTALLED\_APPS
- Write View Function inside views.py file
- Define url for view function of application
  - Open *urls.py*
  - Import views Module of the application  
`from course import views`
  - Write url Pattern  
`urlpatterns = [ path('learndj/', views.learn_django),  
 path('learnpy/', views.learn_python), ]`
  - Save urls.py



# How URL Dispatcher Works

- Django determines the root URLconf (`urls.py`) module to use. Ordinarily, this is the value of the `ROOT_URLCONF` setting, but if the incoming `HttpRequest` object has a `urlconf` attribute (set by middleware), its value will be used in place of the `ROOT_URLCONF` setting.
- Django loads that Python module and looks for the variable `urlpatterns`. This should be a sequence of `django.urls.path()` and/or `django.urls.re_path()` instances.
- Django runs through each URL pattern, in order, and stops at the first one that matches the requested URL, matching against `path_info`.
- Once one of the URL patterns matches, Django imports and calls the given view, which is a Python function (or a class-based view).
  - An instance of `HttpRequest`.
  - If the matched URL pattern contained no named groups, then the matches from the regular expression are provided as positional arguments.
  - The keyword arguments are made up of any named parts matched by the path expression that are provided, overridden by any arguments specified in the optional `kwargs` argument to `django.urls.path()` or `django.urls.re_path()`.
  - If no URL pattern matches, or if an exception is raised during any point in this process, Django invokes an appropriate error-handling view.

# re\_path( )

re\_path(route, view, kwargs=None, name=None) - It returns an element for inclusion in urlpatterns.

Where,

- The route argument should be a string or gettext\_lazy() that contains a regular expression compatible with Python's re module. Strings typically use raw string syntax (r") so that they can contain sequences like \d without the need to escape the backslash with another backslash. When a match is made, captured groups from the regular expression are passed to the view as named arguments if the groups are named, and as positional arguments otherwise. The values are passed as strings, without any type conversion.
- The view argument is a view function or the result of as\_view() for class-based views. It can also be an django.urls.include().
- The kwargs argument allows you to pass additional arguments to the view function or method.
- name is used to perform URL reversing.

# re\_path ()

## urls.py

```
urlpatterns = [  
    re_path(route, view, kwargs=None, name=None)  
]
```

## urls.py

```
urlpatterns = [  
    path(r'^learndj/$', views.learn_Django, {'check': 'OK'}, name='learn_django'),  
]
```

# 13-Multiple Applications inside Project

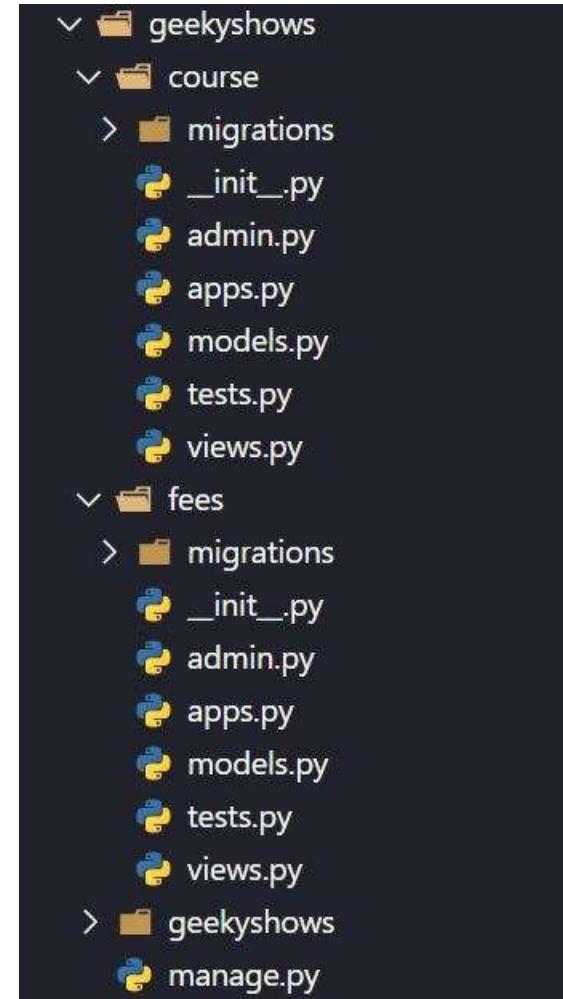
Syntax:- `python manage.py startapp appname`

## Creating One Application inside Project Steps:-

- Go to Project Folder (`cd geekyshows`)
- Run Command `python manage.py startapp course`

## Creating Multiple Applications inside Project Steps:-

- Go to Project Folder (`cd geekyshows`)
- `python manage.py startapp course`
- `python manage.py startapp fees`



# How to Install Application in Our Project

As we know a project can contain multiple application so just creating application inside a project is not enough we also have to install application in our project.

We install application in our project using setting.py file.

setting.py file is available at Project Level which means we can install all the application of project.

This is compulsory otherwise Application won't be recognized by Django.

Open setting.py file

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'application_name1',  
    'application_name2',  
]
```

Save settings.py File

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'course',  
]
```

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'course',  
    'fees',  
]
```

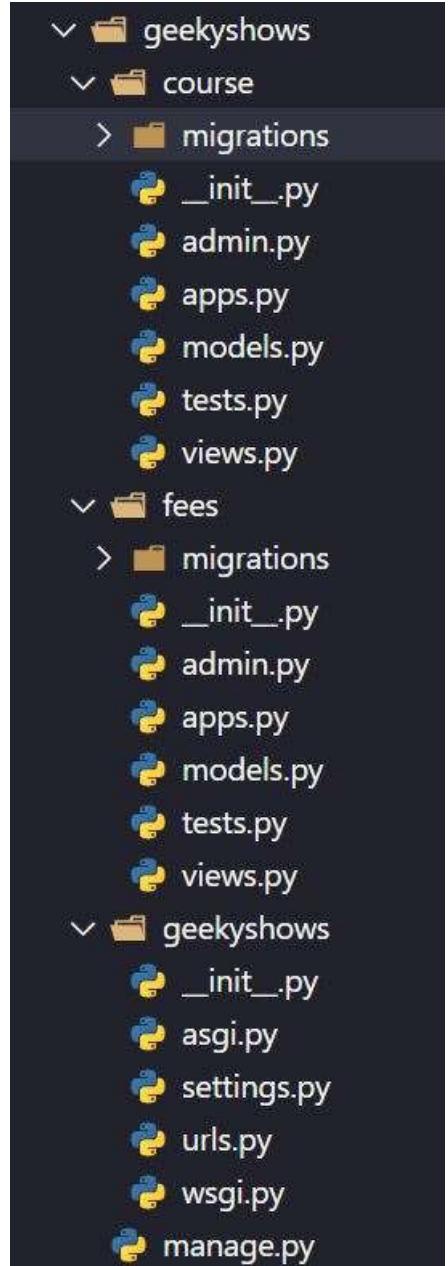
# Function Based Views

## views.py of course

```
from django.http import HttpResponse  
  
def learn_django(request):  
  
    return HttpResponse('Hello Django')
```

## views.py of fees

```
from django.http import HttpResponse  
  
def fees_django(request):  
  
    return HttpResponse('300')
```



# URL Pattern

## views.py of course

```
from django.http import HttpResponse  
  
def learn_django(request):  
  
    return HttpResponse('Hello Django')
```

## views.py of fees

```
from django.http import HttpResponse  
  
def fees_django(request):  
  
    return HttpResponse('300')
```

## urls.py

```
from course import views  
  
from fees import views  
  
urlpatterns = [  
  
    path('learndj/', views.learn_django),  
  
    path('feesdj/', views.fees_django),  
]
```



# URL Pattern

## views.py of course

```
from django.http import HttpResponse
def learn_django(request):
    return HttpResponse('Hello Django')
```

## views.py of fees

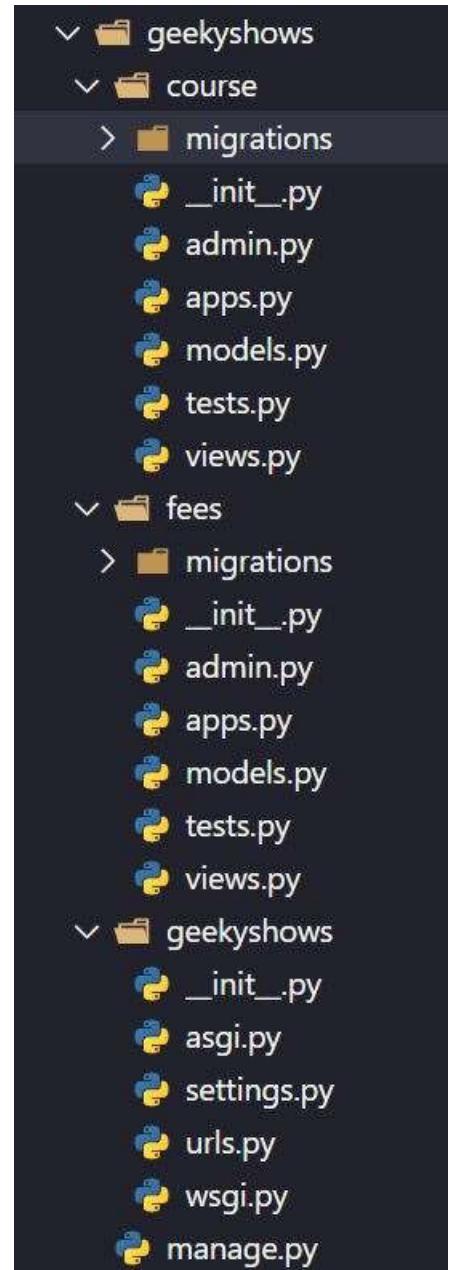
```
from django.http import HttpResponse
def fees_django(request):
    return HttpResponse('300')
```

## urls.py

```
from course import views as cv
from fees import views as fv
urlpatterns = [
    path('learndj/', cv.learn_django),
    path('feesdj/', fv.fees_django),
]
```

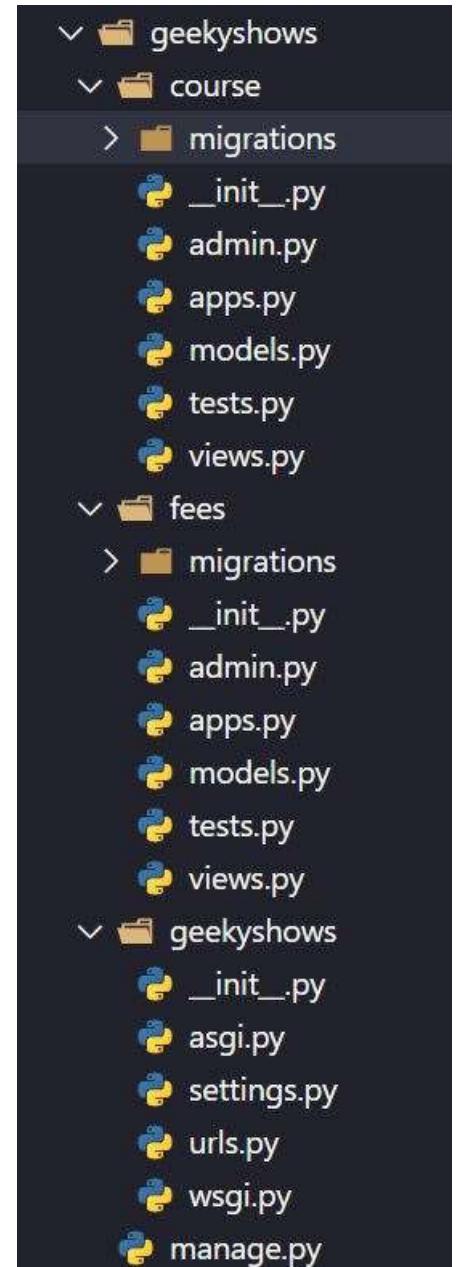
## urls.py

```
from course.views import learn_django
from fees.views import fees_django
urlpatterns = [
    path('learndj/', learn_django),
    path('feesdj/', fees_django),
]
```



# Geeky Steps

- Create Django Project: *django-admin startproject geekyshows*
- Change Directory to Django Project: *cd geekyshows*
- Create Django Application1: *python manage.py startapp course*
- Create Django Application2: *python manage.py startapp fees*
- Add/Install Applications to Django Project (course and fees to geekyshows) using settings.py INSTALLED\_APPS
- Write View Function inside views.py file
- Define url for view function of application using urls.py file



# 14-URL Rewrite Paths inside Project

## views.py of course

```
from django.http import HttpResponse  
  
def learn_django(request):  
    return HttpResponse('Hello Django')
```

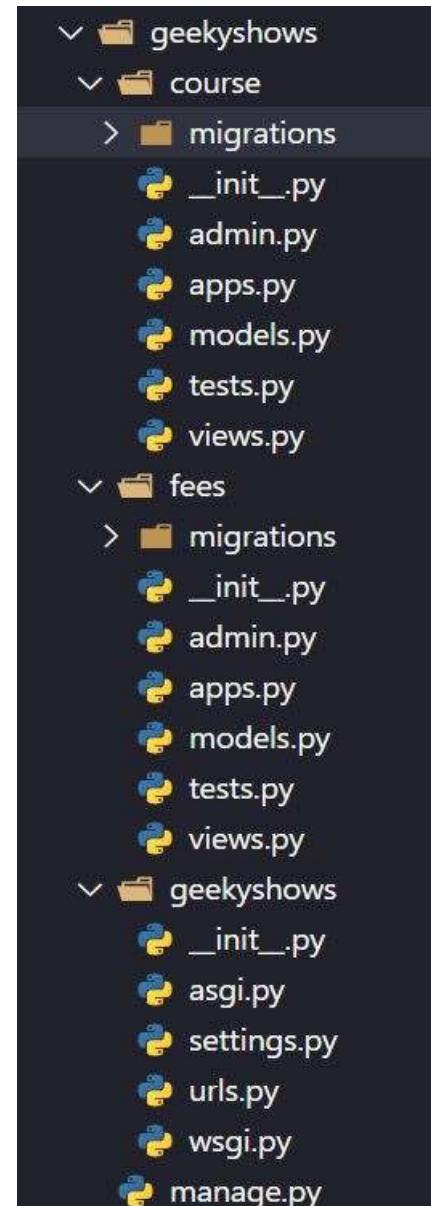
## views.py of fees

```
from django.http import HttpResponse  
  
def fees_django(request):  
    return HttpResponse('300')
```

## urls.py

```
from course import views as cv  
  
from fees import views as fv  
  
urlpatterns = [  
    path('learndj/', cv.learn_django),  
    path('feesdj/', fv.fees_django),
```

J



# Why URL Pattern inside Application

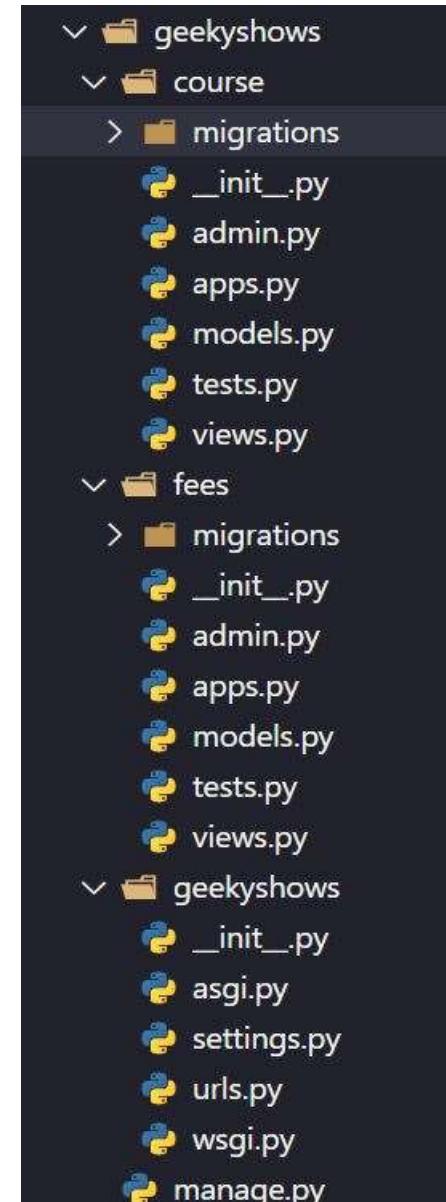
So far we have learnt to define url pattern at project level for our all application.

This approach increases the dependency of applications in project which means if we want to use a particular application for our another project we may face issues.

Our each application should be independent or less depend on project so we could use our applications in different projects easily without worrying about urls.py available in Project Folder.

Following are the benefits of defining url pattern inside Application

- Reduces the dependency of Application
- Enhance Application performance.
- Reusability of application becomes easy.



# include ( )

A function that takes a full Python import path to another URLconf module that should be “included” in this place. Optionally, the application namespace and instance namespace where the entries will be included into can also be specified.

include() also accepts as an argument either an iterable that returns URL patterns or a 2-tuple containing such iterable plus the names of the application namespaces.

urlpatterns can “include” other URLconf modules.

Syntax:-

```
include(module, namespace=None)
```

```
include(pattern_list)
```

```
include((pattern_list, app_namespace), namespace=None)
```

Where,

module – URLconf module (or module name)

namespace (str) – Instance namespace for the URL entries being included

pattern\_list – Iterable of path() and/or re\_path() instances.

app\_namespace (str) – Application namespace for the URL entries being included

# include ( )

Syntax:-

```
include(module, namespace=None)
```

```
include(pattern_list)
```

```
include((pattern_list, app_namespace), namespace=None)
```

Example:-

```
from django.urls import include, path
urlpatterns = [
    path('cor/', include('course.urls')),
]
urlpatterns = [
    path('cor/', include([
        path('learndj/', views.learn_django),
        path('learnpy/', views.learn_python)
    ])),
]
```

path(route, view, kwargs=None, name=None)

The view argument is a view function or the result of as\_view() for class-based views. It can also be an django.urls.include().

```
otherpatterns = [
```

```
    path('learndj/', views.learn_django),
```

```
    path('learnpy/', views.learn_python),
```

```
]
```

```
urlpatterns = [
```

```
    path('cor/', include(otherpatterns)),
```

```
]
```

# Write URL Pattern inside Application

- Create an urls.py file inside each application (in case multiple application).
- Write all url pattern related to application, in urls.py file available inside application.
- Include Application's urls.py file inside Project's urls.py file.

## **views.py in Application Folder**

```
from django.http import HttpResponseRedirect
def learn_django(request):
    return HttpResponseRedirect('Hello Django')
def learn_python(request):
    return HttpResponseRedirect('<h1>Hello Python</h1>')
```

## **urls.py in Application Folder**

```
from course import views
urlpatterns = [
    path('learndj/', views.learn_django),
    path('learnpy/', views.learn_python),
]
```

## **urls.py in Project Folder**

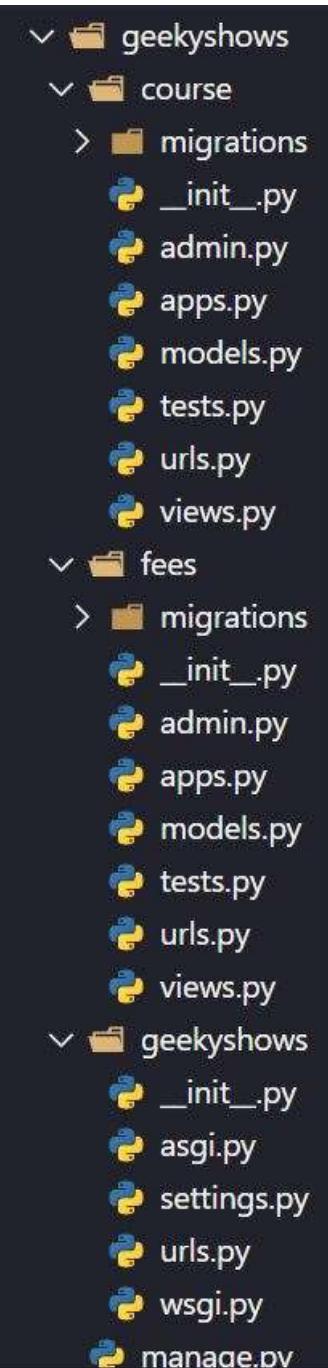
```
from django.urls import path, include
urlpatterns = [
    path('cor/', include('course.urls')),
]
```

Package Name

Module Name

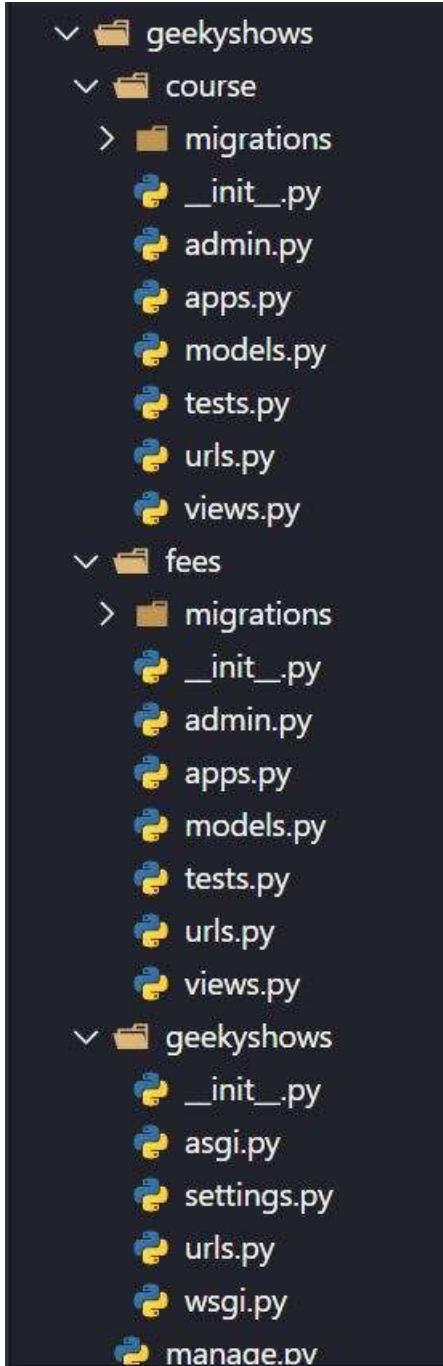
http://127.0.0.1:8000/cor/learndj

http://127.0.0.1:8000/cor/learnpy



# Geeky Steps

- Create Django Project: *django-admin startproject geekyshows*
- Change Directory to Django Project: *cd geekyshows*
- Create Django Application: *python manage.py startapp course*
- Add/Install Application to Django Project using settings.py file  
**INSTALLED\_APPS**
- Write View Function inside views.py file
- Create an **urls.py** file inside each application (in case multiple application).
- Write all url pattern related to application, in urls.py file available inside application.
- Include Application's urls.py file inside Project's urls.py file.



# 15-Template

A template is a text file. It can generate any text-based format (HTML, XML, CSV, etc.).

A template contains variables, which get replaced with values when the template is evaluated, and tags, which control the logic of the template.

Template is used by view function to represent the data to user.

User sends request to view then view contact template afterthat view get information from the template and then view gives response to the users.

# 16-Create Template Folder and Files

We create **templates** folder inside Project Folder. **templates** folder will contain all html files.

geekyshows

templates ← This is templates Folder.

geekyshows

\_\_init\_\_.py

settings.py

urls.py

wsgi.py

manage.py

course

fees

geekyshows

templates

courseone.html

coursetwo.html

feesone.html

feestwo.html

Template files

geekyshows

\_\_init\_\_.py

settings.py

urls.py

wsgi.py

manage.py

course

fees

# Add Templates in settings.py

geekyshows

**templates**

**courseone.html**

**coursetwo.html**

**feesone.html**

**feestwo.html**

geekyshows

**\_\_init\_\_.py**

**settings.py**

**urls.py**

**wsgi.py**

**manage.py**

**course**

**fees**

## settings.py

Old Version Django 3.0:

TEMPLATES\_DIR = os.path.join(BASE\_DIR, 'templates')

TEMPLATES\_DIR = BASE\_DIR / 'templates'

INSTALLED\_APPS = [  
 'course',  
 'fees'  
]

TEMPLATES = [  
 {  
 'DIRS': [TEMPLATES\_DIR],  
 }  
]

Django Version 3.1

Directories where the engine  
should look for template  
source files, in search order.

# Geeky Steps

- Create Django Project: *django-admin startproject geekyshows*
- Create Django Application1: *python manage.py startapp course*
- Create Django Application2: *python manage.py startapp fees*
- Add/Install Applications to Django Project (course and fees to geekyshows) using settings.py INSTALLED\_APPS
- Create **templates** folder inside Root Project Directory
- Add **templates** directory in settings.py

```
TEMPLATES_DIR = os.path.join(BASE_DIR, 'templates')
```

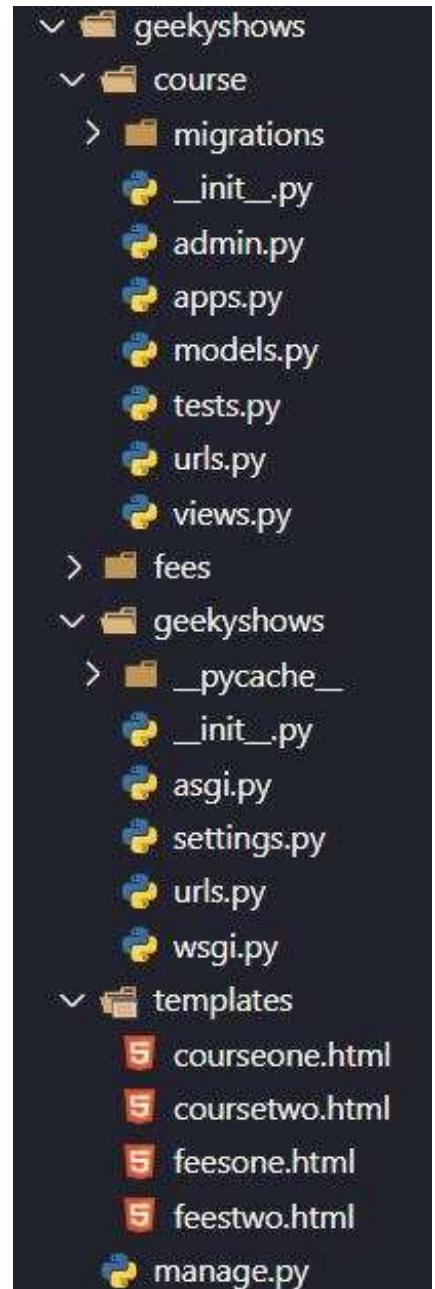
```
TEMPLATES_DIR = BASE_DIR / 'templates'
```

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

Old Django Version 3.0

Django Version 3.1

- Create template files inside **templates** folder
- Write View Function inside views.py file
- Define url for view function of application using urls.py file



# Write Templates Files

When we create Template file for application we separate business logic and presentation from the application *views.py* file.

Now we will write business logic in *views.py* file and presentation code in template file.

*templates*

*courseone.html*

```
<html>
  <head>
    <style> ..... </style>
  </head>
  <body>
    <h1>Hello Django</h1>
  </body>
  <script>.....</script>
</html>
```

*templates*

*coursetwo.html*

```
<html>
  <body>
    <h1>Hello Python</h1>
  </body>
</html>
```

*templates*

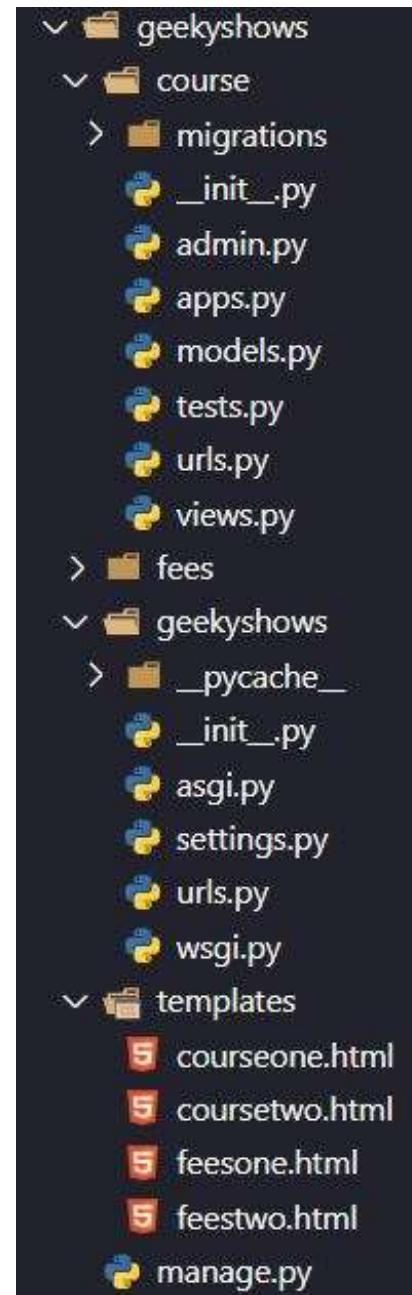
*feesone.html*

```
<html>
  <body>
    <h1>300</h1>
  </body>
</html>
```

*templates*

*feestwo.html*

```
<html>
  <body>
    <h1>200</h1>
  </body>
</html>
```



# Rendering Templates Files

By Creating Template file for application we separate business logic and presentation from the application *views.py* file. Now we will write business logic in *views.py* file and presentation code in html file.

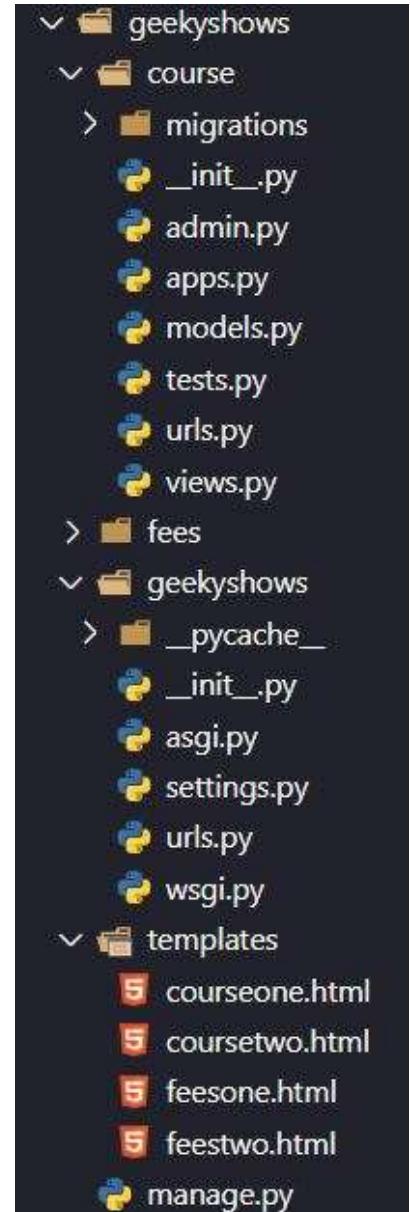
Still *views.py* will be responsible to process the template files for this we will use *render()* function in *views.py* file.

## views.py

```
from django.shortcuts import render

def function_name(request):
    Dynamic Data, if else, any python code logic
    return render(request, template_name, context=dict_name,
                  content_type=MIME_type, status=None, using=None)

def learn_django(request):
    return render(request, 'courseone.html')
```



# render ( )

render ( ) Function - It combines a given template with a given context dictionary and returns an `HttpResponse` object with that rendered text.

Syntax:-

```
render(request, template_name, context=dict_name, content_type=MIME_type, status=None, using=None)
```

Where,

`request` – The request object used to generate this response.\*

`template_name` – The full name of a template to use or sequence of template names. If a sequence is given, the first template that exists will be used. \*

`context` – A dictionary of values to add to the template context. By default, this is an empty dictionary. If a value in the dictionary is callable, the view will call it just before rendering the template.

`content_type` – The MIME type to use for the resulting document. Defaults to 'text/html'.

`status` – The status code for the response. Defaults to 200.

`using` – The NAME of a template engine to use for loading the template.

# **render ( )**

Syntax:-

```
render(request, template_name, context=dict_name, content_type=MIME_type, status=None, using=None)
```

Example:-

```
render(request, 'courseone.html', context=cname, content_type='application/xhtml+xml')
```

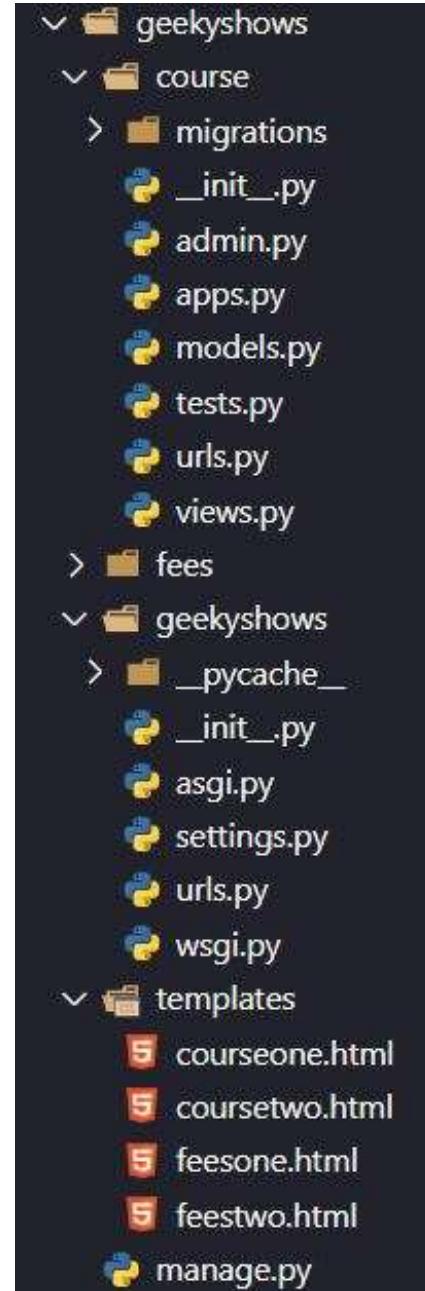
*templates/*

*courseone.html*

```
<html>
  <head>
    <style> .....
  </head>
  <body>
    <h1>Hello Django</h1>
  </body>
  <script>.....
</html>
```

**views.py**

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



# Geeky Steps

- Create Django Project: *django-admin startproject geekyshows*
- Create Django Application1: *python manage.py startapp course*
- Create Django Application2: *python manage.py startapp fees*
- Add/Install Applications to Django Project (course and fees to geekyshows) using settings.py INSTALLED\_APPS
- Create **templates** folder inside Root Project Directory
- Add **templates** directory in settings.py

```
TEMPLATES_DIR = os.path.join(BASE_DIR, 'templates')
```

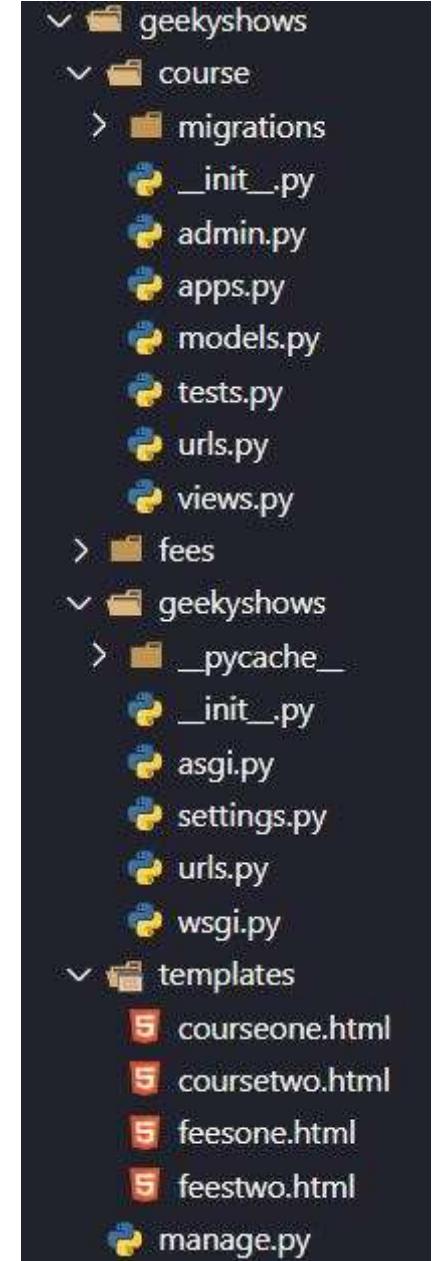
```
TEMPLATES_DIR = BASE_DIR / 'templates'
```

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

Old Django Version 3.0

Django Version 3.1

- Create template files inside templates folder
- Write View Function inside views.py file
- Define url for view function of application using urls.py file
- Write Template files code



# Create Template Folder and Files

We can create separate folder inside templates folder for applications then each application will contain only those html file which are related to them. This will enhance readability and separate html files according to applications.

geekyshows

templates

courseone.html  
coursetwo.html  
feesone.html  
feestwo.html

geekyshows

\_\_init\_\_.py  
settings.py  
urls.py  
wsgi.py

manage.py

course

fees

geekyshows

templates

course  
fees

html files related to course app

html files related to fees app

geekyshows

\_\_init\_\_.py  
settings.py  
urls.py  
wsgi.py

manage.py

course

fees

Its naming convention but not compulsory to write name as application name

# Create Template Folder and Files

geekyshows

**templates**

**courseone.html**  
**coursetwo.html**  
**feesone.html**  
**feestwo.html**

geekyshows

**\_\_init\_\_.py**  
**settings.py**  
**urls.py**  
**wsgi.py**

**manage.py**

**course**

**fees**

geekyshows

**templates**

**course**

**courseone.html**  
**coursetwo.html**

**fees**

**feesone.html**  
**feestwo.html**

geekyshows

**\_\_init\_\_.py**  
**settings.py**  
**urls.py**  
**wsgi.py**

**manage.py**

**course**

**fees**

These files are  
called as  
template files

# Add Templates in settings.py

geekyshows

templates

course

courseone.html

coursetwo.html

fees

feesone.html

feestwo.html

geekyshows

\_\_init\_\_.py

settings.py

urls.py

wsgi.py

manage.py

course

fees

## settings.py

Old Version Django 3.0:

```
TEMPLATES_DIR = os.path.join(BASE_DIR, 'templates')
```

```
TEMPLATES_DIR = BASE_DIR / 'templates'
```

```
INSTALLED_APPS = [  
    'course',  
    'fees'  
]
```

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

Django Version 3.1

Directories where the engine  
should look for template  
source files, in search order.

# Geeky Steps

- Create Django Project: *django-admin startproject geekyshows*
- Create Django Application1: *python manage.py startapp course*
- Create Django Application2: *python manage.py startapp fees*
- Add/Install Applications to Django Project (course and fees to geekyshows) using settings.py INSTALLED\_APPS
- Create **templates** folder inside Root Project Directory
- Add **templates** directory in settings.py

```
TEMPLATES_DIR = os.path.join(BASE_DIR, 'templates')
```

```
TEMPLATES_DIR = BASE_DIR / 'templates'
```

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

Old Django Version 3.0

Django Version 3.1

- Create Separate Directory for each application, inside templates directory
- Create template files inside templates/temp\_application folder/directory
- Write View Function inside views.py file
- Define url for view function of application using urls.py file



# Write Templates Files

When we create Template file for application we separate business logic and presentation from the application *views.py* file.

Now we will write business logic in *views.py* file and presentation code in template file.

*templates/course*

*courseone.html*

```
<html>
  <head>
    <style> ..... </style>
  </head>
  <body>
    <h1>Hello Django</h1>
  </body>
  <script>.....</script>
</html>
```

*templates/course*

*coursetwo.html*

```
<html>
  <body>
    <h1>Hello Python</h1>
  </body>
</html>
```

*templates/fees*

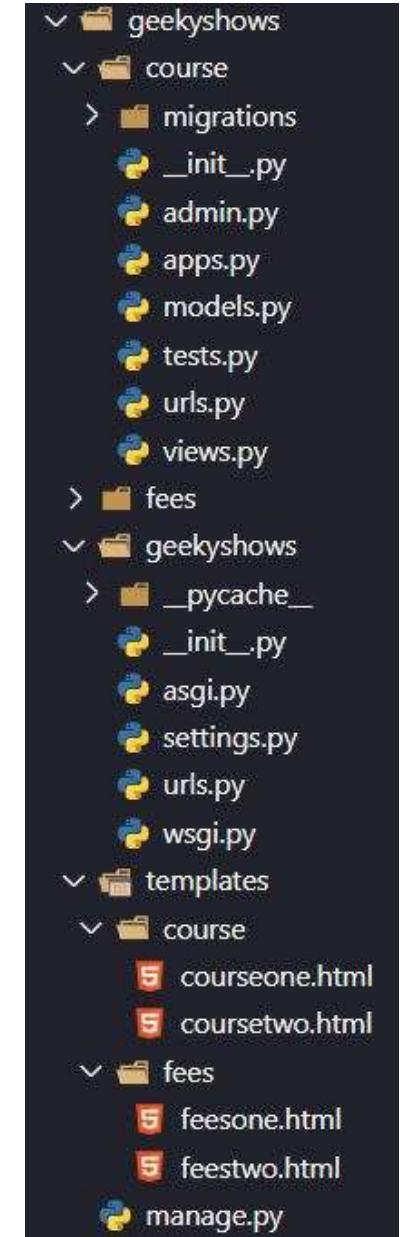
*feestwo.html*

```
<html>
  <body>
    <h1>200</h1>
  </body>
</html>
```

*templates/fees*

*feestwo.html*

```
<html>
  <body>
    <h1>300</h1>
  </body>
</html>
```



# Rendering Templates Files

By Creating Template file for application we separate business logic and presentation from the application *views.py* file. Now we will write business logic in *views.py* file and presentation code in html file.

Still *views.py* will be responsible to process the template files for this we will use *render()* function in *views.py* file.

## views.py

```
from django.shortcuts import render

def function_name(request):
    Dynamic Data, if else, any python code logic
    return render(request, template_name, context=dict_name,
                  content_type=MIME_type, status=None, using=None)

def learn_django(request):
    return render(request, 'course/courseone.html')
```



# render ( )

render ( ) Function - It combines a given template with a given context dictionary and returns an `HttpResponse` object with that rendered text.

Syntax:-

```
render(request, template_name, context=dict_name, content_type=MIME_type, status=None, using=None)
```

Where,

`request` – The request object used to generate this response.\*

`template_name` – The full name of a template to use or sequence of template names. If a sequence is given, the first template that exists will be used. \*

`context` – A dictionary of values to add to the template context. By default, this is an empty dictionary. If a value in the dictionary is callable, the view will call it just before rendering the template.

`content_type` – The MIME type to use for the resulting document. Defaults to 'text/html'.

`status` – The status code for the response. Defaults to 200.

`using` – The NAME of a template engine to use for loading the template.

# **render ( )**

Syntax:-

```
render(request, template_name, context=dict_name, content_type=MIME_type, status=None, using=None)
```

Example:-

```
render(request, 'courseone.html', context=cname, content_type='application/xhtml+xml')
```

```
render(request, 'course/courseone.html', context=cname, content_type='application/xhtml+xml')
```

```
render(request, 'course/coursetwo.html')
```

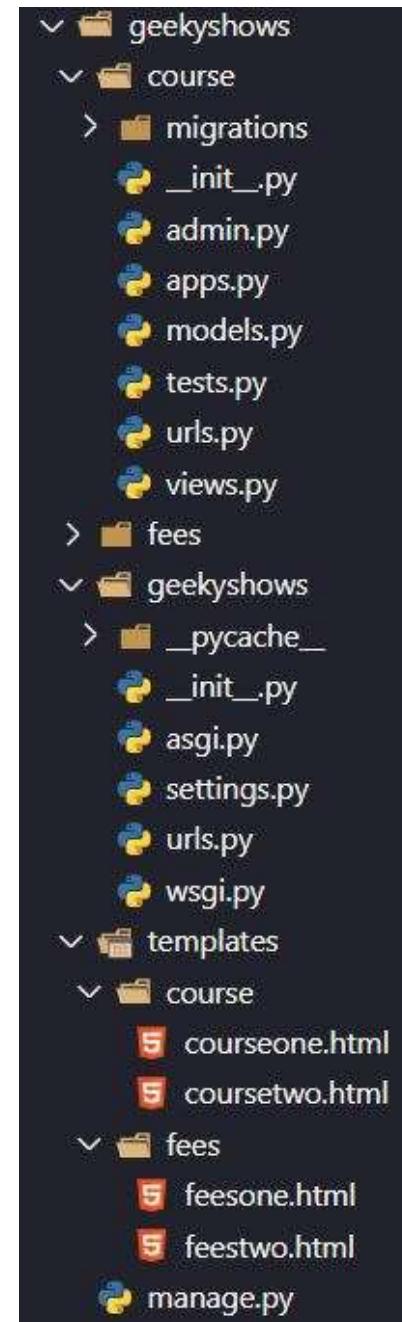
*templates/course*

*courseone.html*

```
<html>
  <head>
    <style> .....
  </head>
  <body>
    <h1>Hello Django</h1>
  </body>
  <script>.....
</html>
```

**views.py**

```
from django.shortcuts import render
def learn_django(request):
    return render(request, 'course/courseone.html')
```



# Geeky Steps

- Create Django Project: *django-admin startproject geekyshows*
- Create Django Application1: *python manage.py startapp course*
- Create Django Application2: *python manage.py startapp fees*
- Add/Install Applications to Django Project (course and fees to geekyshows) using `settings.py` `INSTALLED_APPS`
- Create **templates** folder inside Root Project Directory
- Add **templates** directory in `settings.py`

```
TEMPLATES_DIR = os.path.join(BASE_DIR, 'templates')
```

```
TEMPLATES_DIR = BASE_DIR / 'templates'
```

```
TEMPLATES = [ {
```

```
    'DIRS': [TEMPLATES_DIR], }
```

Old Django Version 3.0

Django Version 3.1

- Create Separate Directory for each application, inside **templates** directory
- Create template files inside `templates/temp_application` folder/directory
- Write View Function inside `views.py` file
- Define url for view function of application using `urls.py` file
- Write Template files code



# 17-Dynamic Template Files using DTL

**templates/course**

**courseone.html**

```
<html>
  <head>
    <style> ......... </style>
  </head>
  <body>
    <h1>Welcome to GeekyShows</h1>
    <h1>Hello {{ cname }}</h1>
  </body>
  <script>.....</script>
</html>
```

Variable

# Dynamic Template Files

## views.py

```
from django.shortcuts import render
def function_name(request):
    Dynamic Data, if else, any python code logic
    return render(request, template_name, context=dict_name, content_type=MIME_type, status=None, using=None)
```

## views.py

```
from django.shortcuts import render
def learn_django(request):
    coursename = {'cname': 'Django'}
    return render(request, 'course/courseone.html', context=coursename)
    return render(request, 'course/courseone.html', coursename)
    return render(request, 'course/courseone.html', {'cname': 'Django'})
```

# Dynamic Template Files

## views.py

```
from django.shortcuts import render

def learn_django(request):
    cname = 'Django'
    duration = '4 Months'
    seats = 10
    django_details = {'nm':cname, 'du':duration, 'st':seats}
    return render(request, 'course/courseone.html', django_details)
```

## *templates/course*

### courseone.html

```
<html>
<body>
    <h2> Course Name:{{nm}} Duration:{{du}} and Total Seats: {{st}}</h2>
</body>
</html>
```



# **18-Requirements**

- Install Python
- Install Django
- How to Create Django Project, Apps and do settings
- How to Create Templates and do settings

# Django Template Language

Django's template language is designed to strike a balance between power and ease. It's designed to feel comfortable to those used to working with HTML.

## courseone.html

```
<html>
  <body>
    <h2> Course Name:{{nm}} Duration:{{du}} and Total Seats: {{st}}</h2>
  </body>
</html>
```

Jinja2 - Jinja is a modern and designer-friendly templating language for Python, modelled after Django's templates. It is fast, widely used and secure with the optional sandboxed template execution environment.

python pip install jinja2

'BACKEND': 'django.template.backends.jinja2.Jinja2',

# Variables

Variables look like this: {{ variable }} When the template engine encounters a variable, it evaluates that variable and replaces it with the result.

## Rules:-

Variable names consist of any combination of alphanumeric characters and the underscore.

Variable name should not start with underscore.

Variable name can not have spaces or punctuation characters.

Syntax:- {{variable}}

Example:- {{nm}}, {{name1}}, {{first\_name}}

# Dynamic Template Files

## views.py

```
from django.shortcuts import render

def learn_django(request):
    cname = 'Django'
    duration = '4 Months'
    seats = 10
    django_details = {'nm':cname, 'du':duration, 'st':seats}
    return render(request, 'course/courseone.html', django_details)
```

## *templates/course*

### courseone.html

```
<html>
<body>
    <h2> Course Name:{{nm}} Duration:{{du}} and Total Seats: {{st}}</h2>
</body>
</html>
```



# Filters

When we need to modify variable before displaying we can use filters. Pipe ‘|’ is used to apply filter.

Syntax:- { {variable | filter} }

Example:- { {name|upper} }

Some of filters take arguments.

Syntax:- { {variable | filter : argument} }

Example:- { {article|truncateword:40} }

Filter can be chained.

Syntax:- { {variable | filter | filter} }

Example:- { {article|upper} }

Example:- { {article|upper|truncateword:40} }

# Filters

capfirst – It capitalizes the first character of the value. If the first character is not a letter, this filter has no effect.

Example:- {{value|capfirst}}

default - If value evaluates to False, uses the given default. Otherwise, uses the value.

Example:- {{ value|default:"nothing" }}

If value is "" (the empty string), the output will be nothing.

length - It returns the length of the value. This works for both strings and lists. The filter returns 0 for an undefined variable.

Example:- {{ value|length }}

lower - It converts a string into all lowercase.

Example:- {{ value|lower }}

# Filters

upper - It converts a string into all uppercase.

Example:- {{ value|upper }}

slice - It returns a slice of the list. Uses the same syntax as Python's list slicing.

Example:- {{ some\_list|slice:"2" }}

truncatechars - It truncates a string if it is longer than the specified number of characters. Truncated strings will end with a translatable ellipsis character ("...").

Argument: Number of characters to truncate to

Example:- {{ value|truncatechars:7 }}

truncatewords - It truncates a string after a certain number of words. Newlines within the string will be removed.

Argument: Number of words to truncate after

Example:- {{ value|truncatewords:2 }}

# Filters

date – It formats a date according to the given format.

Example:- { {value|date:“D d M Y”} }

time – It formats a time according to the given format.

Example:- { {value|time:“H:i”} }

# Day

Format Character	Description	Example
d	Day of the month, 2 digits with leading zeros	01 to 31
j	Day of the month without leading zeros	1 to 31
D	Day of the week, textual, 3 letters	Fri
l	Day of the week, textual, long	Friday
S	English ordinal suffix for day of the month, 2 characters	st, nd, rd or th
w	Day of the week, digits without leading zeros	0 (Sunday) to 6 (Saturday)
z	Day of the year	1 to 366

# Week

Format Character	Description	Example
W	ISO-8601 week number of year, with weeks starting on Monday	1, 53

# Month

Format Character	Description	Example
m	Month, 2 digits with leading zeros	01 to 12
n	Month without leading zeros	1 to 12
M	Month, textual, 3 letters	Jan
b	Month, textual, 3 letters, lowercase	jan
E	Month, locale specific alternative representation usually used for long date representation	'listopada' (for Polish locale, as opposed to 'Listopad')
F	Month, textual, long	January
N	Month abbreviation in Associated Press style. Proprietary extension	Jan., Feb., March, May
t	Number of days in the given month	28 to 31

# Year

Format Character	Description	Example
y	Year, 2 digits	99
Y	Year, 4 digits	1999
L	Boolean for whether it's a leap year	True or False
o	ISO-8601 week-numbering year, corresponding to the ISO-8601 week number (W) which uses leap weeks. See Y for the more common year format	1999

# Time

Format Character	Description	Example
g	Hour, 12-hour format without leading zeros	1 to 12
G	Hour, 24-hour format without leading zeros	0 to 23
h	Hour, 12-hour format	01 to 12
H	Hour, 24-hour format	00 to 23
i	Minutes	00 to 59
s	Seconds, 2 digits with leading zeros	00 to 59
u	Microseconds	000000 to 999999
a	'a.m.' or 'p.m.' (Note that this is slightly different than PHP's output, because this includes periods to match Associated Press style.)	a.m.
A	'AM' or 'PM'	AM
f	Time, in 12-hour hours and minutes, with minutes left off if they're zero. Proprietary extension	1, 1:30
p	Time, in 12-hour hours, minutes and 'a.m.'/'p.m.', with minutes left off if they're zero and the special-case strings 'midnight' and 'noon' if appropriate. Proprietary extension	1 a.m., 1:30 p.m., midnight, noon, 12:30 p.m.

# Timezone

Format Character	Description	Example
e	Timezone name. Could be in any format, or might return an empty string, depending on the datetime	", 'GMT', '-500', 'US/Eastern', etc
I	Daylight Savings Time, whether it's in effect or not	1 or 0
O	Difference to Greenwich time in hours	+0200
T	Time zone of this machine	EST, MDT
Z	Time zone offset in seconds. The offset for timezones west of UTC is always negative, and for those east of UTC is always positive	-43200 to 43200

# Date/Time

Format Character	Description	Example
c	ISO 8601 format. (Note: unlike others formatters, such as “Z”, “O” or “r”, the “c” formatter will not add timezone offset if value is a naive datetime)	2008-01-02T10:30:00.000123+02:00, or 2008-01-02T10:30:00.000123 if the datetime is naive
r	RFC 5322 formatted date	'Thu, 21 Dec 2000 16:01:07 +0200'
U	Seconds since the Unix Epoch (January 1 1970 00:00:00 UTC)	

# Predefined Formats

Format	Description	Example
DATE_FORMAT	Default: 'N j, Y'	Feb. 9, 2020
DATETIME_FORMAT	Default: 'N j, Y, P'	Feb. 9, 2020, 10 p.m.
SHORT_DATE_FORMAT	Default: 'm/d/Y'	12/31/2020
SHORT_DATETIME_FORMAT	Default: 'm/d/Y P'	12/31/2020 4 p.m.
TIME_FORMAT	Default: "H:i"	"01:24"

Example:- {{ value|date:"SHORT\_DATE\_FORMAT" }}

Example:- {{ value|time:"TIME\_FORMAT" }}

# Filters

## floatformat

When used without an argument, rounds a floating-point number to one decimal place but only if there's a decimal part to be displayed.

Value	Template	Output
56.24321	{ { value floatformat } }	56.2
56.00000	{ { value floatformat } }	56
56.37000	{ { value floatformat } }	56.4

If used with a numeric integer argument, floatformat rounds a number to that many decimal places.

Value	Template	Output
56.24321	{ { value floatformat:3 } }	56.243
56.00000	{ { value floatformat:3 } }	56.000
56.37000	{ { value floatformat:3 } }	56.370

# Filters

## floatformat

Particularly useful is passing 0 (zero) as the argument which will round the float to the nearest integer.

Value	Template	Output
56.24321	<code>{{value floatformat:"0"}}</code>	56
56.00000	<code>{{value floatformat:"0"}}</code>	56
56.37000	<code>{{value floatformat:"0"}}</code>	56

If the argument passed to floatformat is negative, it will round a number to that many decimal places but only if there's a decimal part to be displayed.

Value	Template	Output
56.24321	<code>{{value floatformat:"-3"}}</code>	56.243
56.00000	<code>{{value floatformat:"-3"}}</code>	56
56.37000	<code>{{value floatformat:"-3"}}</code>	56.370

# **if Tag**

{% if %} tag - The {% if %} tag evaluates a variable, and if that variable is “true” (i.e. exists, is not empty, and is not a false boolean value).

Syntax:-

```
{% if variable %}
```

.....

```
{% endif %}
```

Example:-

```
{% if nm %}
```

```
</h1>Hello {{nm}}</h1>
```

```
{% endif %}
```

```
{% if nm or st %}  
</h1>Seat Available</h1>  
{% endif %}
```

```
{% if not st%}  
</h1>Seat Not Available</h1>  
{% endif %}
```

```
{% if nm and st %}
```

```
</h1> For Course{{nm}} {{st}} Seat Available</h1>
```

```
{% endif %}
```

# if Tag with condition

Syntax:-

```
{% if condition %}  
.....  
{% endif %}
```

Example:-

```
{% if nm == 'Django' %}  
    </h1>Hello {{nm}}</h1>  
{% endif %}
```

```
{% if nm == 'Django' or st == 5 %}  
    </h1>{{nm}} Seat Available</h1>  
{% endif %}
```

```
{% if not st == 5 %}  
    </h1>Seat Not Available</h1>  
{% endif %}
```

```
{% if nm == 'Django' and st==5 %}  
    </h1>{{nm}} Seat Available</h1>  
{% endif %}
```

if tags may also use the operators ==, !=, <, >, <=, >=, **in**, **not in**, **is**, and **is not**

# if Tag with filter

Syntax:-

```
{% if variable|filter % }
```

.....

```
{% endif % }
```

Example:-

```
{% if nm|length >= 6 % }
```

```
</h1>Hello {{nm}}</h1>
```

```
{% endif % }
```

# **if else Tag**

Syntax:-

```
{% if variable %}
```

.....

```
{% else %}
```

.....

```
{% endif %}
```

Example:-

```
{% if nm %}
```

```
</h1>Hello {{nm}}</h1>
```

```
{% else %}
```

```
<h1>No Course Available</h1>
```

```
{% endif %}
```

# if else Tag with Condition

Syntax:-

```
{% if condition %}
```

.....

```
{% else %}
```

.....

```
{% endif %}
```

Example:-

```
{% if nm =='Django'%}
```

```
</h1>Hello {{nm}}</h1>
```

```
{% else %}
```

```
<h1>No Course Available</h1>
```

```
{% endif %}
```

# **if elif Tag**

Syntax:-

```
{% if variable %}  
.....  
{% elif variable %}  
.....  
{% else %}  
.....  
{% endif %}
```

Example:-

```
{% if nm %}  
    </h1>Hello {{nm}}</h1>  
{% elif st %}  
    </h1>Seats {{st}}</h1>  
{% else %}  
    </h1>No Course Available</h1>  
{% endif %}
```

# if elif Tag with condition

Syntax:-

```
{% if condition %}
```

.....

```
{% elif condition %}
```

.....

```
{% else %}
```

.....

```
{% endif %}
```

Example:-

```
{% if nm=='Django' %}
```

```
</h1>Hello {{ nm }}</h1>
```

```
{% elif st==5 %}
```

```
</h1>Seats {{ st }}</h1>
```

```
{% else %}
```

```
<h1>No Course Available</h1>
```

```
{% endif %}
```

# **Dot Lookup**

Technically, when the template system encounters a dot, it tries the following lookups, in this order:

- Dictionary lookup
- Attribute or method lookup
- Numeric index lookup

# **for loop Tag**

Syntax:-

```
{% for variable in variables %}  
  {{ variable }}  
{% endfor %}
```

Example:-

```
<ul>  
  {% for stu in student %}  
    <li>{{ stu }}</li>  
  {% endfor %}  
</ul>
```

Syntax:-

```
{% for variable in variables %}  
  {{ variable }}  
{% empty %}  
Empty  
{% endfor %}
```

Syntax:-

```
{% for key, value in data.items %}  
  {{ key }}: {{ value }}  
{% endfor %}
```

# Predefined forloop Variable

Variable	Description
forloop.counter	The current iteration of the loop (1-indexed)
forloop.counter0	The current iteration of the loop (0-indexed)
forloop.revcounter	The number of iterations from the end of the loop (1-indexed)
forloop.revcounter0	The number of iterations from the end of the loop (0-indexed)
forloop.first	True if this is the first time through the loop
forloop.last	True if this is the last time through the loop
forloop.parentloop	For nested loops, this is the loop surrounding the current one

Example:-

```
{% for stu in student %}  
  {{forloop.counter}} {{stu}}  
{% endfor %}
```

# 19-Templates inside Project

## settings.py

Old Version Django 3.0:

```
TEMPLATES_DIR = os.path.join(BASE_DIR, 'templates')
```

```
TEMPLATES_DIR = BASE_DIR / 'templates'
```

```
INSTALLED_APPS = [
    'course',
    'fees'
]
```

```
TEMPLATES = [
    {
        'DIRS': [TEMPLATES_DIR],
        'APP_DIRS': True,
    }
]
```

Django Version 3.1

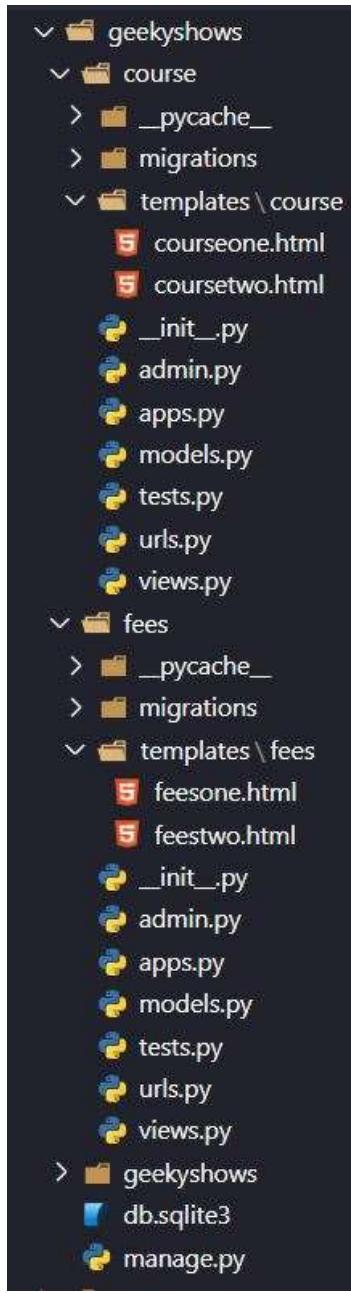


# Templates inside Application

## settings.py

```
INSTALLED_APPS = [  
    'course',  
    'fees'  
]
```

```
TEMPLATES = [  
    {  
        'DIRS': [ ],  
        'APP_DIRS': True,  
    }  
]
```

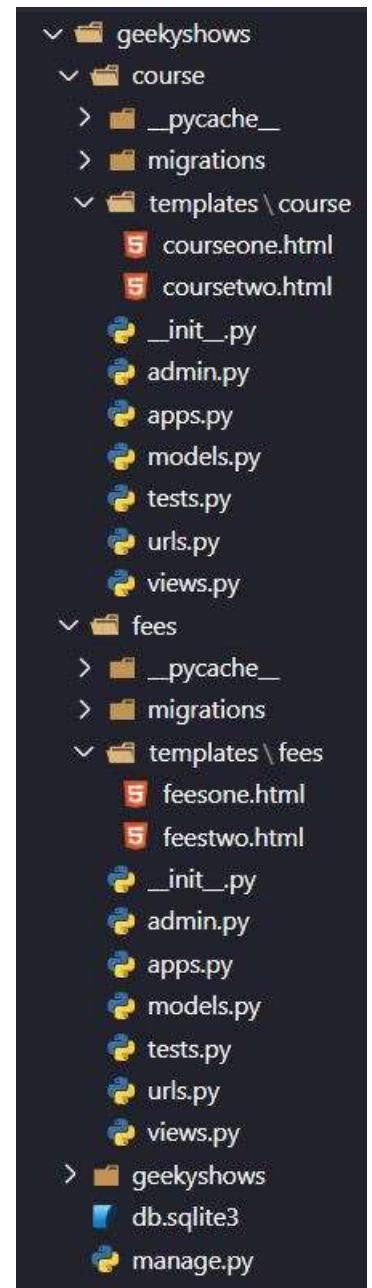


# Geeky Steps

- Create Django Project: *django-admin startproject geekyshows*
- Create Django Application1: *python manage.py startapp course*
- Create Django Application2: *python manage.py startapp fees*
- Add/Install Applications to Django Project (course and fees to geekyshows) using settings.py INSTALLED\_APPS
- Create **templates** folder inside each application
- Check ‘APP\_DIRS’ : True in settings.py

```
TEMPLATES = [ {'APP_DIRS' : True } ]
```

- Create **folder** inside **app/templates** directory for template files
- Create template files inside **app/templates/folder**
- Write View Function inside views.py file
- Define url for view function of application using urls.py file
- Write Template files code



## settings.py

Old Version Django 3.0:

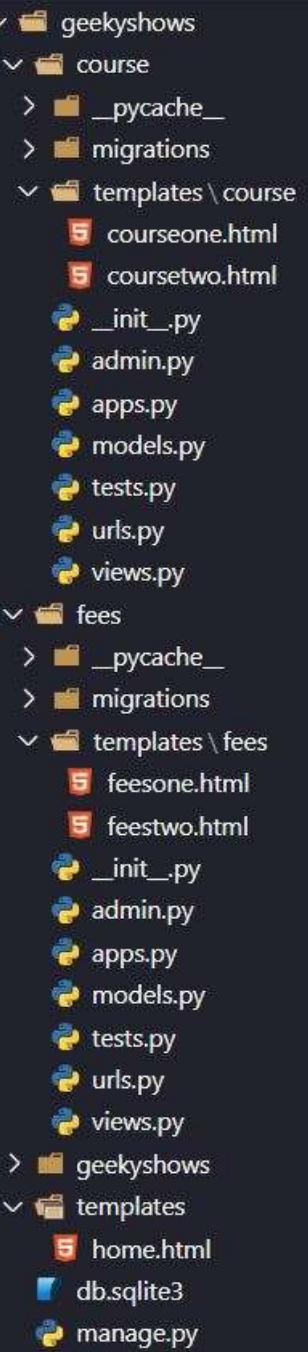
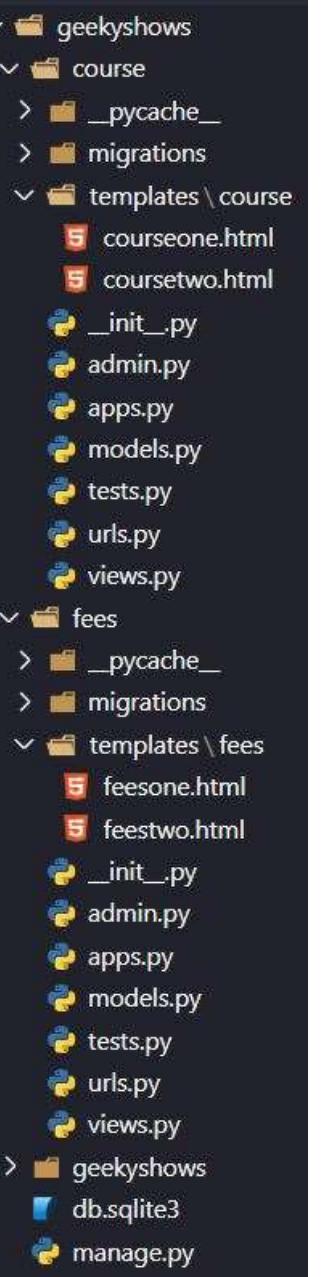
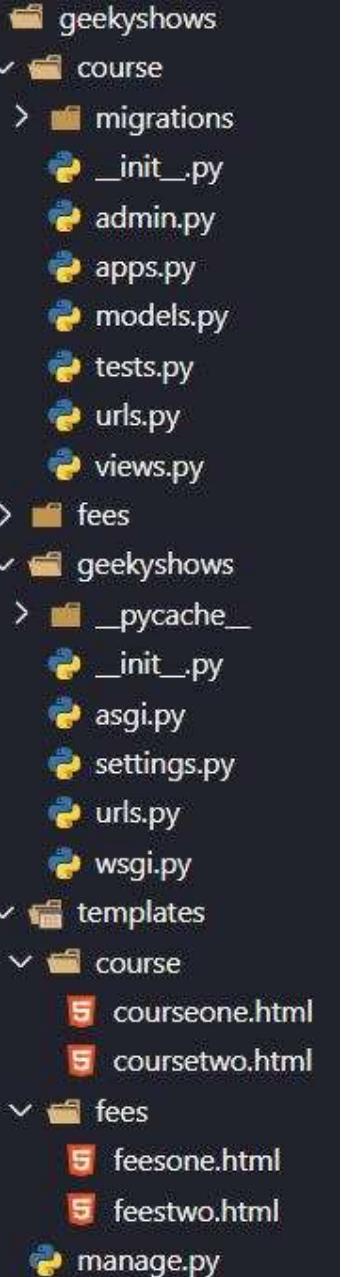
```
TEMPLATES_DIR = os.path.join(BASE_DIR, 'templates')
```

```
TEMPLATES_DIR = BASE_DIR / 'templates'
```

```
INSTALLED_APPS = [  
    'course',  
    'fees'  
]
```

```
TEMPLATES = [  
    {  
        'DIRS': [TEMPLATES_DIR],  
        'APP_DIRS': True,  
    }  
]
```

Django Version 3.1



# Geeky Steps

- Create Django Project: *django-admin startproject geekyshows*
- Create Django Application1: *python manage.py startapp course*
- Create Django Application2: *python manage.py startapp fees*
- Add/Install Applications to Django Project (course and fees to geekyshows) using settings.py INSTALLED\_APPS
- Create **templates** folder inside each application and inside Root Project Folder
- Check ‘APP\_DIRS’ : True in settings.py

```
TEMPLATES = [ {'APP_DIRS': True } ]
```

- Add **templates** directory which is inside Root Project Folder, in settings.py

```
TEMPLATES_DIR = os.path.join(BASE_DIR, 'templates')
```

```
TEMPLATES_DIR = BASE_DIR / 'templates'
```

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

Old Django Version 3.0

Django Version 3.1

- Create **folder** inside **app/templates** directory for template files
- Create template files inside **app/templates/folder**
- Create template files inside templates folder which is inside Root Project Folder
- Write View Function inside views.py file
- Define url for view function of application using urls.py file
- Write Template files code



# 20-Static Files

CSS files, Javascript Files, image files, video files etc are considered as static files in Django.

Django provides `django.contrib.staticfiles` to help you manage them.

`django.contrib.staticfiles` collects static files from each of your applications (and any other places you specify) into a single location that can easily be served in production.

# How to Create Static Folder and Files

We create **static** folder inside Root Project Folder then inside **static** folder we create required folders which will contain all static files respectively like css folder will contain all css files, image folder will contain all images and so on.

geekyshows

**static**

    css

      style.css

      custom.css

    images

      love.jpg

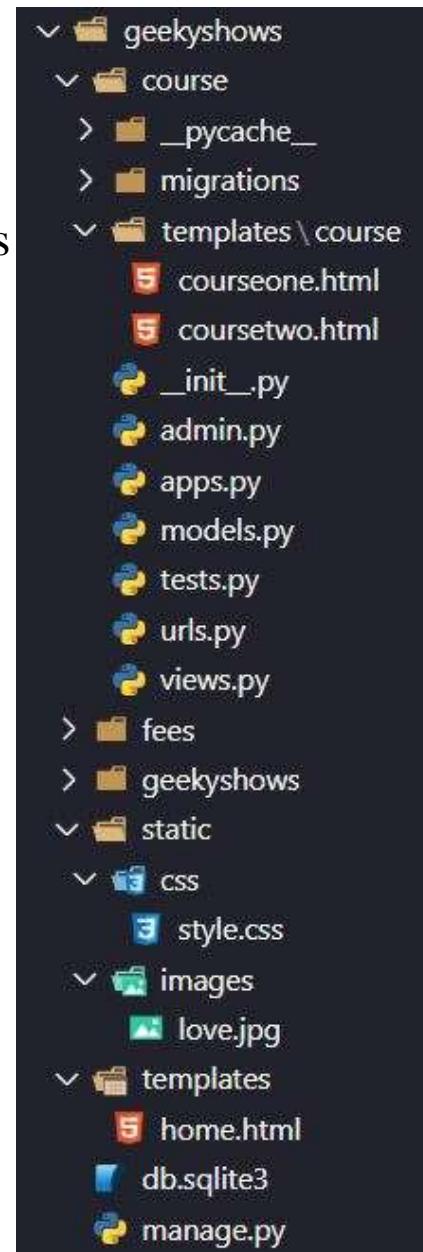
      pic1.jpg

    templates

  geekyshows

  manage.py

course



# Add Static in settings.py

## settings.py

Old Django Version 3.0

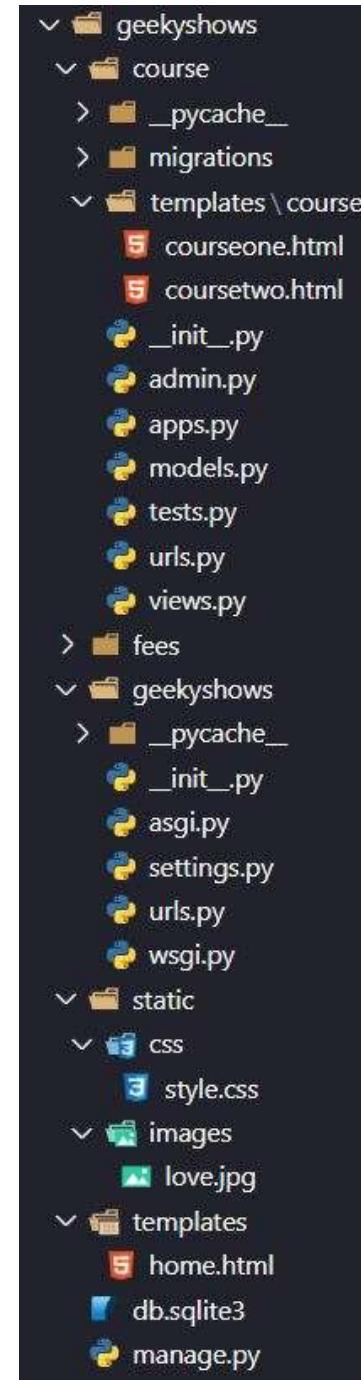
```
TEMPLATES_DIR = os.path.join(BASE_DIR, 'templates')
STATIC_DIR = os.path.join(BASE_DIR, 'static')
```

TEMPLATES\_DIR = BASE\_DIR / 'templates'  
**STATIC\_DIR = BASE\_DIR / 'static'**      }    Django Version 3.1

```
INSTALLED_APPS = [
    'course'
]
```

```
TEMPLATES = [
    {
        'DIRS': [TEMPLATES_DIR]
    }
]
```

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



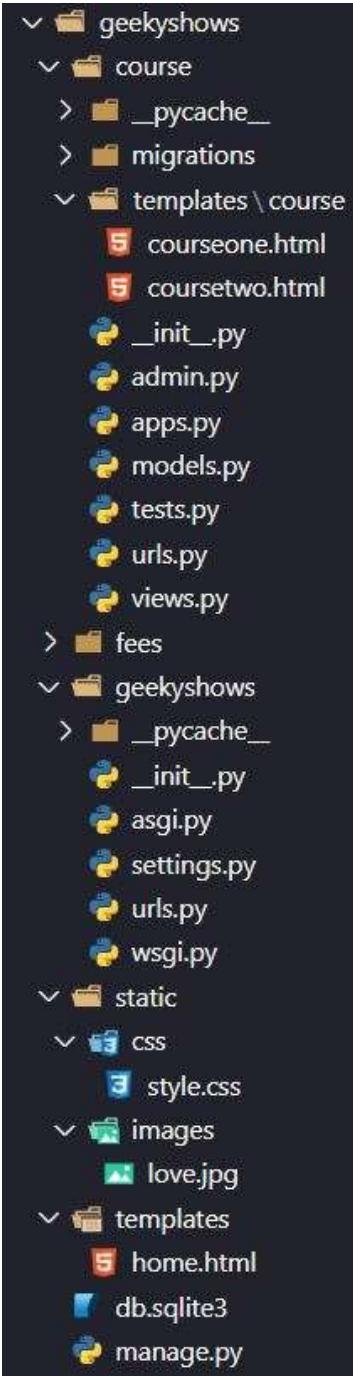
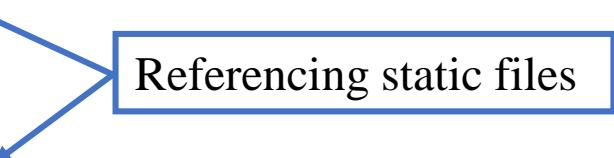
# Use Static Files in Template Files

- First Load Static Files
- Reference Static Files

**templates/course**

**courseone.html**

```
<!DOCTYPE html>
{% load static %}          // Loading Static Files
<html>
    <link href='{% static "css/style.css" %}'>
    <body>
        <h1>Fees {{fe}} </h1>
        <img src='{% static "images/love.jpg" %}'>
    </body>
</html>
```



# load Template Tag

{% load module\_name % } – It loads a custom template tag set.

Example:- {% load emotags % }

Example:- {% load geek.mytags % }

Example:- {% load emotags geek.mytags % }

Template would load all the tags and filters registered in emotags and mytags located in package geek.

You can also selectively load individual filters or tags from a library or module, using the from argument.

Example - {% load cry lol from emotags % }

The template tags/filters named cry and lol will be loaded from emotags.

# static Template Tag

{% static filename %} – This tag is used to link to static files that are saved in STATIC\_ROOT. If the django.contrib.staticfiles app is installed, the tag will serve files using url() method of the storage specified by STATICFILES\_STORAGE.

Syntax:-

{% load static %}

{% static filename %}

{% static path/filename %}

{% static path/filename as variable %}

Example:-

```
<link rel="stylesheet" href="{% static 'style.css' %}" >  
<link rel="stylesheet" href="{% static 'css/style.css' %}" >  
  
{% static "images/love.jpg" as mylove %}  

```

There's also a second form you can use to avoid extra processing if you need the value multiple times.

```
{% load static %}  
{% get_static_prefix as STATIC_PREFIX %}  
  

```

**STATIC\_URL** – This is the URL to use when referring to static file located in STATIC\_ROOT. It must end in a slash if set to a non-empty value.

Example:- “/static/”

Example:- “http://static.example.com/”

**STATIC\_ROOT** – This is absolute path to the directory where *collectstatic* will collect static files for deployment. It is by default None.

Example:- “/var/www/example.com/static/”

Example:- os.path.join(BASE\_DIR, ‘static/’)

**STATICFILES\_DIRS** - This setting defines the additional locations the *staticfiles* app will traverse if the *FileSystemFinder* finder is enabled, e.g. if you use the *collectstatic* or *findstatic* management command or use the static file serving view. It is by default an empty list.

STATICFILES\_DIRS = [

    "/home/special.geek.com/geek/static",

    "/home/geek.com/geek/static",

    "/opt/webfiles/common",

]

**STATICFILES\_STORAGE** - The file storage engine to use when collecting static files with the `collectstatic` management command.

Default: '*django.contrib.staticfiles.storage.StaticFilesStorage*'

A ready-to-use instance of the storage backend defined in this setting can be found at  
*django.contrib.staticfiles.storage.staticfiles\_storage*

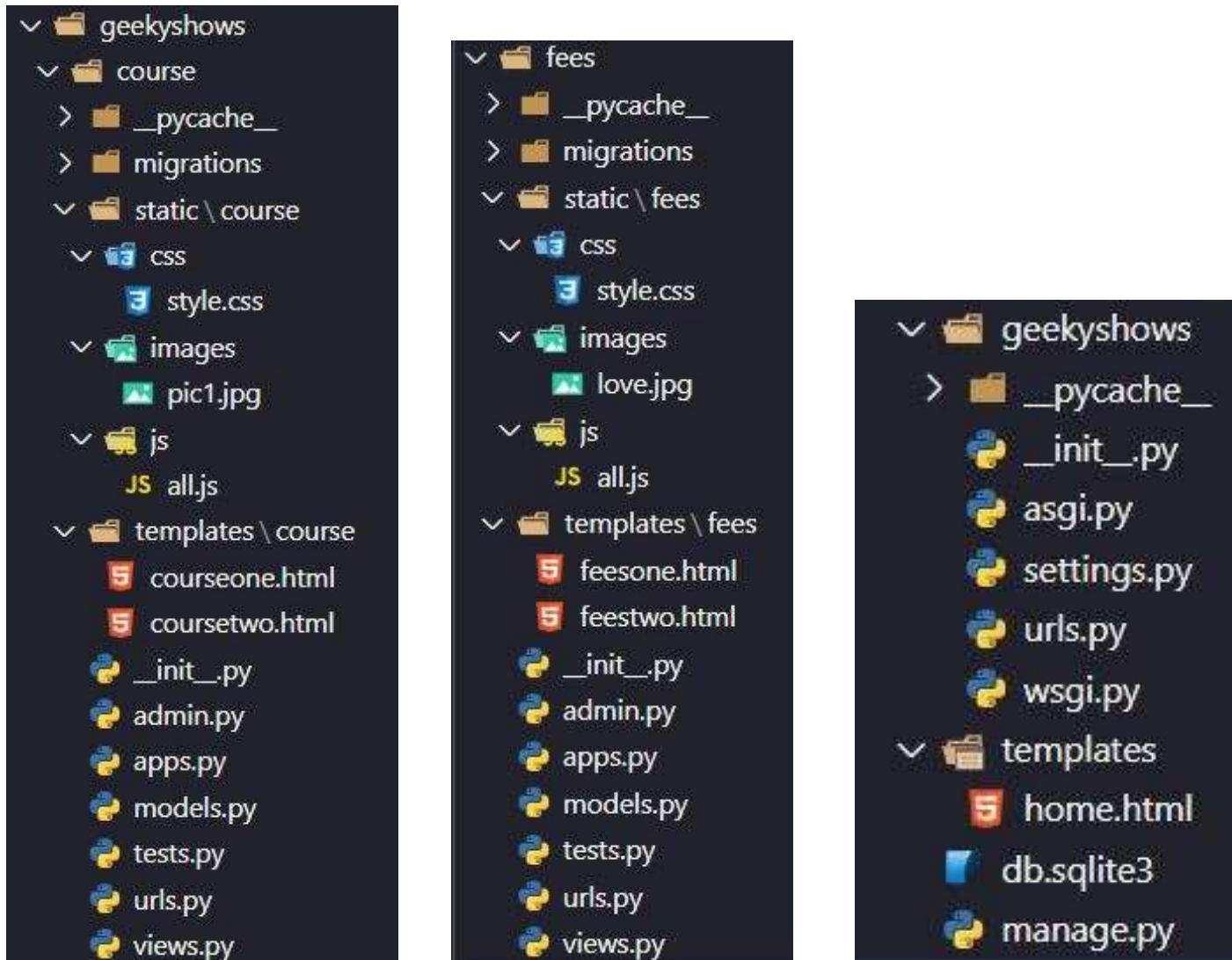
# Geeky Steps

- Create Django Project: *django-admin startproject geekyshows*
- Create Django Application1: *python manage.py startapp course*
- Create Django Application2: *python manage.py startapp fees*
- Add/Install Applications to Django Project (course and fees to geekyshows) using settings.py INSTALLED\_APPS
- Create **templates** folder inside each application and inside Root Project Folder
- Check ‘APP\_DIRS’ : True in settings.py
- Add **templates** directory which is inside Root Project Folder, in settings.py
- Create **folder** inside **app/templates** directory for template files
- Create template files inside **app/templates/folder**
- Create template files inside templates folder which is inside Root Project Folder
- Create **static** folder inside Root Project Folder
- Create css, js, images, video etc folder inside **static** folder
- Create static files inside css, js, images, video etc folder.
- Write View Function inside views.py file
- Define url for view function of application using urls.py file
- Write Template files code
- Write Static file code



# How to Create Static Folder and Files inside Application

We create **static** folder inside Application Folder then inside **static** folder we create required folders which will contain all static files respectively like css folder will contain all css files, image folder will contain all images and so on.



# Add Static in settings.py

## settings.py

## Old Django Version 3.0

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

```
TEMPLATES_DIR = BASE_DIR / 'templates'  
STATIC_DIR = BASE_DIR / 'static'
```

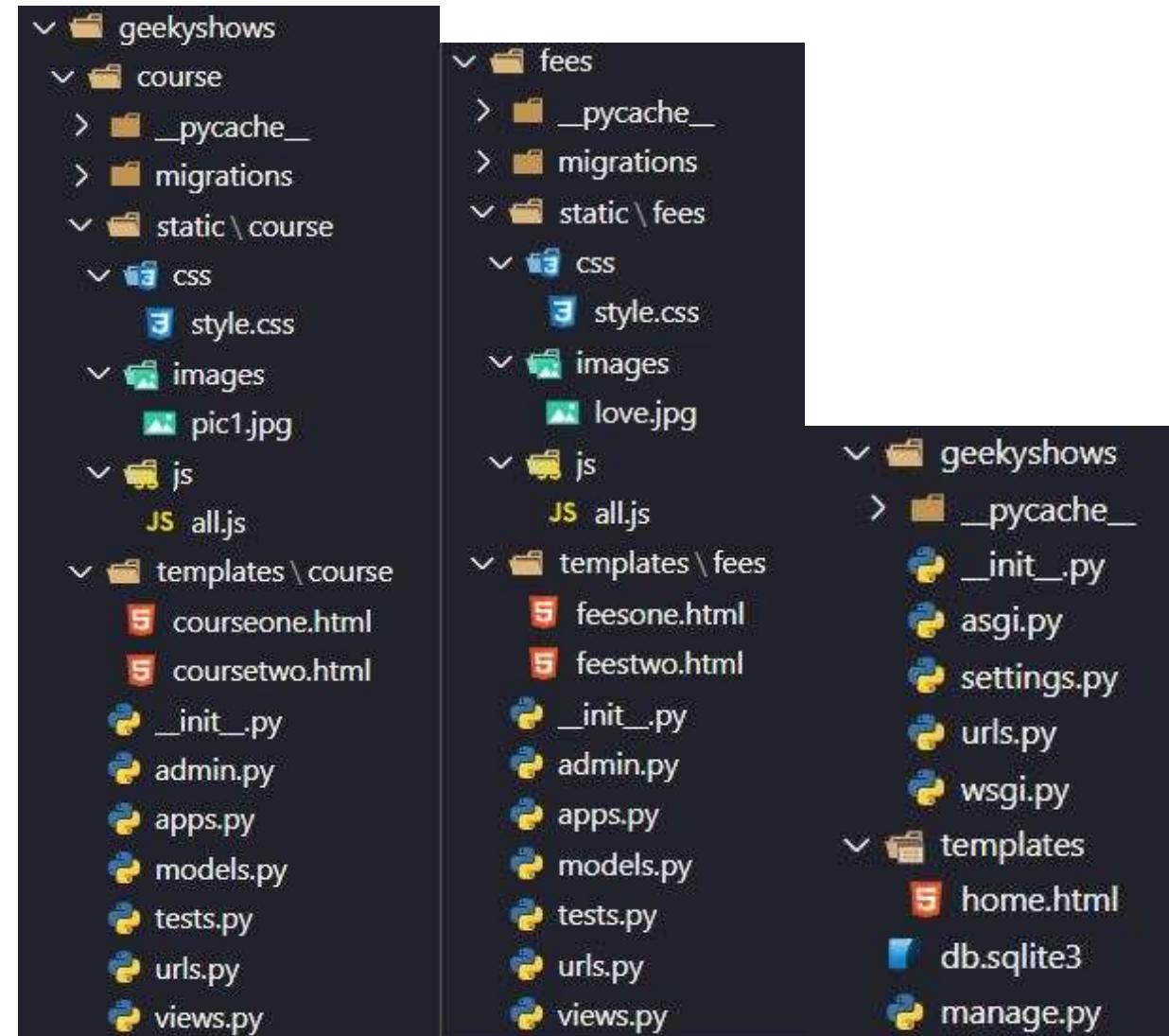
```
INSTALLED_APPS = [  
    'course'  
]
```

```
TEMPLATES = [  
    {  
        'DIRS': [TEMPLATES_DIR]  
    }  
]
```

STATIC URL = '/static/'

**STATICFILES\_DIRS** = [ **STATIC\_DIR** ]

Django Version 3.1



# Use Static Files in Template Files

- First Load Static Files
- Reference Static Files

templates/course

courseone.html

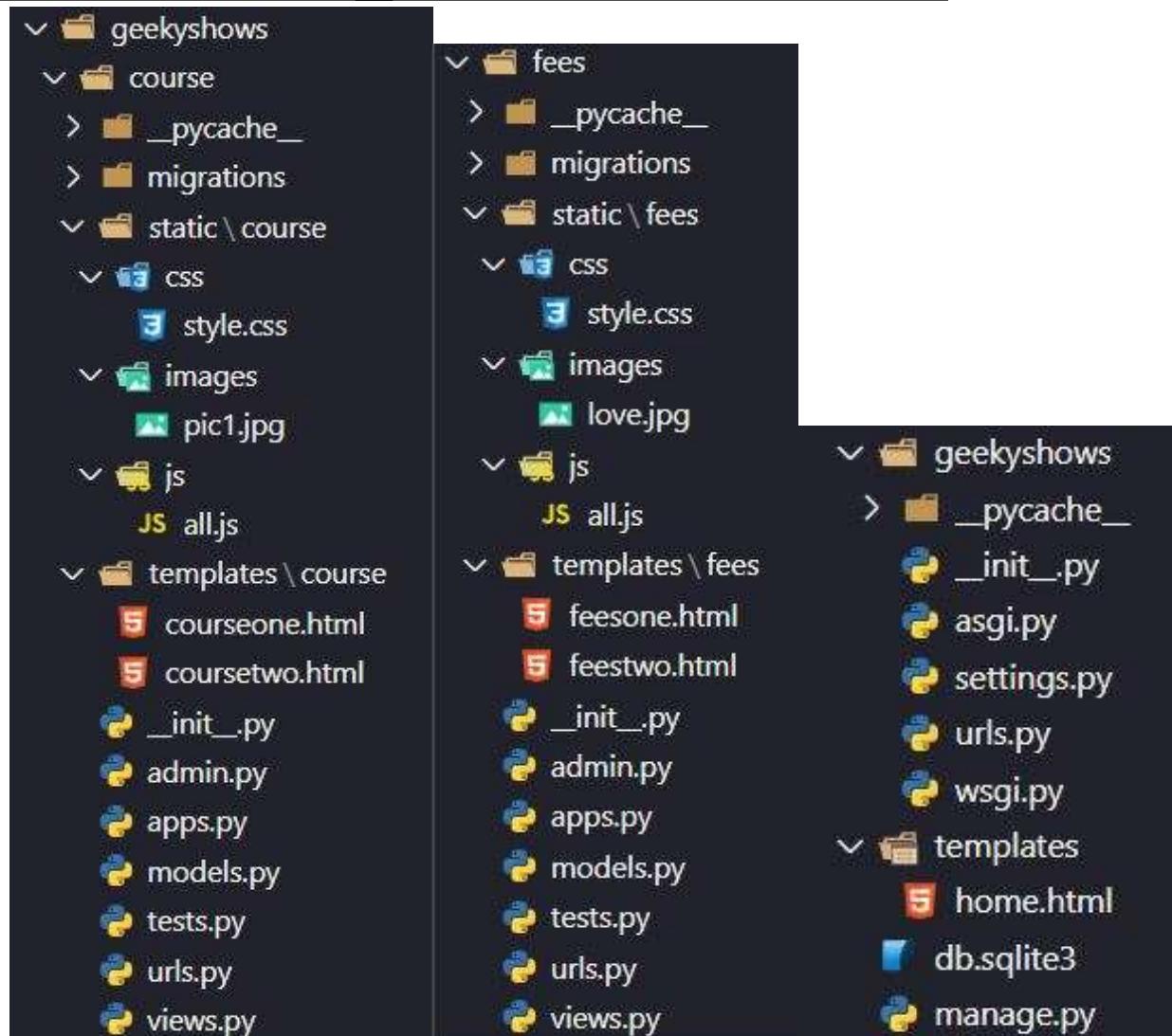
```
<!DOCTYPE html>  
{%load static %}          // Loading Static Files
```

<html>

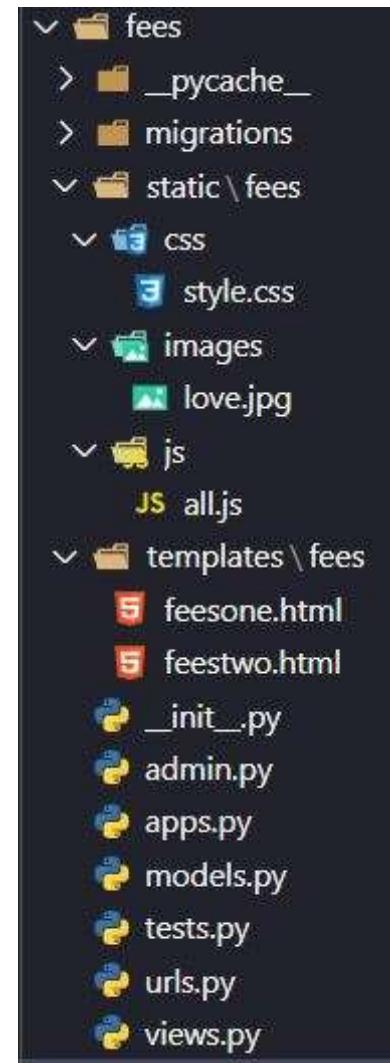
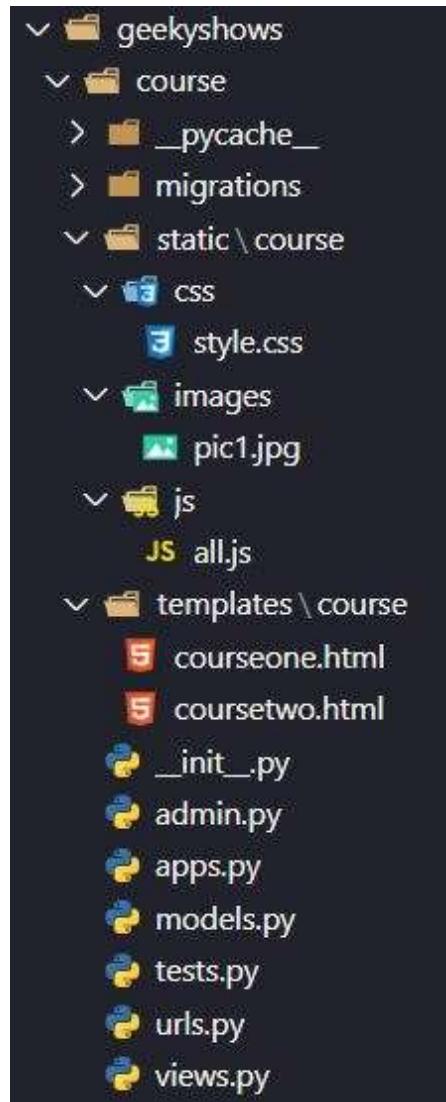
```
    <link href='{%static "course/css/style.css"}'>  
    <body>  
        <h1>Fees {{fe}} </h1>  
        <img src='{%static "course/images/pic1.jpg"}'>  
    </body>
```

</html>

Referencing  
static files



# How to Create Static Folder and Files inside Project & Application



# Add Static in settings.py

## settings.py

Old Django Version 3.0

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

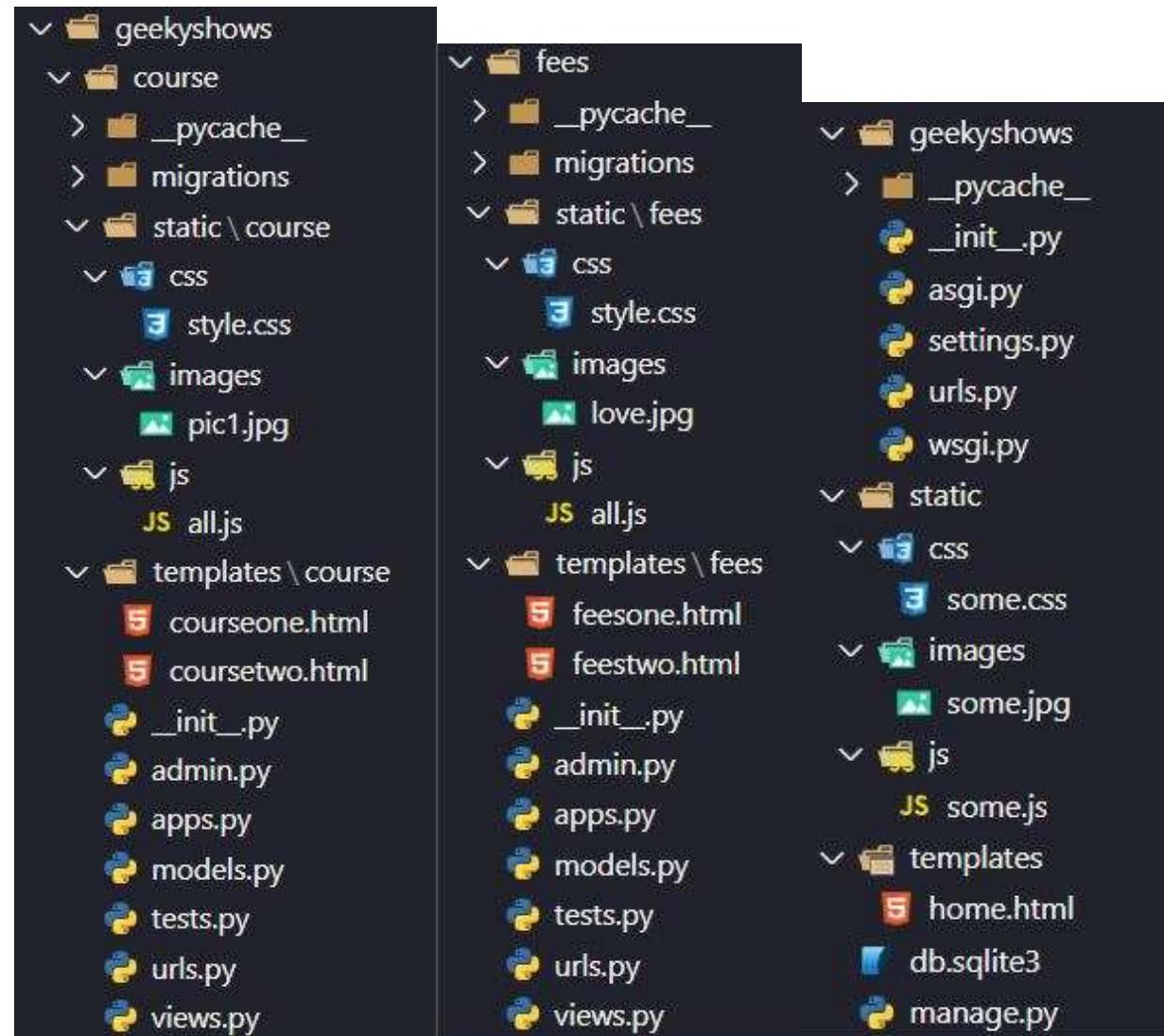
TEMPLATES\_DIR = BASE\_DIR / 'templates'  
**STATIC\_DIR = BASE\_DIR / 'static'**

```
INSTALLED_APPS = [
    'course'
]
```

```
TEMPLATES = [
    {
        'DIRS': [TEMPLATES_DIR]
    }
]
```

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

Django Version 3.1



# Use Static Files in Template Files

- First Load Static Files
- Reference Static Files

```
<!DOCTYPE html>
{%load static%}           // Loading Static Files
<html>
    <link href='{%static "css/some.css"}'>
    <body>
        <h1>Fees {{fe}} </h1>
        <img src='{%static "course/images/pic1.jpg"}'>
    </body>
</html>
```



# static Template Tag

get\_media\_prefix - Similar to the get\_static\_prefix, get\_media\_prefix populates a template variable with the media prefix MEDIA\_URL, e.g.:

```
{% load static %}  
<body data-media-url="{% get_media_prefix %}">
```

By storing the value in a data attribute, we ensure it's escaped appropriately if we want to use it in a JavaScript context.

# 21-Template Inheritance/ Template Extending

Template inheritance allows you to build a base “skeleton” template that contains all the common elements of your site and defines blocks that child templates can override.

The *extends* tag is used to inherit template.

*extends* tag tells the template engine that this template “extends” another template.

When the template system evaluates this template, first it locates the parent let’s assume, “base.html”.

At that point, the template engine will notice the block tags in base.html and replace those blocks with the contents of the child template.

You can use as many levels of inheritance as needed.

# **extends Tag**

{% extends %} – The *extends* tag is used to inherit template. It tells the template engine that this template “extends” another template. It has no end tag.

Syntax:-

```
{% extends 'parent_template_name' %}  
{% extends variable %}
```

Example:-

```
{% extends "./base1.html" %}  
{% extends "../base2.html" %}  
{% extends "./my/base3.html" %}  
{% extends somthing %}
```

# **block Tag**

{% block %} – The *block* tag is used to for overriding specific parts of a template.

Syntax:-

{% block blockname %}....{% endblock %}

{% block blockname %}....{% endblock blockname %}

Example:-

{% block title %} ..... {% endblock %}

{% block content %} ..... {% endblock content %}

# Rules

- If We use { % extends % } in a template, it must be the first template tag in that template. Template inheritance won't work, otherwise.
- More { % block % } tags in our base templates are better.
- Child templates don't have to define all parent blocks, so we can fill in reasonable defaults in a number of blocks, then only define the ones we need later.
- We Can't define multiple block tags with the same name in the same template.
- If We need to get the content of the block from the parent template, the { { block.super } } variable will do the trick.

# Creating Base/Parent Template and Child Template

We write common codes in base template and create blocks for code which may vary page to page. Later this template will be inherited by child templates and child template will override created blocks.

## **home.html**

```
<html>
<head>
<title>Home</title>
</head>
<body>
<h1>Hello I am Home Page</h1>
</body>
</html>
```

## **base.html**

```
<html>
<head>
<title>% block title %</title>
</head>
<body>
    % block content %
</body>
</html>
```

## **about.html**

```
<html>
<head>
<title>About</title>
</head>
<body>
<h1>Hello I am About Page</h1>
</body>
</html>
```

## **home.html**

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

```
{% block title %}
```

```
    Home
```

```
{% endblock %}
```

```
{% block content %}
```

```
    <h1>Hello I am Home Page </h1>
```

```
{% endblock content %}
```

## **about.html**

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

```
{% block title %}
```

```
    About
```

```
{% endblock %}
```

```
{% block content %}
```

```
    <h1>Hello I am About Page </h1>
```

```
{% endblock content %}
```

# Creating Base/Parent Template and Child Template

## **base.html**

```
<html>
<head>
<title>{% block title %} Other {% endblock %}</title>
</head>
<body>
{% block content %} {% endblock content %}
</body>
</html>
```

## **home.html**

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

{% block title %}
    Home
{% endblock %}

{% block content %}
    <h1>Hello I am Home Page</h1>
{% endblock content %}
```

## **about.html**

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

{% block content %}
    <h1>Hello I am About Page</h1>
{% endblock content %}
```

## **home.html**

```
<html>
<head>
<title>Home</title>
</head>
<body>
<h1>Hello I am Home Page</h1>
</body>
</html>
```

## **about.html**

```
<html>
<head>
<title>Other</title>
</head>
<body>
<h1>Hello I am About Page</h1>
</body>
</html>
```

# Creating Base/Parent Template and Child Template

## **base.html**

```
<html>
<head>
<title>{% block title %} Other {% endblock %}</title>
</head>
<body>
{% block content %} {% endblock content %}
</body>
</html>
```

## **home.html**

```
{% extends 'base.html' %}
{% block title %} {{block.super}}
Home{% endblock %}

{% block content %}
<h1>Hello I am Home Page </h1>
{% endblock content %}
```

## **about.html**

```
{% extends 'base.html' %}
{% block content %}
</h1>Hello I am About Page</h1>
{% endblock content %}
```

## **home.html**

```
<html>
<head>
<title>Other Home</title>
</head>
<body>
<h1>Hello I am Home Page</h1>
</body>
</html>
```

## **about.html**

```
<html>
<head>
<title>Other</title>
</head>
<body>
<h1>Hello I am About Page</h1>
</body>
</html>
```

# Creating Base/Parent Template and Child Template

## **base.html**

```
<html>
<head>
<title>{% block title %} Other {% endblock %}</title>
</head>
<body>
    {% block content %} {% endblock content %}
    {% block somecontent1 %} {% endblock somecontent1 %}
    {% block somecontent2 %} {% endblock somecontent2 %}
    {% block somecontent3 %} {% endblock somecontent3 %}
</body>
</html>
```

## **home.html**

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

    {% block title %} Home
    {% endblock %}

    {% block content %}
        Hello I am Home Page
    {% endblock content %}
```

## **about.html**

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

    {% block somecontent2 %}
        Hello I am About Page
    {% endblock somecontent2 %}
```

## **22-url Tag**

{% url %} - It returns an absolute path reference (a URL without the domain name) matching a given view and optional parameters. Any special characters in the resulting path will be encoded using `iri_to_uri()`.

Syntax:-

```
{% url 'urlname' %}
```

```
{% url 'urlname' as var %}
```

```
{% url 'urlname' arg1=value1 arg2=value2 %}
```

```
{% url 'urlname' value1 value2 %}
```

# path()

path(route, view, kwargs=None, name=None) - It returns an element for inclusion in urlpatterns.

Where,

- The route argument should be a string or gettext\_lazy() that contains a URL pattern. The string may contain angle brackets e.g. <username> to capture part of the URL and send it as a keyword argument to the view. The angle brackets may include a converter specification like the int part of <int:id> which limits the characters matched and may also change the type of the variable passed to the view. For example, <int:id> matches a string of decimal digits and converts the value to an int.
- The view argument is a view function or the result of as\_view() for class-based views. It can also be an django.urls.include().
- The kwargs argument allows you to pass additional arguments to the view function or method. It should be a dictionary.
- name is used to perform URL reversing.

# path ()

## urls.py

```
urlpatterns = [  
    path(route, view, kwargs=None, name=None)  
]
```

## urls.py

```
urlpatterns = [  
    path('learndj/', views.learn_Django, {'check': 'OK'}, name='learn_django'),  
]
```

```
urlpatterns = [
    path('about/', views.about),
]

<a href="/about">About</a>
<a href="{{ab}}>About</a>

<a href="{% url 'aboutus' %}">About</a>

{% url 'aboutus' as abc %}
<a href="{{abc}}>About</a>

urlpatterns = [ path('about/', views.about, name='aboutus'),]
```

```
def about(request):
    return render(request, 'core/about.html', {'ab':'/about'})
```

## 23-Include Tag

{% include %} Tag – It loads a template and renders it with the current context. This is a way of “including” other templates within a template. Each include is a completely independent rendering process.

The template name can either be a variable or a hard-coded (quoted) string, in either single or double quotes.

Syntax:-

```
{% include temp_var_name %}  
{% include "template_name.html" %}  
{% include "folder/template_name.html" %}
```

Example:-

```
{% include topvar %}  
{% include "topcourse.html" %}  
{% include "fees/extrafees.html" %}
```

# Include Tag

We can pass additional context explicitly to the template using *with* keyword.

```
{% include "topcourse.html" with p="PHP" d="Django" %}
```

If we want to render the context only with the variables provided (or even no variables at all), use the *only* option. No other variables are available to the included template.

```
{% include "topcourse.html" with p="PHP" only %}
```

# 24-Deploy Application on Pythonanywhere

1. Check module/requirement list

pip freeze

2. Create a Requirements Files

pip freeze > req.txt

3. Define STATIC\_ROOT value

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

4. Compress Project

5. Upload Compressed Project on Server

# Deploy Application on Pythonanywhere

6. After Uploading Done Click Consoles Tab then open Bash

7. Create Virtual Env

```
mkvirtualenv envname --python=/usr/bin/python3.8
```

8. Make sure you are inside env then Uncompress Project

```
unzip projectfilename
```

9. Go to Web Tab then Add a new Web App (Choose Manual Configuration and Python Version)

10. Find virtualenv and write your env name

# Deploy Application on Pythonanywhere

11. Go to Console Tab and Open Base

12. Enter into your env

workon envname

13. dir to see list of dir an files

14. Change Directory to projectfolder

cd projectfolder

15. Install Requirements

pip install -r req.txt

# Deploy Application on Pythonanywhere

16. Go to Web Tab

open wsgi.py file

17. Remove Code except DJANGO Section then edit it and save

18. Go to Files and open inner project folder then open settings.py file then edit and save

DEBUG = False

ALLOWED\_HOSTS = ['\*']

19. Go to consoles Tab open Bash and Active env

# Deploy Application on Pythonanywhere

20. Change Directory to Project Folder then run collectstatic command

```
python manage.py collectstatic
```

21. GO to Web Tab and look for Static Files Section there write URL and Directory

URL = /static/

Directory = /home/magicextra/resumeproject/static

22. Reload

# 25-Object Relational Mapper (ORM)

Object-Relational Mapper (ORM), which enables application to interact with database such as SQLite, MySQL, PostgreSQL, Oracle.

ORMs automatically create a database schema from defined classes or models. It generate SQL from Python code for a particular database which means developer do not need to write SQL Code.

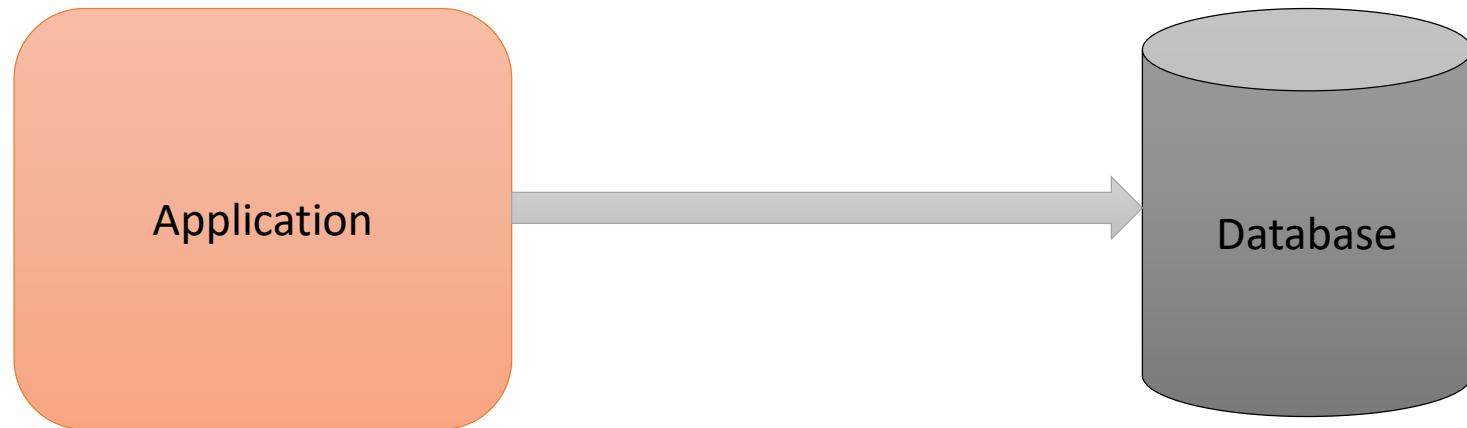
ORM maps objects attributes to respective table fields.

It is easier to change the database if we use ORMs hence project becomes more portable.

Django's ORM is just a way to create SQL to query and manipulate your database and get results in a pythonic fashion.

ORMs use connectors to connect databases with a web application.

# Object Relation Mapper (ORM)



# Object Relation Mapper (ORM)

```
class Student(models.Model):  
    stuid=models.IntegerField()  
    stuname=models.CharField(max_length=70)  
    stuemail=models.EmailField(max_length=70)  
    stupass=models.CharField(max_length=70)
```



```
CREATE TABLE "enroll_student" (  
    "id" integer NOT NULL PRIMARY KEY  
AUTOINCREMENT,  
    "stuid" integer NOT NULL,  
    "stuname" varchar(70) NOT NULL,  
    "stuemail" varchar(70) NOT NULL,  
    "stupass" varchar(70) NOT NULL  
);
```



id	stuid	stuname	stuemail	stupass

# QuerySet

A QuerySet can be defined as a list containing all those objects we have created using the Django model.

QuerySets allow you to read the data from the database, filter it and order it.

# **26-Model**

A model is the single, definitive source of information about your data.  
It contains the essential fields and behaviors of the data you're storing.  
Generally, each model maps to a single database table.

# Model Class

Model class is a class which will represent a table in database.

Each model is a Python class that subclasses `django.db.models.Model`

Each attribute of the model represents a database field.

With all of this, Django gives you an automatically-generated database-access API  
Django provides built-in database by default that is sqlite database.

We can use other database like MySQL, Oracle SQL etc.

# Create Our Own Model Class

models.py file which is inside application folder, is required to create our own model class.

Our own model class will inherit Python's Model Class.

Syntax:-

```
class ClassName(models.Model):  
    field_name=models.FieldType(arg, options)
```

Example:-

models.py

```
class Student(models.Model):  
    stuid=models.IntegerField()  
    stuname=models.CharField(max_length=70)  
    stuemail=models.EmailField(max_length=70)  
    stupass=models.CharField(max_length=70)
```

Length is required in CharField Type

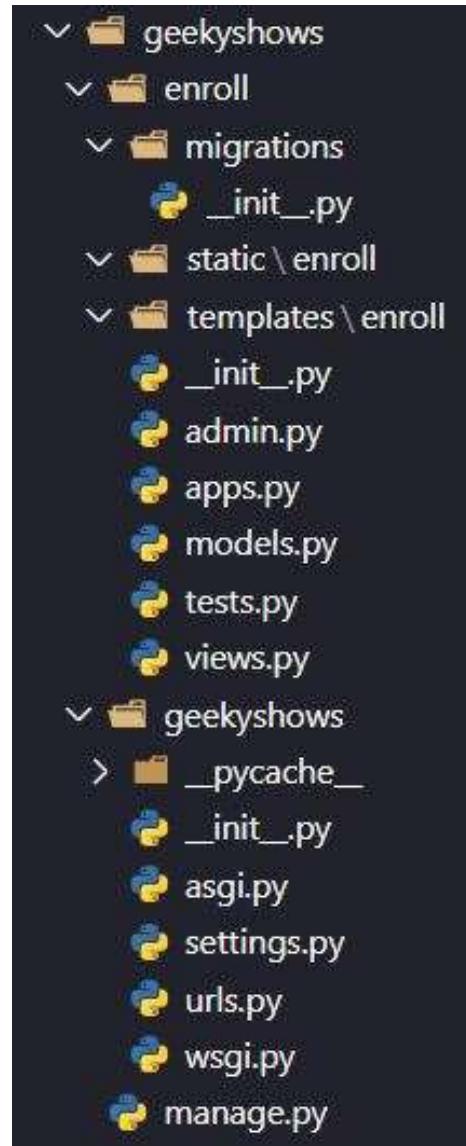
- This class will create a table with columns and their data types
- Table Name will be ApplicationName\_ClassName, in this case it will be enroll\_student
- Field name will become table's Column Name, in this case it will be stuid, stuname, stuemail, stupass with their data type.
- As we have not mentioned primary key in any of these columns so it will automatically create a new column named 'id' Data Type Integer with primary key and auto increment.

# Create Our Own Model Class

## models.py

```
class Student(models.Model):
    stuid=models.IntegerField()
    stuname=models.CharField(max_length=70)
    stuemail=models.EmailField(max_length=70)
    stupass=models.CharField(max_length=70)
```

```
CREATE TABLE "enroll_student" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "stuid" integer NOT NULL,
    "stuname" varchar(70) NOT NULL,
    "stuemail" varchar(70) NOT NULL,
    "stupass" varchar(70) NOT NULL
);
```



# Rules

- Field Name instantiated as a class attribute and represents a particular table's Column name.
- Field Type is also known as Data Type.
- A field name cannot be a Python reserved word, because that would result in a Python syntax error.
- A field name cannot contain more than one underscore in a row, due to the way Django's query lookup syntax works.
- A field name cannot end with an underscore.

# How to use Models

Once you have defined your models, you need to tell Django you're going to use those models.

- Open settings.py file
- Write app name which contains models.py file in `INSTALLED_APPS = [ ]`
- Open Terminal
- Run `python manage.py makemigrations`
- Run `python manage.py migrate`

# Migrations

Migrations are Django’s way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema.

- **makemigrations** – This is responsible for creating new migrations based on the changes you have made to your models.
- **migrate** – This is responsible for applying and unapplying migrations.
- **sqlmigrate** – This displays the SQL statements for a migration.
- **showmigrations** – This lists a project’s migrations and their status.

# makemigrations and migrate

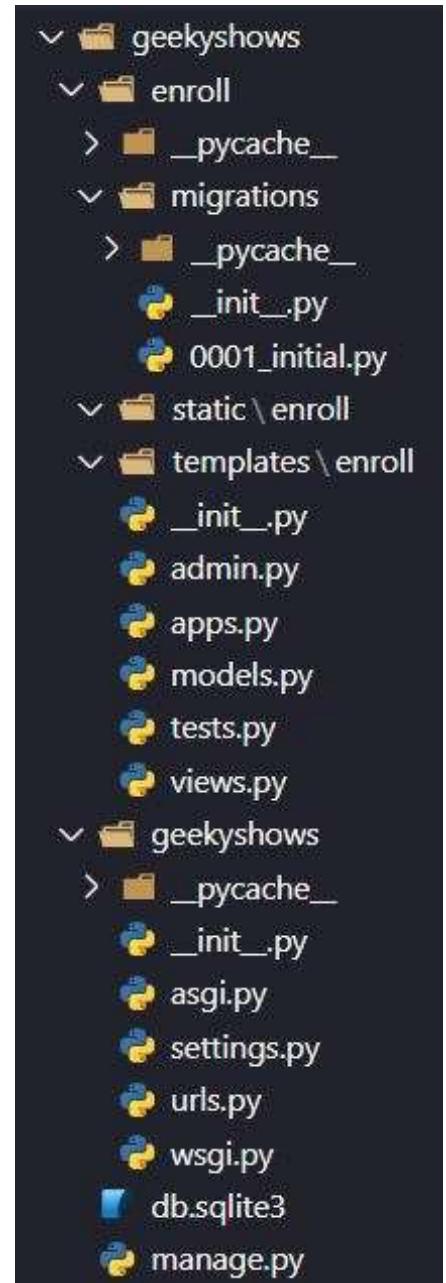
- **makemigrations** is used convert model class into sql statements. This will also create a file which will contain sql statements. This file is located in Application's migrations folder.

Syntax:- python manage.py makemigrations

- **migrate** is used to execute sql statements generated by makemigrations. This command will execute All Application's (including built-in applications) SQL Statements if available. After execution of sql statements table will be created.

Syntax:- python manage.py migrate

Note – If you make any change in your own model class you are required to run makemigrations and migrate command only then you will get those changes in your application.



# Display SQL Statement

We can retrieve SQL Statement by using below command:-

Syntax:-

```
python manage.py sqlmigrate application_name dbfile_name
```

Example:-

```
python manage.py sqlmigrate enroll 0001
```

Note – File name can be found inside Application's migrations folder.

## **models.py**

```
class Student(models.Model):
    stuid=models.IntegerField()
    stuname=models.CharField(max_length=70)
    stuemail=models.EmailField(max_length=70)
    stupass=models.CharField(max_length=70)
```

## **python manage.py makemigrations**

```
CREATE TABLE "enroll_student" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "stuid" integer NOT NULL,
    "stuname" varchar(70) NOT NULL,
    "stuemail" varchar(70) NOT NULL,
    "stupass" varchar(70) NOT NULL
);
```

## enroll/migrations/0001\_initial.py

```
from django.db import migrations, models
class Migration(migrations.Migration):
    initial = True
    dependencies = []
    operations = [
        migrations.CreateModel(
            name='Student',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('stuid', models.IntegerField()),
                ('stuname', models.CharField(max_length=70)),
                ('stuemail', models.EmailField(max_length=70)),
                ('stupass', models.CharField(max_length=70)),
            ],
        ),
    ]
```

The “initial migrations” for an app are the migrations that create the first version of that app’s tables. Usually an app will have one initial migration, but in some cases of complex model interdependencies it may have two or more.

Initial migrations are marked with an `initial = True` class attribute on the migration class. If an `initial` class attribute isn’t found, a migration will be considered “initial” if it is the first migration in the app.

A list of migrations this one depends on

A list of Operation classes that define what this migration does

## **enroll/migrations/0002\_student\_comment.py**

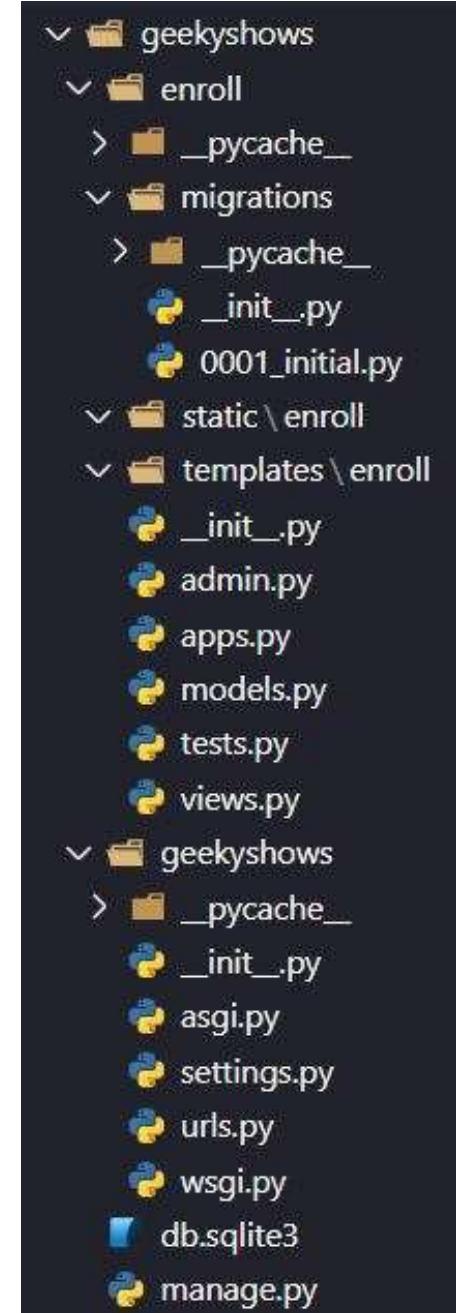
```
from django.db import migrations, models
```

```
class Migration(migrations.Migration):
```

```
    dependencies = [('enroll', '0001_initial'), ] ← A list of migrations this one depends on  
    operations = [← A list of Operation classes that define what this migration does  
        migrations.AddField(  
            model_name='student',  
            name='comment',  
            field=models.CharField(default='not available', max_length=40),  
        ), ]
```

# Geeky Steps

- Create Django Project: *django-admin startproject geekyshows*
- Create Django Application: *python manage.py startapp enroll*
- Add/Install Applications to Django Project using settings.py INSTALLED\_APPS
- Create **templates** folder inside application
- Create **static** folder inside application
- Open models.py file which is inside application
- Write Model Class
- Run *python manage.py makemigrations* Command
- A migration file will be generate automatically inside **migrations** folder
- Run *python manage.py migrate* Command
- Database table will be created automatically
- Write View Function inside views.py file
- Define url for view function of application using urls.py file
- Write Template files code
- Write Static file code



# **Built-in Field Options**

null – It can contain either True or False. If True, Django will store empty values as NULL in the database. Default is False.

Avoid using null on string-based fields such as CharField and TextField. If a string-based field has null=True, that means it has two possible values for “no data”: NULL, and the empty string.

Example:- null=True/False

blank – It can contain either True or False. If True, the field is allowed to be blank. This is different than null. null is purely database-related, whereas blank is validation-related. If a field has blank=True, form validation will allow entry of an empty value. If a field has blank=False, the field will be required. Default is False.

Example:- blank=True/False

# **Built-in Field Options**

default - The default value for the field. This can be a value or a callable object. If callable it will be called every time a new object is created.

Example:- default='Not Available'

verbose\_name - A human-readable name for the field. If the verbose name isn't given, Django will automatically create it using the field's attribute name, converting underscores to spaces.

Example:- verbose\_name='Student Name'

db\_column - The name of the database column to use for this field. If this isn't given, Django will use the field's name.

Example- db\_column = 'Student Name'

# Built-in Field Options

`primary_key` – If True, this field is the primary key for the model.

If you don't specify `primary_key=True` for any field in your model, Django will automatically add an `AutoField` to hold the primary key, so you don't need to set `primary_key=True` on any of your fields unless you want to override the default primary-key behavior.

The primary key field is read-only. If you change the value of the primary key on an existing object and then save it, a new object will be created alongside the old one.

`primary_key=True` implies `null=False` and `unique=True`. Only one primary key is allowed on an object.

Example:- `primary_key=True`

# **Built-in Field Options**

unique - If True, this field must be unique throughout the table.

This is enforced at the database level and by model validation.

If you try to save a model with a duplicate value in a unique field, a django.db.IntegrityError will be raised by the model's save() method.

This option is valid on all field types except ManyToManyField and OneToOneField.

When unique is True, you don't need to specify db\_index, because unique implies the creation of an index.

Example:- unique=True

# Built-in Field Types

**IntegerField** - An integer. Values from -2147483648 to 2147483647 are safe in all databases supported by Django. It uses `MinValueValidator` and `MaxValueValidator` to validate the input based on the values that the default database supports. The default form widget for this field is a `NumberInput` when `localize` is `False` or `TextInput` otherwise.

Example:- `roll = models.IntegerField()`

**BigIntegerField** - A 64-bit integer, much like an `IntegerField` except that it is guaranteed to fit numbers from -9223372036854775808 to 9223372036854775807. The default form widget for this field is a `TextInput`.

Example:- `mobile = models.BigIntegerField()`

# Built-in Field Types

AutoField - An IntegerField that automatically increments according to available IDs. You usually won't need to use this directly; a primary key field will automatically be added to your model if you don't specify.

Example:- stuid = models.AutoField()

FloatField - A floating-point number represented in Python by a float instance. The default form widget for this field is a NumberInput when localize is False or TextInput otherwise.

Example:- fees = models.FloatField()

# Built-in Field Types

CharField - A string field, for small- to large-sized strings. For large amounts of text, use TextField. The default form widget for this field is a TextInput. CharField has one extra required argument: max\_length

The maximum length (in characters) of the field.

Example:- first\_name = models.CharField(max\_length=30)

TextField - A large text field. The default form widget for this field is a Textarea. If you specify a max\_length attribute, it will be reflected in the Textarea widget of the auto-generated form field. However it is not enforced at the model or database level. Use a CharField for that.

Example:- description = models.TextField(max\_length=150)

# Built-in Field Types

BooleanField - A true/false field. The default form widget for this field is CheckboxInput, or NullBooleanSelect if null=True.

The default value of BooleanField is None when Field.default isn't defined.

Example:- status = models.BooleanField()

EmailField - A CharField that checks that the value is a valid email address using EmailValidator.

Example:- email = models.EmailField()

# Built-in Field Types

URLField - A CharField for a URL, validated by URLValidator. The default form widget for this field is a TextInput. Like all CharField subclasses, URLField takes the optional max\_length argument. If you don't specify max\_length, a default of 200 is used.

Example:- partnersite = models.URLField()

BinaryField - A field to store raw binary data. It can be assigned bytes, bytearray, or memoryview. By default, BinaryField sets editable to False, in which case it can't be included in a ModelForm. BinaryField has one extra optional argument: max\_length

The maximum length (in characters) of the field.

Example:- profile\_img = models.BinaryField()

# Model Operations

`CreateModel(name, fields, options=None, bases=None, managers=None)` – It creates a new model in the project history and a corresponding table in the database to match it.

Where,

- name is the model name, as would be written in the `models.py` file.
- fields is a list of 2-tuples of (`field_name`, `field_instance`). The field instance should be an unbound field (so just `models.CharField(...)`, rather than a field taken from another model).
- options is an optional dictionary of values from the model's Meta class.
- bases is an optional list of other classes to have this model inherit from; it can contain both class objects as well as strings in the format "appname.ModelName" if you want to depend on another model (so you inherit from the historical version). If it's not supplied, it defaults to inheriting from the standard `models.Model`.
- managers takes a list of 2-tuples of (`manager_name`, `manager_instance`). The first manager in the list will be the default manager for this model during migrations.

# Model Operations

DeleteModel(name) – It deletes the model from the project history and its table from the database.

RenameModel(old\_name, new\_name) – It renames the model from an old name to a new one.

You may have to manually add this if you change the model's name and quite a few of its fields at once to the autodetector, this will look like you deleted a model with the old name and added a new one with a different name, and the migration it creates will lose any data in the old table.

AlterModelTable(name, table) – It changes the model's table name (the db\_table option on the Meta subclass).

AlterUniqueTogether(name, unique\_together)- It changes the model's set of unique constraints (the unique\_together option on the Meta subclass).

# Model Operations

AlterIndexTogether(name, index\_together) – It changes the model’s set of custom indexes (the index\_together option on the Meta subclass).

AlterOrderWithRespectTo(name, order\_with\_respect\_to) – It makes or deletes the \_order column needed for the order\_with\_respect\_to option on the Meta subclass.

AlterModelOptions(name, options) – It stores changes to miscellaneous model options (settings on a model’s Meta) like permissions and verbose\_name. Does not affect the database, but persists these changes for RunPython instances to use. options should be a dictionary mapping option names to values.

AlterModelManagers(name, managers) – It alters the managers that are available during migrations.

# Model Operations

AddField(model\_name, name, field, preserve\_default=True) – It adds a field to a model.

Where

model\_name is the model's name.

name is the field's name.

field is an unbound Field instance (the thing you would put in the field declaration in models.py for example, models.IntegerField(null=True)).

The preserve\_default argument indicates whether the field's default value is permanent and should be baked into the project state (True), or if it is temporary and just for this migration (False) - usually because the migration is adding a non-nullble field to a table and needs a default value to put into existing rows. It does not affect the behavior of setting defaults in the database directly - Django never sets database defaults and always applies them in the Django ORM code.

# Model Operations

`RemoveField(model_name, name)` – It removes a field from a model.

Bear in mind that when reversed, this is actually adding a field to a model. The operation is reversible (apart from any data loss, which of course is irreversible) if the field is nullable or if it has a default value that can be used to populate the recreated column. If the field is not nullable and does not have a default value, the operation is irreversible.

`AlterField(model_name, name, field, preserve_default=True)` – It alters a field's definition, including changes to its type, null, unique, db\_column and other field attributes.

The `preserve_default` argument indicates whether the field's default value is permanent and should be baked into the project state (`True`), or if it is temporary and just for this migration (`False`) - usually because the migration is altering a nullable field to a non-nullable one and needs a default value to put into existing rows. It does not affect the behavior of setting defaults in the database directly - Django never sets database defaults and always applies them in the Django ORM code.

Not all changes are possible on all databases - for example, you cannot change a text-type field like `models.TextField()` into a number-type field like `models.IntegerField()` on most databases.

# Model Operations

`RenameField(model_name, old_name, new_name)` – It changes a field's name (and, unless `db_column` is set, its column name).

`AddIndex(model_name, index)` – It creates an index in the database table for the model with `model_name`. `index` is an instance of the `Index` class.

`RemoveIndex(model_name, name)` – It removes the index named `name` from the model with `model_name`.

`AddConstraint(model_name, constraint)` – It creates a constraint in the database table for the model with `model_name`.

`RemoveConstraint(model_name, name)` – It removes the constraint named `name` from the model with `model_name`.

# **27-Show Table Data to User**

- Writing Code to get data from database in views.py then pass it to template files using render function
- Get Data which is passed by render function of views.py file in template file

# **all ()**

all ( ) – It returns a copy of current QuerySet or QuerySet Subclass.

Syntax:-

ModelClassName.objects.all()

# Writing Code to get DB data in views.py

First import your own model class from models.py

Example:-

views.py

```
from enroll.models import Student  
  
def studentinfo(request):  
    stud = Student.objects.all()  
    return render(request, 'enroll/studetails.html', {'stu':stud})
```

# Get Data from views.py in template file

templates/enroll / studetails.html

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Student Page</h1>
    { % if stu  %}
      <h1>Show Data</h1>
      { % for st in stu %}
        <h5>{ { st.stuid } }</h5>
        <h5>{ { st.stuname } }</h5>
      { % endfor %}
    { % else %}
      <h1>No Data</h1>
    { % endif %}
  </body>
</html>
```

# **28-Admin Application**

It is a built-in application provided by Django.

This application provides admin interface for CRUD operations without writing sql statements.

It reads metadata from your models to provide a quick, model-centric interface where trusted users can manage content on your site.

Admin Application can be accessed using <http://127.0.0.1:8000/admin>

Super User is required to login into Admin Application

# Create Super User

We need super user to login into admin interface of the admin application.

*createsuperuser* command is used to create super user.

Syntax:- `python manage.py createsuperuser`

# How to Register Model

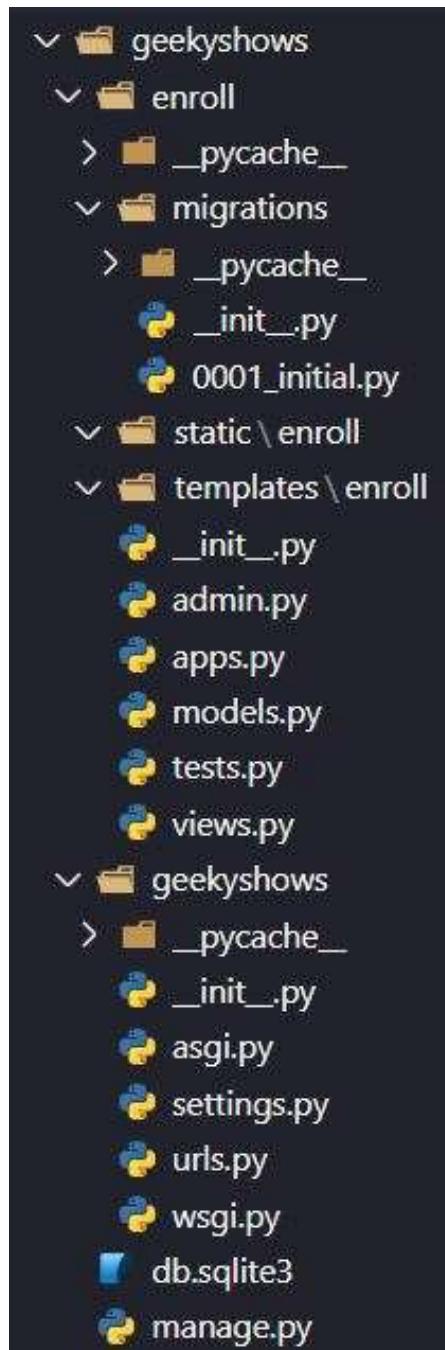
We are registering our table which we has created using model class, to default admin interface.

To Register Follow:-

- Open *admin.py* file which is inside Application Folder
- Import your own Model Class created inside Application's *models.py*
- *admin.site.register(ModelClassName)*

Example:-

- Open *admin.py*
- *from enroll.models import Student*
- *admin.site.register(Student)*



# `__str__()` Method

The `__str__()` method is called whenever you call `str()` on an object. To display an object in the Django admin site and as the value inserted into a template when it displays an object. Thus, you should always return a nice, human-readable representation of the model from the `__str__()` method.

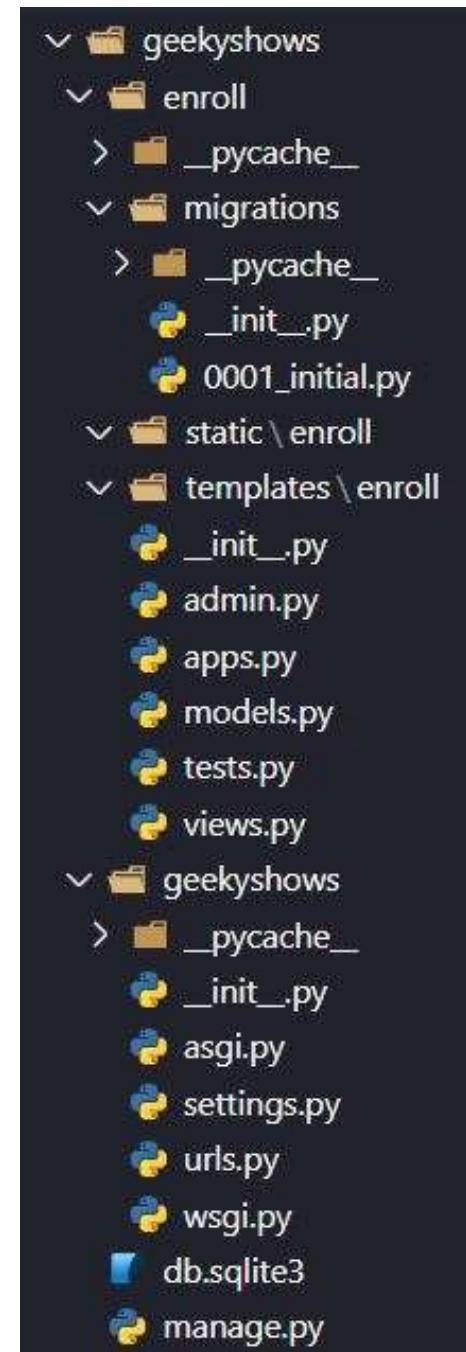
Write this Method in your own model class which is inside `models.py` file.

Syntax:-

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

Example:-

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



# ModelAdmin

The ModelAdmin class is the representation of a model in the admin interface.

To show table's all data in admin interface we have to create an ModelAdmin class in admin.py file of Application folder.

Syntax:-

## Creating Class

```
Class ModelAdminClassName(admin.ModelAdmin):  
    ModelAdmin Options  
        list_display=(‘fieldname1’, ‘fieldname2’, .....)
```

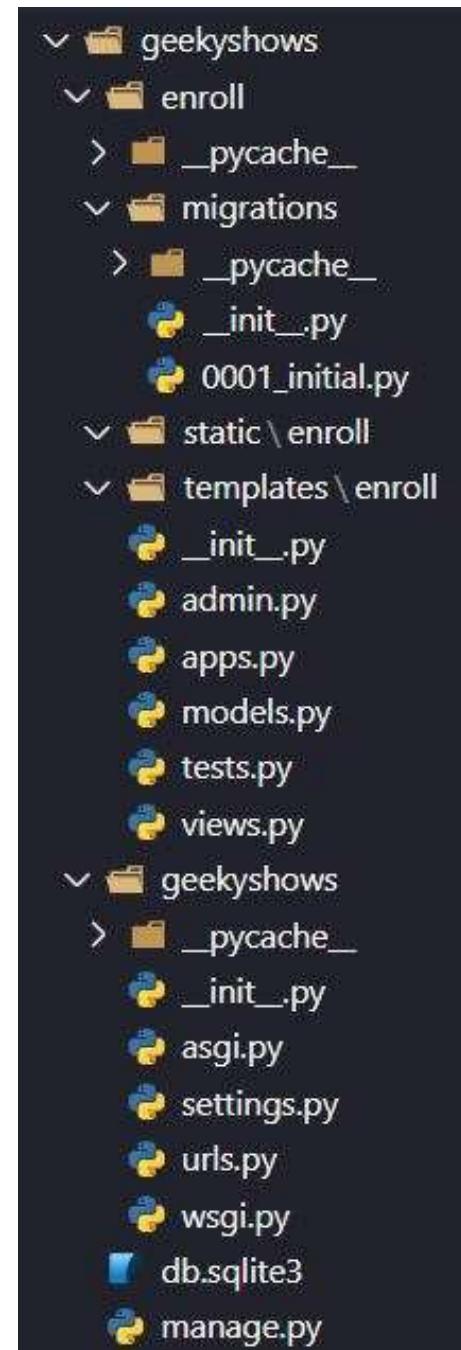
## Register Above Created Class

```
admin.site.register(ModelClassName, ModelAdminClassName)
```

Example: -

```
class StudentAdmin(admin.ModelAdmin):  
    list_display=(‘id’, ‘stuid’, ‘stuname’)
```

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



# list\_display

Set `list_display` to control which fields are displayed on the change list page of the admin. If you don't set `list_display`, the admin site will display a single column that displays the `__str__()` representation of each object.

There are four types of values that can be used in `list_display`:

- The name of a model field.
- A callable that accepts one argument, the model instance.
- A string representing a ModelAdmin method that accepts one argument, the model instance.
- A string representing a model attribute or method (without any required arguments).

# **Register Model by Decorator**

A decorator can be used to register ModelAdmin Classes.

Syntax:- @admin.register(ModelClassName1, ModelClassName2,... ,site=custom\_admin\_site )

## **Register Model Classes**

```
@admin.register(ModelClassName)
```

## **Creating Class**

```
Class ModelAdminClassName(admin.ModelAdmin):
```

```
    list_display=(‘fieldname1’, ‘fieldname2’, .....)
```

Example: -

```
@admin.register(Student)
```

```
class StudentAdmin(admin.ModelAdmin):
```

```
    list_display=(‘id’, ‘stuid’, ‘stuname’)
```

## 29-Django Form

Django's form functionality can simplify and automate vast portions of work like data prepared for display in a form, rendered as HTML, edit using a convenient interface, returned to the server, validated and cleaned up etc and can also do it more securely than most programmers would be able to do in code they wrote themselves.

Django handles three distinct parts of the work involved in forms:

- preparing and restructuring data to make it ready for rendering
- creating HTML forms for the data
- receiving and processing submitted forms and data from the client

# **Bound and Unbound Forms**

If it's bound to a set of data, it's capable of validating that data and rendering the form as HTML with the data displayed in the HTML.

If it's unbound, it cannot do validation (because there's no data to validate!), but it can still render the blank form as HTML.

# Create Django Form using Form Class

To create Django form we have to create a new file inside application folder lets say file name is **forms.py**. Now we can write below code inside **forms.py** to create a form:-

Syntax:-

```
from django import forms

class FormClassName(forms.Form):
    label=forms.FieldType()
    label=forms.FieldType(label='display_label')
```

Example:-

```
from django import forms

class StudentRegistration(forms.Form):
    name=forms.CharField()                      # Here length is not required
    email=forms.EmailField()
```

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

# Display Form to User

- Create an object of Form class in **views.py** then pass object to template files
- Use Form object in template file

# Creating Form object in views.py

First of all create form object inside **views.py** file then pass this object to template file as a dict.

views.py

```
from .forms import StudentRegistration  
  
def showformdata(request):  
    fm = StudentRegistration()  
    return render(request, 'enroll/userregistration.html', {'form':fm})
```

# Get object from views.py in template file

```
templates/enroll / userregistration.html
<!DOCTYPE html>
<html>
  <body>
    {{form}}
  </body>
</html>
```

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

Note - Form object won't provide form tag and button you have to write them manually in template file.

# Get object from views.py in template file

`{{form}}` doesn't add form tag and button so we have to add them manually.

templates/enroll / userregistration.html

```
<!DOCTYPE html>
<html>
  <body>
    <form action ="" method="get">
      {{form}}
      <input type="submit" value="Submit">
    </form>
  </body>
</html>
```

```
<form action ="" method="get">
  <tr>
    <th> <label for="id_name">Name:</label></th>
    <td> <input type="text" name="name" required id="id_name"> </td>
  </tr>
  <tr>
    <th> <label for="id_email">Email:</label></th>
    <td><input type="email" name="email" required id="id_email"></td>
  </tr>
  <input type="submit" value="Submit">
</form>
```

- The output does not include the `<table>` and `</table>` tags, nor does it include the `<form>` and `</form>` tags or an `<input type="submit">` tag. It's your job to do that.
- Each field type has a default HTML representation. `CharField` is represented by an `<input type="text">` and `EmailField` by an `<input type="email">`.
- The HTML *name* for each tag is taken directly from its attribute name in the `StudentRegistration` class.
- The text label for each field e.g. 'Name:' and 'Email:' is generated from the field name by converting all underscores to spaces and upper-casing the first letter.
- Each text label is surrounded in an HTML `<label>` tag, which points to the appropriate form field via its id.
- Its id, in turn, is generated by prepending 'id\_' to the field name. The id attributes and `<label>` tags are included in the output by default.
- The output uses HTML5 syntax, targeting `<!DOCTYPE html>`.

# 30-Form Rendering Options

- {{form}} will render them all
- {{ form.as\_table }} will render them as table cells wrapped in <tr> tags
- {{ form.as\_p }} will render them wrapped in <p> tags
- {{ form.as\_ul }} will render them wrapped in <li> tags
- {{ form.name\_of\_field }} will render field manually as given

```
<form action ="" method="get">  
    {{form}}  
    <input type="submit" value="Submit">  
</form>
```

```
<form action ="" method="get">  
    {{form.as_p}}  
    <input type="submit" value="Submit">  
</form>
```

# Configure id attribute

auto\_id – The id attribute values are generated by prepending id\_ to the form field names. This behavior is configurable, though, if you want to change the id convention or remove HTML id attributes and <label> tags entirely.

Use the auto\_id argument to the Form constructor to control the id and label behavior. This argument must be True, False or a string. By default, auto\_id is set to the string 'id\_%s'.

Example:-

```
fm = StudentRegistration(auto_id=False)  
fm = StudentRegistration(auto_id='geeky')
```

- If auto\_id is set to a string containing the format character '%s', then the form output will include <label> tags, and will generate id attributes based on the format string.
- If auto\_id is set to True, then the form output will include <label> tags and will use the field name as its id for each form field.
- If auto\_id is False, then the form output will not include <label> tags nor id attributes.
- If auto\_id is set to any other true value – such as a string that doesn't include %s – then the library will act as if auto\_id is True.

# Configure label tag

label\_suffix - A translatable string (defaults to a colon (:)) in English) that will be appended after any label name when a form is rendered.

It's possible to customize that character, or omit it entirely, using the label\_suffix parameter.

The label suffix is added only if the last character of the label isn't a punctuation character (in English, those are ., !, ? or :)

```
fm = StudentRegistration(label_suffix=' ')
```

# Dynamic initial values

initial - initial is used to declare the initial value of form fields at runtime.

To accomplish this, use the initial argument to a Form. This argument, if given, should be a dictionary mapping field names to initial values. Only include the fields for which you're specifying an initial value; it's not necessary to include every field in your form.

If a Field defines initial and you include initial when instantiating the Form, then the latter initial will have precedence.

# Ordering Form Field

order\_fields(field\_order) – This method is used to rearrange the fields any time with a list of field names as in field\_order. By default Form.field\_order=None, which retains the order in which you define the fields in your form class.

If field\_order is a list of field names, the fields are ordered as specified by the list and remaining fields are appended according to the default order.

Unknown field names in the list are ignored.

```
fm = StudentRegistration()
```

```
fm.order_fields(field_order=['email', 'name'])
```

# Form Rendering Options

- {{form}} will render them all
- {{ form.as\_table }} will render them as table cells wrapped in <tr> tags
- {{ form.as\_p }} will render them wrapped in <p> tags
- {{ form.as\_ul }} will render them wrapped in <li> tags
- {{ form.name\_of\_field }} will render field manually as given

```
<form action ="" method="get">  
    {{form}}  
    <input type="submit" value="Submit">  
</form>
```

```
<form action ="" method="get">  
    {{form.as_p}}  
    <input type="submit" value="Submit">  
</form>
```

# Render Form Field Manually

Each field is available as an attribute of the form using {{form.name\_of\_field}}

{{ field.label }} - The label of the field.

Example:- {{ form.name.label }}

{{ field.label\_tag }} - The field's label wrapped in the appropriate HTML <label> tag. This includes the form's label\_suffix. The default label\_suffix is a colon:

Example:- {{ form.name.label\_tag }}

{{ field.id\_for\_label }} - The ID that will be used for this field.

Example:- {{ form.name.id\_for\_label }}

# Render Form Field Manually

`{{ field.value }}` - The value of the field.

Example:- `{{ form.name.value }}`

`{{ field.html_name }}` - The name of the field that will be used in the input element's name field. This takes the form prefix into account, if it has been set.

Example:- `{{ form.name.html_name }}`

`{{ field.help_text }}` - Any help text that has been associated with the field.

Example:- `{{ form.name.help_text }}`

`{{ field.field }}` - The Field instance from the form class that this BoundField wraps. You can use it to access Field attributes.

Example:- `{{form.name.field.help_text}}`

# Render Form Field Manually

`{{ field.is_hidden }}` - This attribute is True if the form field is a hidden field and False otherwise. It's not particularly useful as a template variable, but could be useful in conditional tests such as:

```
{% if field.is_hidden %}  
  {# Do something special #}  
{% endif %}
```

# **Loop Form**

- Loop Form Field
- Loop Form Hidden Field and Visible Field

# Render Form Field Manually

`{{ field.errors }}` - It outputs a `<ul class="errorlist">` containing any validation errors corresponding to this field. You can customize the presentation of the errors with a `{% for error in field.errors %}` loop. In this case, each object in the loop is a string containing the error message.

Example:- `{{ form.name.errors }}`

```
<ul class="errorlist">
    <li>Enter Your Name</li>
</ul>
```

`{{ form.non_field_errors }}` - This should be at the top of the form and the template lookup for errors on each field.

Example:- `{{ form.non_field_errors }}`

```
<ul class="errorlist nonfield">
    <li>Generic validation error</li>
</ul>
```

# 31-Form Field

A form's fields are themselves classes; they manage form data and perform validation when a form is submitted.

Syntax:- `FieldType(**kwargs)`

Example:-

`IntegerField()`

`CharField(required)`

`CharField(required, widget=forms.PasswordInput)`

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

# Field Arguments

required - It take True or False value. By default, each Field class assumes the required value is True.

label - The label argument lets you specify the “human-friendly” label for the field. This is used when the Field is displayed in a Form.

label\_suffix - The label\_suffix argument lets you override the form’s label\_suffix on a per-field basis.

initial - The initial argument lets you specify the initial value to use when rendering this Field in an unbound Form.

disabled - The disabled boolean argument, when set to True, disables a form field using the disabled HTML attribute so that it won’t be editable by users. Even if a user tampers with the field’s value submitted to the server, it will be ignored in favor of the value from the form’s initial data.

# Field Arguments

help\_text - The help\_text argument lets you specify descriptive text for this Field. If you provide help\_text, it will be displayed next to the Field when the Field is rendered.

error\_messages - The error\_messages argument lets you override the default messages that the field will raise. Pass in a dictionary with keys matching the error messages you want to override.

Example:- name=forms.CharField(error\_messages={'required':'Enter Your Name'})

validators - The validators argument lets you provide a list of validation functions for the field.

localize - The localize argument enables the localization of form data input, as well as the rendered output.

widget - The widget argument lets you specify a Widget class to use when rendering the Field.

# Widget

A widget is Django's representation of an HTML input element.

The widget handles the rendering of the HTML, and the extraction of data from a GET/POST dictionary that corresponds to the widget.

The HTML generated by the built-in widgets uses HTML5 syntax, targeting <!DOCTYPE html>.

Whenever you specify a field on a form, Django will use a default widget that is appropriate to the type of data that is to be displayed.

Each field type has an appropriate default Widget class, but these can be overridden as required.

Form fields deal with the logic of input validation and are used directly in templates.

Widgets deal with rendering of HTML form input elements on the web page and extraction of raw submitted data.

Example:-

TextInput

Textarea

# Widget

**attrs** - A dictionary containing HTML attributes to be set on the rendered widget.

If you assign a value of True or False to an attribute, it will be rendered as an HTML5 boolean attribute.

Example:-

```
feedback=forms.CharField(widget=forms.TextInput(attrs={'class':'somecss1 somecss2',  
'id':'uniqueid'}))
```

# Built-in Widgets

TextInput - It renders as: <input type="text" ...>

Example:- name = forms.CharField(widget=forms.TextInput)

NumberInput - It renders as: <input type="number" ...>

EmailInput - It renders as: <input type="email" ...>

URLInput – It renders as: <input type="url" ...>

PasswordInput - It renders as: <input type="password" ...>

It take one optional argument *render\_value* which determines whether the widget will have a value filled in when the form is re-displayed after a validation error (default is False).

# Built-in Widgets

HiddenInput - It renders as: <input type="hidden" ...>

DateInput - It renders as: <input type="text" ...>

It takes one optional argument *format*. The format in which this field's initial value will be displayed.

If no *format* argument is provided, the default format is the first format found in  
DATE\_INPUT\_FORMATS.

DateTimeInput - It renders as: <input type="text" ...>

It takes one optional argument *format*. The format in which this field's initial value will be displayed.

If no *format* argument is provided, the default format is the first format found in  
DATETIME\_INPUT\_FORMATS.

By default, the microseconds part of the time value is always set to 0. If microseconds are required, use a subclass with the supports\_microseconds attribute set to True.

# Built-in Widgets

TimeInput - It renders as: <input type="text" ...>

It takes one optional argument *format*. The format in which this field's initial value will be displayed.

If no *format* argument is provided, the default format is the first format found in  
TIME\_INPUT\_FORMATS

Textarea - It renders as: <textarea>...</textarea>

CheckboxInput - It renders as: <input type="checkbox" ...>

It takes one optional argument *check\_test* which is a callable that takes the value of the CheckboxInput and returns True if the checkbox should be checked for that value.

# Built-in Widgets

Select - It renders as: <select><option ...>...</select>

*choices* attribute is optional when the form field does not have a choices attribute. If it does, it will override anything you set here when the attribute is updated on the Field.

NullBooleanSelect - Select widget with options ‘Unknown’, ‘Yes’ and ‘No’

SelectMultiple - Similar to Select, but allows multiple selection: <select multiple>...</select>

RadioSelect - Similar to Select, but rendered as a list of radio buttons within <li> tags:

```
<ul>
  <li><input type="radio" name="..."></li>
  ...
</ul>
```

# Built-in Widgets

CheckboxSelectMultiple - Similar to SelectMultiple, but rendered as a list of checkboxes:

```
<ul>
  <li><input type="checkbox" name="..." ></li>
  ...
</ul>
```

FileInput - It renders as: <input type="file" ...>

ClearableFileInput - It renders as: <input type="file" ...> with an additional checkbox input to clear the field's value, if the field is not required and has initial data.

# Built-in Widgets

MultipleHiddenInput – It renders as: multiple `<input type="hidden" ...>` tags

A widget that handles multiple hidden widgets for fields that have a list of values.

choices - This attribute is optional when the form field does not have a choices attribute. If it does, it will override anything you set here when the attribute is updated on the Field.

SplitDateTimeWidget - Wrapper (using MultiWidget) around two widgets: DateInput for the date, and TimeInput for the time. Must be used with SplitDateTimeField rather than DateTimeField.

SplitDateTimeWidget has several optional arguments:

*date\_format* Similar to DateInput.format

*time\_format* Similar to TimeInput.format

*date\_attrs* and *time\_attrs* Similar to Widget.attrs. A dictionary containing HTML attributes to be set on the rendered DateInput and TimeInput widgets, respectively. If these attributes aren't set, Widget.attrs is used instead.

# Built-in Widgets

`SplitHiddenDateTimeWidget` - Similar to `SplitDateTimeWidget`, but uses `HiddenInput` for both date and time.

`SelectDateWidget` - Wrapper around three `Select` widgets: one each for month, day, and year.

It takes several optional arguments:

*years* - An optional list/tuple of years to use in the “year” select box. The default is a list containing the current year and the next 9 years.

*months* - An optional dict of months to use in the “months” select box.

*empty\_label* - If the `DateField` is not required, `SelectDateWidget` will have an empty choice at the top of the list (default). You can change the text of this label with the *empty\_label* attribute.

*empty\_label* can be a string, list, or tuple. When a string is used, all select boxes will each have an empty choice with this label. If *empty\_label* is a list or tuple of 3 string elements, the select boxes will have their own custom label. The labels should be in this order ('`year_label`', '`month_label`', '`day_label`').

```
<!DOCTYPE html>
<html>
  <body>
    <form method="get">
      {{form}}
      <input type="submit" value="Submit">
    </form>
  </body>
</html>
```

```
<!DOCTYPE html>
<html>
  <body>
    <form method="post">
      {{form}}
      <input type="submit" value="Submit">
    </form>
  </body>
</html>
```

# 32-GET and POST

- GET should be used only for requests that do not affect the state of the system.
- Any request that could be used to change the state of the system should use POST.
- GET would also be unsuitable for a password form, because the password would appear in the URL, and thus, also in browser history and server logs, all in plain text. Neither would it be suitable for large quantities of data, or for binary data, such as an image. A Web application that uses GET requests for admin forms is a security risk: it can be easy for an attacker to mimic a form's request to gain access to sensitive parts of the system.
- POST, coupled with other protections like Django's CSRF protection offers more control over access.
- GET, by contrast, bundles the submitted data into a string, and uses this to compose a URL. The URL contains the address where the data must be sent, as well as the data keys and values.
- Django's login form is returned using the POST method, in which the browser bundles up the form data, encodes it for transmission, sends it to the server, and then receives back its response.
- GET is suitable for things like a web search form, because the URLs that represent a GET request can easily be bookmarked, shared, or resubmitted.

# What is Cross Site Request Forgery (CSRF/ XSRF)

A Cross-site request forgery hole is when a malicious site can cause a visitor's browser to make a request to your server that causes a change on the server. The server thinks that because the request comes with the user's cookies, the user wanted to submit that form.



# What is Cross Site Request Forgery (CSRF)

Depending on which forms on your site are vulnerable, an attacker might be able to do the following to your victims:

- Log the victim out of your site.
- Change the victim's site preferences on your site.
- Post a comment on your site using the victim's login.
- Transfer funds to another user's account.

Attacks can also be based on the victim's IP address rather than cookies:

- Post an anonymous comment that is shown as coming from the victim's IP address.
- Modify settings on a device such as a wireless router or cable modem.
- Modify an intranet wiki page.
- Perform a distributed password-guessing attack without a botnet.

# Cross Site Request Forgery (CSRF) Token

Django provides CSRF Protection with csrf\_token which we need to add inside form tag. This token will add a hidden input field with random value in form tag.

templates/enroll / userregistration.html

```
<!DOCTYPE html>
<html>
  <body>
    <form method="post"> {% csrf_token %}
      {{form}}
      <input type="submit" value="Submit">
    </form>
  </body>
</html>
```

# Checking which form data has changed

`has_changed()` - This method is used on your Form when you need to check if the form data has been changed from the initial data.

`has_changed()` will be True if the data from `request.POST` differs from what was provided in `initial` or False otherwise. The result is computed by calling `Field.has_changed()` for each field in the form.

Example:- `Form.has_changed()`

`changed_data` - The `changed_data` attribute returns a list of the names of the fields whose values in the form's bound data (usually `request.POST`) differ from what was provided in `initial`. It returns an empty list if no data differs.

Example:- `Form.changed_data`

# **33-Get Django Form Data**

- Validate Data/ Field Validation
- Get Cleaned Data

# How to send GET Request

- Open browser and write url hit enter this is by default get request
- Anchor Tag
- Form tag contains method='GET'
- Form tag with specifying method is by default GET

# How to send POST Request

- form tag contains method='POST'

# Django Form and Field Validation

`is_valid( )` - This method is used to run validation and return a Boolean designating whether the data was valid as True or not as False. This returns True or False.

Syntax:- `Form.is_valid()`

`cleaned_data` – This attribute is used to access clean data. Each field in a Form class is responsible not only for validating data, but also for “cleaning” it normalizing it to a consistent format. This is a nice feature, because it allows data for a particular field to be input in a variety of ways, always resulting in consistent output. Once you’ve created a Form instance with a set of data and validated it, you can access the clean data via its `cleaned_data` attribute.

Any text based field such as `CharField` or `EmailField` always cleans the input into a string.

If your data does not validate, the `cleaned_data` dictionary contains only the valid fields.

`cleaned_data` will always only contain a key for fields defined in the Form, even if you pass extra data when you define the Form.

When the Form is valid, `cleaned_data` will include a key and value for all its fields, even if the data didn’t include a value for some optional fields.

# Get Django Form Data in Terminal

## 1. First of all Create form inside forms.py file

```
forms.py
from django import forms
class StudentRegistration(forms.Form):
    name=forms.CharField()
    email=forms.EmailField()
```

## 2. Get submitted Data in views.py

```
views.py
from .forms import StudentRegistration
def showformdata(request):
    if request.method=='POST':
        fm = StudentRegistration(request.POST)
        if fm.is_valid():
            print('Form Validated')
            print('Name:', fm.cleaned_data['name'])
            print('email:', fm.cleaned_data['email'])
    else:
        fm = StudentRegistration()
    return render(request, 'enroll/userregistration.html', {'form':fm})
```

## 3. Get object from views.py in template file

```
templates/enroll / userregistration.html
<!DOCTYPE html>
<html>
    <body>
        <form method="POST"> {% csrf_token% }
            {{form.as_p}}
        <input type="submit" value="Submit">
    </form>
    </body>
</html>
```

# 34-Form Field

A form's fields are themselves classes; they manage form data and perform validation when a form is submitted.

Syntax:- `FieldType(**kwargs)`

Example:-

`IntegerField()`

`CharField(required)`

`CharField(required, widget=forms.PasswordInput)`

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

# Built-in Fields

CharField(\*\*kwargs)

Default widget: TextInput

Empty value: Whatever you've given as empty\_value.

Normalizes to: A string.

Uses MaxLengthValidator and MinLengthValidator if max\_length and min\_length are provided. Otherwise, all inputs are valid.

Error message keys: required, max\_length, min\_length

It has three optional arguments for validation:

*max\_length* and *min\_length* - If provided, these arguments ensure that the string is at most or at least the given length.

*strip* - If True (default), the value will be stripped of leading and trailing whitespace.

*empty\_value* - The value to use to represent “empty”. Defaults to an empty string.

Example:- name=forms.CharField(min\_length=5, max\_length=50, error\_messages={'required':'Enter Your Name'})

# Built-in Fields

BooleanField(\*\*kwargs)

Default widget: CheckboxInput

Empty value: False

Normalizes to: A Python True or False value.

Validates that the value is True (e.g. the check box is checked) if the field has required=True.

Error message keys: required

# Built-in Fields

IntegerField(\*\*kwargs)

Default widget: NumberInput when Field.localize is False, else TextInput.

Empty value: None

Normalizes to: A Python integer.

Validates that the given value is an integer. Uses MaxValueValidator and MinValueValidator if max\_value and min\_value are provided. Leading and trailing whitespace is allowed, as in Python's int() function.

Error message keys: required, invalid, max\_value, min\_value

The max\_value and min\_value error messages may contain %(limit\_value)s, which will be substituted by the appropriate limit.

It takes two optional arguments for validation:

*max\_value* and *min\_value* - These control the range of values permitted in the field.

# Built-in Fields

DecimalField(\*\*kwargs)

Default widget: NumberInput when Field.localize is False, else TextInput.

Empty value: None

Normalizes to: A Python decimal.

Validates that the given value is a decimal. Uses MaxValueValidator and MinValueValidator if max\_value and min\_value are provided. Leading and trailing whitespace is ignored.

Error message keys: required, invalid, max\_value, min\_value, max\_digits, max\_decimal\_places, max\_whole\_digits

The max\_value and min\_value error messages may contain %(limit\_value)s, which will be substituted by the appropriate limit. Similarly, the max\_digits, max\_decimal\_places and max\_whole\_digits error messages may contain %(max)s.

It takes four optional arguments:

*max\_value* and *min\_value* - These control the range of values permitted in the field, and should be given as decimal.Decimal values.

*max\_digits* - The maximum number of digits (those before the decimal point plus those after the decimal point, with leading zeros stripped) permitted in the value.

*decimal\_places* - The maximum number of decimal places permitted.

# Built-in Fields

FloatField(\*\*kwargs)

Default widget: NumberInput when Field.localize is False, else TextInput.

Empty value: None

Normalizes to: A Python float.

Validates that the given value is a float. Uses MaxValueValidator and MinValueValidator if max\_value and min\_value are provided. Leading and trailing whitespace is allowed, as in Python's float() function.

Error message keys: required, invalid, max\_value, min\_value

It takes two optional arguments for validation, *max\_value* and *min\_value*. These control the range of values permitted in the field.

# Built-in Fields

SlugField(\*\*kwargs)

Default widget: TextInput

Empty value: " (an empty string)

Normalizes to: A string.

Uses validate\_slug or validate\_unicode\_slug to validate that the given value contains only letters, numbers, underscores, and hyphens.

Error messages: required, invalid

This field is intended for use in representing a model SlugField in forms.

It takes an optional parameter:

*allow\_Unicode* - A boolean instructing the field to accept Unicode letters in addition to ASCII letters.  
Defaults to False.

# Built-in Fields

EmailField(\*\*kwargs)

Default widget: EmailInput

Empty value: " (an empty string)

Normalizes to: A string.

Uses EmailValidator to validate that the given value is a valid email address, using a moderately complex regular expression.

Error message keys: required, invalid

It has two optional arguments for validation, *max\_length* and *min\_length*. If provided, these arguments ensure that the string is at most or at least the given length.

# Built-in Fields

URLField(\*\*kwargs)

Default widget: URLInput

Empty value: " (an empty string)

Normalizes to: A string.

Uses URLValidator to validate that the given value is a valid URL.

Error message keys: required, invalid

It takes the following optional arguments:

*max\_length* and *min\_length* - These are the same as CharField.max\_length and CharField.min\_length.

# Built-in Fields

DateField(\*\*kwargs)

Default widget: DateInput

Empty value: None

Normalizes to: A Python datetime.date object.

Validates that the given value is either a datetime.date, datetime.datetime or string formatted in a particular date format.

Error message keys: required, invalid

It takes one optional argument:

*input\_formats* - A list of formats used to attempt to convert a string to a valid datetime.date object.

If no input\_formats argument is provided, the default input formats are taken from DATE\_INPUT\_FORMATS if USE\_L10N is False, or from the active locale format DATE\_INPUT\_FORMATS key if localization is enabled. See also format localization.

# Built-in Fields

TimeField(\*\*kwargs)

Default widget: TimeInput

Empty value: None

Normalizes to: A Python datetime.time object.

Validates that the given value is either a datetime.time or string formatted in a particular time format.

Error message keys: required, invalid

It takes one optional argument:

*input\_formats* - A list of formats used to attempt to convert a string to a valid datetime.time object.

If no input\_formats argument is provided, the default input formats are taken from TIME\_INPUT\_FORMATS if USE\_L10N is False, or from the active locale format TIME\_INPUT\_FORMATS key if localization is enabled. See also format localization.

# Built-in Fields

DateTimeField(\*\*kwargs)

Default widget: DateTimeInput

Empty value: None

Normalizes to: A Python datetime.datetime object.

Validates that the given value is either a datetime.datetime, datetime.date or string formatted in a particular datetime format.

Error message keys: required, invalid

It takes one optional argument:

*input\_formats* - A list of formats used to attempt to convert a string to a valid datetime.datetime object.

If no input\_formats argument is provided, the default input formats are taken from DATETIME\_INPUT\_FORMATS if USE\_L10N is False, or from the active locale format DATETIME\_INPUT\_FORMATS key if localization is enabled. See also format localization.

# Built-in Fields

DurationField(\*\*kwargs)

Default widget: TextInput

Empty value: None

Normalizes to: A Python timedelta.

Validates that the given value is a string which can be converted into a timedelta. The value must be between datetime.timedelta.min and datetime.timedelta.max.

Error message keys: required, invalid, overflow.

Accepts any format understood by parse\_duration().

# Built-in Fields

FileField(\*\*kwargs)

Default widget: ClearableFileInput

Empty value: None

Normalizes to: An UploadedFile object that wraps the file content and file name into a single object.

Can validate that non-empty file data has been bound to the form.

Error message keys: required, invalid, missing, empty, max\_length

It has two optional arguments for validation, *max\_length* and *allow\_empty\_file*. If provided, these ensure that the file name is at most the given length, and that validation will succeed even if the file content is empty.

To learn more about the UploadedFile object.

When you use a FileField in a form, you must also remember to bind the file data to the form.

The max\_length error refers to the length of the filename. In the error message for that key, %(max)d will be replaced with the maximum filename length and %(length)d will be replaced with the current filename length.

# Built-in Fields

FilePathField(\*\*kwargs)

Default widget: Select

Empty value: " (an empty string)

Normalizes to: A string.

Validates that the selected choice exists in the list of choices.

Error message keys: required, invalid\_choice

The field allows choosing from files inside a certain directory. It takes five extra arguments; only path is required:

path - The absolute path to the directory whose contents you want listed. This directory must exist.

recursive - If False (the default) only the direct contents of path will be offered as choices. If True, the directory will be descended into recursively and all descendants will be listed as choices.

match - A regular expression pattern; only files with names matching this expression will be allowed as choices.

allow\_files - Optional. Either True or False. Default is True. Specifies whether files in the specified location should be included. Either this or allow\_folders must be True.

allow\_folders - Optional. Either True or False. Default is False. Specifies whether folders in the specified location should be included. Either this or allow\_files must be True.

# Built-in Fields

ImageField(\*\*kwargs)

Default widget: ClearableFileInput

Empty value: None

Normalizes to: An UploadedFile object that wraps the file content and file name into a single object.

Validates that file data has been bound to the form. Also uses FileExtensionValidator to validate that the file extension is supported by Pillow.

Error message keys: required, invalid, missing, empty, invalid\_image

Using an ImageField requires that Pillow is installed with support for the image formats you use.

# Built-in Fields

ChoiceField(\*\*kwargs)

Default widget: Select

Empty value: " (an empty string)

Normalizes to: A string.

Validates that the given value exists in the list of choices.

Error message keys: required, invalid\_choice

The invalid\_choice error message may contain %(value)s, which will be replaced with the selected choice.

It takes one extra argument:

choices - Either an iterable of 2-tuples to use as choices for this field, or a callable that returns such an iterable. This argument accepts the same formats as the choices argument to a model field. See the model field reference documentation on choices for more details. If the argument is a callable, it is evaluated each time the field's form is initialized. Defaults to an empty list.

# Built-in Fields

TypedChoiceField(\*\*kwargs) - Just like a ChoiceField, except TypedChoiceField takes two extra arguments, coerce and empty\_value.

Default widget: Select

Empty value: Whatever you've given as empty\_value.

Normalizes to: A value of the type provided by the coerce argument.

Validates that the given value exists in the list of choices and can be coerced.

Error message keys: required, invalid\_choice

It takes extra arguments:

coerce - A function that takes one argument and returns a coerced value. Examples include the built-in int, float, bool and other types. Defaults to an identity function. Note that coercion happens after input validation, so it is possible to coerce to a value not present in choices.

empty\_value - The value to use to represent “empty.” Defaults to the empty string; None is another common choice here. Note that this value will not be coerced by the function given in the coerce argument, so choose it accordingly.

# **Built-in Fields**

UUIDField(\*\*kwargs)

Default widget: TextInput

Empty value: " (an empty string)

Normalizes to: A UUID object.

Error message keys: required, invalid

This field will accept any string format accepted as the hex argument to the UUID constructor.

# Built-in Fields

ComboField(\*\*kwargs)

Default widget: TextInput

Empty value: " (an empty string)

Normalizes to: A string.

Validates the given value against each of the fields specified as an argument to the ComboField.

Error message keys: required, invalid

It takes one extra required argument:

fields - The list of fields that should be used to validate the field's value (in the order in which they are provided).

# Built-in Fields

GenericIPAddressField(\*\*kwargs) - A field containing either an IPv4 or an IPv6 address.

Default widget: TextInput

Empty value: " (an empty string)

Normalizes to: A string. IPv6 addresses are normalized as described below.

Validates that the given value is a valid IP address.

Error message keys: required, invalid

The IPv6 address normalization follows RFC 4291#section-2.2 section 2.2, including using the IPv4 format suggested in paragraph 3 of that section, like ::ffff:192.0.2.0. For example, 2001:0::0:01 would be normalized to 2001::1, and ::ffff:0a0a:0a0a to ::ffff:10.10.10.10. All characters are converted to lowercase.

It takes two optional arguments:

protocol - Limits valid inputs to the specified protocol. Accepted values are both (default), IPv4 or IPv6. Matching is case insensitive.

unpack\_ipv4 - Unpacks IPv4 mapped addresses like ::ffff:192.0.2.1. If this option is enabled that address would be unpacked to 192.0.2.1. Default is disabled. Can only be used when protocol is set to 'both'.

# Built-in Fields

MultipleChoiceField(\*\*kwargs)

Default widget: SelectMultiple

Empty value: [] (an empty list)

Normalizes to: A list of strings.

Validates that every value in the given list of values exists in the list of choices.

Error message keys: required, invalid\_choice, invalid\_list

The invalid\_choice error message may contain %(value)s, which will be replaced with the selected choice.

It takes one extra required argument, choices, as for ChoiceField.

# Built-in Fields

TypedMultipleChoiceField(\*\*kwargs)

Just like a MultipleChoiceField, except TypedMultipleChoiceField takes two extra arguments, coerce and empty\_value.

Default widget: SelectMultiple

Empty value: Whatever you've given as empty\_value

Normalizes to: A list of values of the type provided by the coerce argument.

Validates that the given values exists in the list of choices and can be coerced.

Error message keys: required, invalid\_choice

The invalid\_choice error message may contain %(value)s, which will be replaced with the selected choice.

It takes two extra arguments, coerce and empty\_value, as for TypedChoiceField.

# Built-in Fields

NullBooleanField(\*\*kwargs)

Default widget: NullBooleanSelect

Empty value: None

Normalizes to: A Python True, False or None value.

Validates nothing (i.e., it never raises a ValidationError).

# Built-in Fields

RegexField(\*\*kwargs)

Default widget: TextInput

Empty value: " (an empty string)

Normalizes to: A string.

Uses RegexValidator to validate that the given value matches a certain regular expression.

Error message keys: required, invalid

Takes one required argument:

regex - A regular expression specified either as a string or a compiled regular expression object.

Also takes max\_length, min\_length, and strip, which work just as they do for CharField.

strip - Defaults to False. If enabled, stripping will be applied before the regex validation.

# Built-in Fields

MultiValueField(fields=(), \*\*kwargs)

Default widget: TextInput

Empty value: " (an empty string)

Normalizes to: the type returned by the compress method of the subclass.

Validates the given value against each of the fields specified as an argument to the MultiValueField.

Error message keys: required, invalid, incomplete

Aggregates the logic of multiple fields that together produce a single value.

This field is abstract and must be subclassed. In contrast with the single-value fields, subclasses of MultiValueField must not implement clean() but instead - implement compress().

Takes one extra required argument:

fields - A tuple of fields whose values are cleaned and subsequently combined into a single value. Each value of the field is cleaned by the corresponding field in fields – the first value is cleaned by the first field, the second value is cleaned by the second field, etc. Once all fields are cleaned, the list of clean values is combined into a single value by compress().

# Built-in Fields

Also takes some optional arguments:

`require_all_fields` - Defaults to True, in which case a required validation error will be raised if no value is supplied for any field.

When set to False, the `Field.required` attribute can be set to False for individual fields to make them optional. If no value is supplied for a required field, an incomplete validation error will be raised.

A default incomplete error message can be defined on the `MultiValueField` subclass, or different messages can be defined on each individual field.

`widget` - Must be a subclass of `django.forms.MultiWidget`. Default value is `TextInput`, which probably is not very useful in this case.

`compress(data_list)` - Takes a list of valid values and returns a “compressed” version of those values – in a single value. For example, `SplitDateTimeField` is a subclass which combines a time field and a date field into a `datetime` object.

This method must be implemented in the subclasses.

# Built-in Fields

SplitDateTimeField(\*\*kwargs)

Default widget: SplitDateTimeWidget

Empty value: None

Normalizes to: A Python datetime.datetime object.

Validates that the given value is a datetime.datetime or string formatted in a particular datetime format.

Error message keys: required, invalid, invalid\_date, invalid\_time

Takes two optional arguments:

input\_date\_formats - A list of formats used to attempt to convert a string to a valid datetime.date object.

If no input\_date\_formats argument is provided, the default input formats for DateField are used.

input\_time\_formats - A list of formats used to attempt to convert a string to a valid datetime.time object.

If no input\_time\_formats argument is provided, the default input formats for TimeField are used.

# Built-in Fields

ModelChoiceField(\*\*kwargs)

Default widget: Select

Empty value: None

Normalizes to: A model instance.

Validates that the given id exists in the queryset.

Error message keys: required, invalid\_choice

Allows the selection of a single model object, suitable for representing a foreign key. Note that the default widget for ModelChoiceField becomes impractical when the number of entries increases. You should avoid using it for more than 100 items.

A single argument is required:

queryset - A QuerySet of model objects from which the choices for the field are derived and which is used to validate the user's selection. It's evaluated when the form is rendered.

# Built-in Fields

ModelChoiceField also takes two optional arguments:

`empty_label` - By default the `<select>` widget used by ModelChoiceField will have an empty choice at the top of the list. You can change the text of this label (which is "-----" by default) with the `empty_label` attribute, or you can disable the empty label entirely by setting `empty_label` to `None`:

Note that if a ModelChoiceField is required and has a default initial value, no empty choice is created (regardless of the value of `empty_label`).

`to_field_name` - This optional argument is used to specify the field to use as the value of the choices in the field's widget. Be sure it's a unique field for the model, otherwise the selected value could match more than one object. By default it is set to `None`, in which case the primary key of each object will be used.

The `__str__()` method of the model will be called to generate string representations of the objects for use in the field's choices. To provide customized representations, subclass ModelChoiceField and override `label_from_instance`. This method will receive a model object and should return a string suitable for representing it.

# Built-in Fields

ModelMultipleChoiceField(\*\*kwargs)

Default widget: SelectMultiple

Empty value: An empty QuerySet (self.queryset.none())

Normalizes to: A QuerySet of model instances.

Validates that every id in the given list of values exists in the queryset.

Error message keys: required, list, invalid\_choice, invalid\_pk\_value

The invalid\_choice message may contain %(value)s and the invalid\_pk\_value message may contain %(pk)s, which will be substituted by the appropriate values.

Allows the selection of one or more model objects, suitable for representing a many-to-many relation. As with ModelChoiceField, you can use label\_from\_instance to customize the object representations.

A single argument is required:

queryset - Same as ModelChoiceField.queryset.

Takes one optional argument:

to\_field\_name - Same as ModelChoiceField.to\_field\_name.

# 35-Cleaning and Validating Specific Field

`clean_<fieldname>()` - This method is called on a form subclass where <fieldname> is replaced with the name of the form field attribute.

This method does any cleaning that is specific to that particular attribute, unrelated to the type of field that it is.

This method is not passed any parameters.

You will need to look up the value of the field in `self.cleaned_data` and remember that it will be a Python object at this point, not the original string submitted in the form.

# Cleaning and Validating Specific Field

forms.py

```
from django import forms
class StudentRegistration(forms.Form):
    name=forms.CharField()
    email=forms.EmailField()
    password=forms.CharField(widget=forms.PasswordInput)
    def clean_name(self):
        valname = self.cleaned_data['name'] //      valname = self.cleaned_data.get('name')
        if len(valname) < 4:
            raise forms.ValidationError('Enter more than or equal 4')
        return valname
```

# Validation of Complete Django Form at once

`clean()` – The `clean()` method on a `Field` subclass is responsible for running `to_python()`, `validate()`, and `run_validators()` in the correct order and propagating their errors.

If, at any time, any of the methods raise `ValidationError`, the validation stops and that error is raised.

This method returns the clean data, which is then inserted into the `cleaned_data` dictionary of the form.

Implement a `clean()` method on your Form when you must add custom validation for fields that are interdependent.

Syntax:- `Form.clean()`

# Validation of Complete Django Form at once

forms.py

```
from Django import forms
class StudentRegistration(forms.Form):
    name=forms.CharField()
    email=forms.EmailField()
    def clean(self):
        cleaned_data = super().clean()
        valname=self.cleaned_data['name']
        if len(valname)<4:
            raise forms.ValidationError('Name should be more than or equal 4')
        valemail=self.cleaned_data['email']
        if len(valemail)<10:
            raise forms.ValidationError('Email should be more than or equal 10')
```

# Using Built-in Validators

We can use Built-in Validators, available in django.core module.

forms.py

```
form django.core import validators  
from django import forms  
  
class StudentRegistration(forms.Form):  
    name=forms.CharField(validators=[validators.MaxLengthValidator(10)])  
    email=forms.EmailField()
```

# Create Custom Form Validators

forms.py

```
form django.core import validators  
from django import forms  
  
def starts_with_s(value):  
    if value[0] != 's':  
        raise forms.ValidationError('Name should start with s')
```

```
class StudentRegistration(forms.Form):
```

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

# Form Validation – Match Two field value

Name	<input type="text"/>
Email	<input type="text"/>
Password	<input type="text"/>
Re-Enter Password	<input type="text"/>
<input type="button" value="Submit"/>	

# Form Validation – Match Two field value

forms.py

```
from Django import forms  
  
class StudentRegistration(forms.Form):  
    name=forms.CharField()  
    password=forms.CharField(widget=forms.PasswordInput)  
    rpassword=forms.CharField(widget=forms.PasswordInput)  
  
    def clean(self):  
        cleaned_data = super().clean() ←  
        valpwd=cleaned_data['password']           // cleaned_data.get('password')  
        valrpwd=cleaned_data['rpassword']  
  
        if valpwd != valrpwd :  
            raise forms.ValidationError('Password Not Matched')
```

The call to super().clean() ensures that any validation logic in parent classes is maintained.

# 36-Field Error

`{{ field.errors }}` - It outputs a `<ul class="errorlist">` containing any validation errors corresponding to this field. You can customize the presentation of the errors with a `{% for error in field.errors %}` loop. In this case, each object in the loop is a string containing the error message.

Example:- `{{ form.name.errors }}`

```
<ul class="errorlist">
    <li>Enter Your Name</li>
</ul>
```

`{{ form.non_field_errors }}` - This should be at the top of the form and the template lookup for errors on each field.

Example:- `{{ form.non_field_errors }}`

```
<ul class="errorlist nonfield">
    <li>Generic validation error</li>
</ul>
```

# Styling Django Form Errors

If you render a bound Form object, the act of rendering will automatically run the form's validation if it hasn't already happened, and the HTML output will include the validation errors as a `<ul class="errorlist">` near the field.

We can use this class `errorlist` to style error.

# Styling Django Form Errors

error\_css\_class and required\_css\_class – These Form class hooks can be used to add class attributes to required rows or rows with errors. Rows will be given "error" and/or "required" classes, as needed.

Example:-

```
class StudentRegistration(forms.Form):
```

```
    error_css_class = 'error'
```

```
    required_css_class = 'required'
```

# 37-Form API

## **forms.py**

```
from django.core import validators
from django import forms
class StudentRegistration(forms.Form):
    name = forms.CharField()
    email = forms.EmailField()
    password = forms.CharField(widget=forms.PasswordInput)
```

## **views.py**

```
def showformdata(request):
    if request.method == 'POST':
        fm = StudentRegistration(request.POST)
        if fm.is_valid():
            nm = fm.cleaned_data['name']
            em = fm.cleaned_data['email']
            pw = fm.cleaned_data['password']
            reg = User(name=nm, email=em, password=pw)
            reg.save()
```

## **models.py**

```
from django.db import models
class User(models.Model):
    name = models.CharField(max_length=70)
    email = models.EmailField(max_length=100)
    password = models.CharField(max_length=100)
```

## **templatefile.html**

```
<form action="" method="POST" novalidate>
    {% csrf_token %}
    {{form.as_p}}
    <input type="submit" value="Submit">
</form>
```

# Model Form

## **forms.py**

```
from django.core import validators
from django import forms
class StudentRegistration(forms.Form):
    class Meta(forms.ModelForm):
        fields = ['name', 'password', 'email']
        password = forms.CharField(widget=forms.PasswordInput)
```

## **views.py**

```
def showformdata(request):
    if request.method == 'POST':
        fm = StudentRegistration(request.POST)
        if fm.is_valid():
            nm = fm.cleaned_data['name']
            em = fm.cleaned_data['email']
            pw = fm.cleaned_data['password']
            reg = User(name=nm, email=em, password=pw)
            reg.save()
```

## **models.py**

```
from django.db import models
class User(models.Model):
    name = models.CharField(max_length=70)
    email = models.EmailField(max_length=100)
    password = models.CharField(max_length=100)
```

## **templatefile.html**

```
<form action="" method="POST" novalidate>
    {% csrf_token %}
    {{form.as_p}}
    <input type="submit" value="Submit">
</form>
```

# Model Form

Django provides a helper class that lets you create a Form class from a Django model. This helper class is called as ModelForm.

ModelForm is a regular Form which can automatically generate certain fields.

The fields that are automatically generated depend on the content of the Meta class and on which fields have already been defined declaratively.

Steps:-

- Create Model Class
- Create ModelForm Class

Syntax:-

forms.py

```
class ModelFormClassName(forms.ModelForm):  
    class Meta:  
        model = ModelClassName  
        fields = ['fieldname1', 'fieldname2', 'fieldname3']  
        fields = ('fieldname1', 'fieldname2', 'fieldname3')
```

```
class Registration(forms.ModelForm):  
    class Meta:  
        model = User  
        fields = ['name', 'password', 'email']
```

# Model Form

<b>Model Field</b>	<b>Form Field</b>
AutoField	Not Represented in the Form
BigAutoField	Not Represented in the Form
BigIntegerField	IntegerField with min_value set to -9223372036854775808 and max_value set to 9223372036854775807.
BinaryField	CharField, if editable is set to True on the model field, otherwise not represented in the form.
BooleanField	BooleanField, or NullBooleanField if null=True.
CharField	CharField with max_length set to the model field's max_length and empty_value set to None if null=True.
DateField	DateField
DateTimeField	DateTimeField
DecimalField	DecimalField
DurationField	DurationField

# Model Form

<b>Model Field</b>	<b>Form Field</b>
EmailField	EmailField
FileField	FileField
FilePathField	FilePathField
FloatField	FloatField
ForeignKey	ModelChoiceField
ImageField	ImageField
IntegerField	IntegerField
IPAddressField	IPAddressField
GenericIPAddressField	GenericIPAddressField
ManyToManyField	ModelMultipleChoiceField
NullBooleanField	NullBooleanField
PositiveIntegerField	IntegerField

# Model Form

<b>Model Field</b>	<b>Form Field</b>
PositiveSmallIntegerField	IntegerField
SlugField	SlugField
SmallAutoField	Not represented in the form
SmallIntegerField	IntegerField
TextField	CharField with widget=forms.Textarea
TimeField	TimeField
URLField	URLField
UUIDField	UUIDField

# Model Form

- If the model field has *blank=True*, then *required* is set to *False* on the form field. Otherwise, *required=True*.
- The form field's *label* is set to the *verbose\_name* of the model field, with the first character capitalized.
- The form field's *help\_text* is set to the *help\_text* of the model field.
- If the model field has *choices* set, then the form field's widget will be set to *Select*, with choices coming from the model field's choices. The choices will normally include the blank choice which is selected by default. If the field is required, this forces the user to make a selection. The blank choice will not be included if the model field has *blank=False* and an explicit *default* value (the default value will be initially selected instead).

# Model Form

```
class Registration(forms.ModelForm):  
    class Meta:  
        model = User  
        fields = ['name', 'password', 'email']  
        labels = {'name': 'Enter Name', 'password': 'Enter Password', 'email': 'Enter Email'}  
        help_text = {'name': 'Enter Your Full Name'}  
        error_messages = {'name': {'required': 'Naam Likhna Jaruri Hai'},  
                         'password': {'required': 'Password Likhna Jaruri Hai'}}  
        widgets = { 'password': forms.PasswordInput,  
                   'name': forms.TextInput(attrs={'class': 'myclass', 'placeholder': 'Yaha Naam likhe'})}, }
```

# Model Form

```
class Registration(forms.ModelForm):  
    name = forms.CharField(max_length=50, required=False)  
  
    class Meta:  
        model = User  
        fields = ['name', 'password', 'email']  
        labels = {'name': 'Enter Name', 'password': 'Enter Password', 'email': 'Enter Email'}  
        widgets = {'password':forms.PasswordInput}
```

# save () Method

save (commit=False/True) Method - This method creates and saves a database object from the data bound to the form.

A subclass of ModelForm can accept an existing model instance as the keyword argument instance, if this is supplied, save() will update that instance.

If it's not supplied, save() will create a new instance of the specified model.

If the form hasn't been validated, calling save( ) will do so by checking form.errors.

Syntax:- save (commit=False/True)

If *commit=False*, then it will return an object that hasn't yet been saved to the database. This is useful if you want to do custom processing on the object before saving it, or if you want to use one of the specialized model saving options.

If your model has a many-to-many relation and you specify *commit=False* when you save a form, Django cannot immediately save the form data for the many-to-many relation. This is because it isn't possible to save many-to-many data for an instance until the instance exists in the database.

# Selecting Fields

- Set the fields attribute to field names

```
class Registration(forms.ModelForm):
```

```
    class Meta:
```

```
        model = User
```

```
        fields = ['name', 'password', 'email']
```

- Set the fields attribute to the special value `__all__` to indicate that all fields in the model should be used.

```
class Registration(forms.ModelForm):
```

```
    class Meta:
```

```
        model = User
```

```
        fields = '__all__'
```

# Selecting Fields

- Set the exclude attribute of the ModelForm's inner Meta class to a list of fields to be excluded from the form.

```
class Registration(forms.ModelForm):
```

```
    class Meta:
```

```
        model = User
```

```
        exclude = ['name']
```

# ModelForm Inheritance

You can extend and reuse ModelForms by inheriting them. This is useful if you need to declare extra fields or extra methods on a parent class for use in a number of forms derived from models.

You can also subclass the parent's Meta inner class if you want to change the Meta.fields or Meta.exclude lists.

```
class StudentRegistration(forms.ModelForm):
```

```
    class Meta:
```

```
        model = User
```

```
        fields = ['student_name', 'email', 'password']
```

```
class TeacherRegistration(StudentRegistration):
```

```
    class Meta(StudentRegistration.Meta):
```

```
        fields = ['teacher_name', 'email', 'password']
```

```
class User(models.Model):
```

```
    student_name = models.CharField(max_length=100)
```

```
    teacher_name = models.CharField(max_length=100)
```

```
    email = models.EmailField(max_length=100)
```

```
    password = models.CharField(max_length=100)
```

# ModelForm Inheritance

- Normal Python name resolution rules apply. If you have multiple base classes that declare a Meta inner class, only the first one will be used. This means the child's Meta, if it exists, otherwise the Meta of the first parent, etc.
- It's possible to inherit from both Form and ModelForm simultaneously, however, you must ensure that ModelForm appears first in the MRO. This is because these classes rely on different metaclasses and a class can only have one metaclass.
- It's possible to declaratively remove a Field inherited from a parent class by setting the name to be None on the subclass.

# 38-Dynamic URL

```
urlpatterns = [  
    path('student/', views.show_details, name="detail"),  
    path('student/<my_id>', views.show_details, name="detail"),  
    path('student/<int:my_id>', views.shows_details, name="detail"),  
    path('student /<int:my_id>/<int:my_subid>', views.shows_details, name="detail"),  
    path('student/<int:id>/<int:subid>/<slug:my_slug>', views.shows_details, name="detail"),  
]
```

# Path Converters

- str - Matches any non-empty string, excluding the path separator, '/'. This is the default if a converter isn't included in the expression.
- int - Matches zero or any positive integer. Returns an int.
- slug - Matches any slug string consisting of ASCII letters or numbers, plus the hyphen and underscore characters. For example, building-your-1st-django-site.
- uid - Matches a formatted UUID. To prevent multiple URLs from mapping to the same page, dashes must be included and letters must be lowercase. For example, 075194d3-6885-417e-a8a8-6c931e272f00. Returns a UUID instance.
- path - Matches any non-empty string, including the path separator, '/'. This allows you to match against a complete URL path rather than a segment of a URL path as with str.

# Specifying defaults for view arguments

## urls.py

```
from django.urls import path
from . import views
urlpatterns = [
    path('blog/', views.page),
    path('blog/page/<int:num>/', views.page),
]
```

## views.py

```
def page(request, num=1):
```

# Passing extra options to view functions

```
path(route, view, kwargs=None, name=None)
```

The `kwargs` argument allows you to pass additional arguments to the view function or method. It should be a dictionary.

## urls.py

```
urlpatterns = [  
    path(route, view, kwargs=None, name=None)  
]
```

## urls.py

```
urlpatterns = [  
    path('', views.home, {'check': 'OK'}),  
]
```

## views.py

```
def home(request, check):  
    print(check)
```

# Custom Path Converters

## Create a Path Converter Class

```
class FourDigitYearConverter:  
    regex = '[0-9]{4}'  
  
    def to_python(self, value):  
        return int(value)  
  
    def to_url(self, value):  
        return '%04d' % value
```



converters.py

Where

- regex is an attribute, as a string.
- to\_python(self, value) method, which handles converting the matched string into the type that should be passed to the view function. It should raise ValueError if it can't convert the given value. A ValueError is interpreted as no match and as a consequence a 404 response is sent to the user unless another URL pattern matches.
- to\_url(self, value) method, which handles converting the Python type into a string to be used in the URL.

# Custom Path Converters

## Register Path Converter

**urls.py**

```
from django.urls import path, register_converter
from . import converters, views
register_converter(converters.FourDigitYearConverter, 'yyyy')
urlpatterns = [
    path('session/<yyyy:year>', views.show_details, name="detail")
]
```

# 39-Messages Framework

The messages framework allows you to temporarily store messages in one request and retrieve them for display in a subsequent request.

Django provides full support for cookie- and session-based messaging, for both anonymous and authenticated users.

```
INSTALLED_APPS = [ 'django.contrib.messages' ]
```

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

```
'context_processors': ['django.contrib.messages.context_processors.messages']
```

# Messages Levels and Tags

The messages framework is based on a configurable level architecture similar to that of the Python logging module.

**Message Level** – Message levels allow you to group messages by type so they can be filtered or displayed differently in views and templates.

**Message Tag** – Message tags are a string representation of the message level plus any extra tags that were added directly in the view. Tags are stored in a string and are separated by spaces. Typically, message tags are used as CSS classes to customize message style based on message type.

Level	Tag	Value	Purpose
DEBUG	debug	10	Development related messages that will be ignored or removed in a production deployment
INFO	info	20	Informational messages for the user
SUCCESS	success	25	An action was successful, e.g. “Updated successfully”
WARNING	warning	30	A failure did not occur but may be imminent
ERROR	error	40	An action was not successful or some other failure occurred

# How to use

## Write Message:-

`add_message(request, level, message, extra_tags='', fail_silently=False)` – This method is used to add/write messages.

Setting `fail_silently=True` only hides the `MessageFailure` that would otherwise occur when the messages framework disabled and one attempts to use one of the `add_message` family of methods. It does not hide failures that may occur for other reasons.

```
from django.contrib import messages  
messages.add_message(request, messages.INFO, 'Info le lo info')
```

# How to use

## Write Message by Shortcut Methods:-

```
from django.contrib import messages  
messages.debug(request, '%s SQL statements were executed.' % count)  
messages.info(request, 'Three credits remain in your account.')  
messages.success(request, 'Profile details updated.')  
messages.warning(request, 'Your account expires in three days.')  
messages.error(request, 'Document deleted.')
```

# How to use

## Display Message:-

```
{% if messages %}  
    {% for message in messages %}  
        {% if message.tags %} {{ message.tags }} {% endif %}  
        {{ message }}  
    {% endfor %}  
{% endif %}
```

# Methods

get\_level ( ) - This method is used to retrieved the current effective level.

```
from django.contrib import messages
```

```
current_level = messages.get_level(request)
```

set\_level ( ) - This method is used to set minimum recorded level.

```
from django.contrib import messages
```

```
messages.set_level(request, messages.DEBUG)
```

This will record messages with a level of DEBUG and higher.

## Display Message:-

```
{% if messages %}  
    {% for message in messages %}  
        <p {% if message.tags %} class="alert-{{message.tags}}"{% endif %}>  
            {{message}}  
        </p>  
    {% endfor %}  
{% endif %}
```

# Change Tags

Open **settings.py**

```
from django.contrib.messages import constants as messages
MESSAGE_TAGS = {
    messages.ERROR: 'danger',
}
```

# **get\_messages(request)**

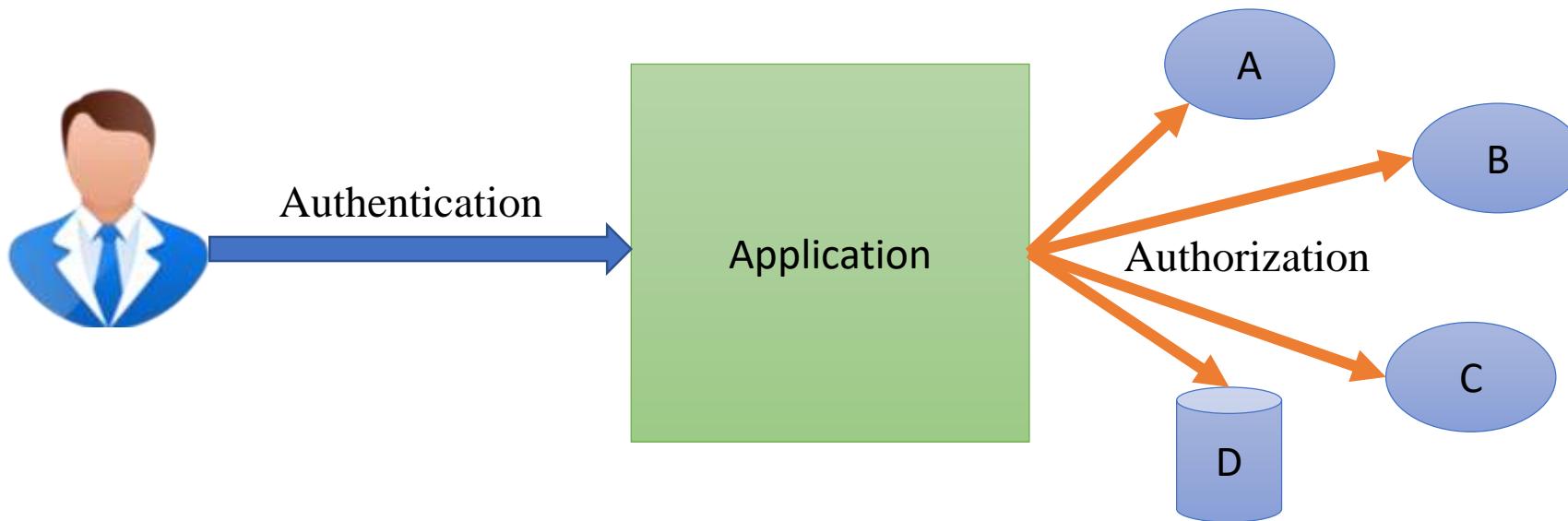
get\_messages(request) – This method is used to get message. If you need to get message out side template you can use get\_messages(request).

```
from django.contrib.messages import get_messages  
storage = get_messages(request)  
for message in storage:  
    do_something_with_the_message(message)
```

# 40-Authentication and Authorization

Authentication – Authentication is about validating your credentials like Username and password to verify your identity.

Authorization – Authorization is the process to determine whether the authenticated user has access to the particular resources. It checks your rights to grant you access to resources such as information, databases, files, etc.



# User Authentication System

Django comes with a user authentication system. It handles user accounts, groups, permissions and cookie-based user sessions.

Django authentication provides both authentication and authorization together and is generally referred to as the authentication system.

By default, the required configuration is already included in the `settings.py` generated by `django-admin startproject`, these consist of two items listed in your `INSTALLED_APPS` setting:

`'django.contrib.auth'` contains the core of the authentication framework, and its default models.

`'django.contrib.contenttypes'` is the Django content type system, which allows permissions to be associated with models you create.

and these items in your `MIDDLEWARE` setting:

`SessionMiddleware` manages sessions across requests.

`AuthenticationMiddleware` associates users with requests using sessions.

# django.contrib.auth

C:\Users\RK\AppData\Local\Programs\Python\Python38-32\Lib\site-packages\django\contrib\auth  
models.py

class User

User Object - User objects are the core of the authentication system.

They typically represent the people interacting with your site and are used to enable things like restricting access, registering user profiles, associating content with creators etc.

Only one class of user exists in Django's authentication framework, i.e., 'superusers' or admin 'staff' users are just user objects with special attributes set, not different classes of user objects.

- Creating Super Users
- Changing Password
- Authenticating User
- Creating Users
- Permissions and Authorization
- Groups
- How to log a user in
- How to log a user out

# User Authentication System

Django comes with a user authentication system. It handles user accounts, groups, permissions and cookie-based user sessions.

Django authentication provides both authentication and authorization together and is generally referred to as the authentication system.

By default, the required configuration is already included in the `settings.py` generated by `django-admin startproject`, these consist of two items listed in your `INSTALLED_APPS` setting:

`'django.contrib.auth'` contains the core of the authentication framework, and its default models.

`'django.contrib.contenttypes'` is the Django content type system, which allows permissions to be associated with models you create.

and these items in your `MIDDLEWARE` setting:

`SessionMiddleware` manages sessions across requests.

`AuthenticationMiddleware` associates users with requests using sessions.

# django.contrib.auth

C:\Users\RK\AppData\Local\Programs\Python\Python38-32\Lib\site-packages\django\contrib\auth  
models.py

class User

User Object - User objects are the core of the authentication system.

They typically represent the people interacting with your site and are used to enable things like restricting access, registering user profiles, associating content with creators etc.

Only one class of user exists in Django's authentication framework, i.e., 'superusers' or admin 'staff' users are just user objects with special attributes set, not different classes of user objects.

# User object Fields

username - Usernames may contain alphanumeric, \_, @, +, . and - characters. Its required and length is 150 characters or fewer.

first\_name – Its optional (blank=True) and length is 30 characters or fewer

last\_name – Its optional (blank=True) and length is 150 characters or fewer.

email – Its optional (blank=True)

password – A hash of, and metadata about, the password. Django doesn't store the raw password. Its required.

groups - Many-to-many relationship to Group.

# User object Fields

user\_permissions - Many-to-many relationship to Permission.

is\_staff - Designates whether this user can access the admin site. It takes Boolean.

is\_active - Designates whether this user account should be considered active. We recommend that you set this flag to False instead of deleting accounts; that way, if your applications have any foreign keys to users, the foreign keys won't break. It takes Boolean.

is\_superuser - Designates that this user has all permissions without explicitly assigning them. It takes Boolean.

last\_login - A datetime of the user's last login.

date\_joined - A datetime designating when the account was created. Is set to the current date/time by default when the account is created.

# User object Fields

`is_authenticated` - This is a way to tell if the user has been authenticated. This does not imply any permissions and doesn't check if the user is active or has a valid session. Its a read-only attribute which is always True.

`is_anonymous` - This is a way of differentiating User and AnonymousUser objects. It's a read-only attribute which is always False

# User object Methods

`get_username()` - Since the User model can be swapped out, you should use this method instead of referencing the `username` attribute directly. It returns the `username` for the user.

`get_full_name()` – It returns the `first_name` plus the `last_name`, with a space in between.

`get_short_name()` – It returns the `first_name`.

`set_password(raw_password)` – It sets the user’s password to the given raw string, taking care of the password hashing. Doesn’t save the User object.

When the `raw_password` is `None`, the password will be set to an unusable password, as if `set_unusable_password()` were used.

`check_password(raw_password)` – It returns `True` if the given raw string is the correct password for the user. (This takes care of the password hashing in making the comparison.)

# User object Methods

`set_unusable_password()` – It marks the user as having no password set. This isn't the same as having a blank string for a password. `check_password()` for this user will never return True. Doesn't save the User object.

You may need this if authentication for your application takes place against an existing external source such as an LDAP directory.

`has_usable_password()` – It returns False if `set_unusable_password()` has been called for this user.

`get_user_permissions(obj=None)` – It returns a set of permission strings that the user has directly.

If `obj` is passed in, only returns the user permissions for this specific object.

`get_group_permissions(obj=None)` – It returns a set of permission strings that the user has, through their groups.

If `obj` is passed in, only returns the group permissions for this specific object.

# User object Methods

`get_all_permissions(obj=None)` – It returns a set of permission strings that the user has, both through group and user permissions.

If obj is passed in, only returns the permissions for this specific object.

`has_perm(perm, obj=None)` – It returns True if the user has the specified permission, where perm is in the format "<app label>.<permission codename>". If the user is inactive, this method will always return False. For an active superuser, this method will always return True.

If obj is passed in, this method won't check for a permission for the model, but for this specific object.

`has_perms(perm_list, obj=None)` – It returns True if the user has each of the specified permissions, where each perm is in the format "<app label>.<permission codename>". If the user is inactive, this method will always return False. For an active superuser, this method will always return True.

If obj is passed in, this method won't check for permissions for the model, but for the specific object.

# User object Methods

`has_module_perms(package_name)` – It returns True if the user has any permissions in the given package (the Django app label). If the user is inactive, this method will always return False. For an active superuser, this method will always return True.

`email_user(subject, message, from_email=None, **kwargs)` – It sends an email to the user. If `from_email` is None, Django uses the `DEFAULT_FROM_EMAIL`. Any `**kwargs` are passed to the underlying `send_mail()` call.

# UserManager Methods

The User model has a custom manager that has the following helper methods (in addition to the methods provided by BaseUserManager):

`create_user(username, email=None, password=None, **extra_fields)` – It creates, saves and returns an User.

The username and password are set as given. The domain portion of email is automatically converted to lowercase, and the returned User object will have `is_active` set to True.

If no password is provided, `set_unusable_password()` will be called.

The `extra_fields` keyword arguments are passed through to the User's `__init__` method to allow setting arbitrary fields on a custom user model.

`create_superuser(username, email=None, password=None, **extra_fields)` – It is same as `create_user()`, but sets `is_staff` and `is_superuser` to True.

# UserManager Methods

`with_perm(perm, is_active=True, include_superuser=True, backend=None, obj=None)` – It returns users that have the given permission perm either in the "`<app label>.<permission codename>`" format or as a Permission instance. Returns an empty queryset if no users who have the perm found.

If `is_active` is True (default), returns only active users, or if False, returns only inactive users. Use None to return all users irrespective of active state.

If `include_superuser` is True (default), the result will include superusers.

If `backend` is passed in and it's defined in `AUTHENTICATION_BACKENDS`, then this method will use it. Otherwise, it will use the backend in `AUTHENTICATION_BACKENDS`, if there is only one, or raise an exception.

# **Group Object Fields**

name - Any characters are permitted. It is required and length is 150 characters or fewer.

Example: 'Awesome Users'.

permissions - Many-to-many field to Permission.

# Permission Object Fields

name – It is required and length is 255 characters or fewer. Example: 'Can vote'.

content\_type - A reference to the django\_content\_type database table, which contains a record for each installed model. It is required.

codename – It is required and length is 100 characters or fewer. Example: 'can\_vote'.

# **authenticate ()**

authenticate(request=None, \*\*credentials) – This is used to verify a set of credentials. It takes credentials as keyword arguments, username and password for the default case, checks them against each authentication backend, and returns a User object if the credentials are valid for a backend.

If the credentials aren't valid for any backend or if a backend raises PermissionDenied, it returns None.

*request* is an optional HttpRequest which is passed on the authenticate() method of the authentication backends.

Example:-

```
user = authenticate(username='sonam', password='geekyshows')
```

# login ()

login(request, user, backend=None) - To log a user in, from a view, use login(). It takes an HttpRequest object and a User object. login() saves the user's ID in the session, using Django's session framework.

When a user logs in, the user's ID and the backend that was used for authentication are saved in the user's session. This allows the same authentication backend to fetch the user's details on a future request.

# logout ()

logout(request) - To log out a user who has been logged in via django.contrib.auth.login(), use django.contrib.auth.logout() within your view. It takes an HttpRequest object and has no return value.

When you call logout(), the session data for the current request is completely cleaned out. All existing data is removed. This is to prevent another person from using the same Web browser to log in and have access to the previous user's session data.

# update\_session\_auth\_hash()

update\_session\_auth\_hash(request, user) - This function takes the current request and the updated user object from which the new session hash will be derived and updates the session hash appropriately. It also rotates the session key so that a stolen session cookie will be invalidated.

# **Permissions and Authorization**

Django comes with a built-in permissions system. It provides a way to assign permissions to specific users and groups of users.

The Django admin site uses permissions as follows:

- Access to view objects is limited to users with the “view” or “change” permission for that type of object.
- Access to view the “add” form and add an object is limited to users with the “add” permission for that type of object.
- Access to view the change list, view the “change” form and change an object is limited to users with the “change” permission for that type of object.
- Access to delete an object is limited to users with the “delete” permission for that type of object.

# **Permissions and Authorization**

myuser.groups.set([group\_list])

myuser.groups.add(group, group, ...)

myuser.groups.remove(group, group, ...)

myuser.groups.clear()

myuser.user\_permissions.set([permission\_list])

myuser.user\_permissions.add(permission, permission, ...)

myuser.user\_permissions.remove(permission, permission, ...)

myuser.user\_permissions.clear()

# **Permissions and Authorization**

When a model is created, Django will automatically create four default permissions for the following actions:

- add: Users with this permission can add an instance of the model.
- delete: Users with this permission can delete an instance of the model.
- change: Users with this permission can update an instance of the model.
- view: Users with this permission can view instances of this model.

Permission names follow a very specific naming convention: appname.action\_modelname

Example:- enroll.delete\_blog

# **perms Template Variable**

The currently logged-in user's permissions are stored in the template variable `{{ perms }}`. This is an instance of `django.contrib.auth.context_processors.PermWrapper`, which is a template-friendly proxy of permissions.

Example:-

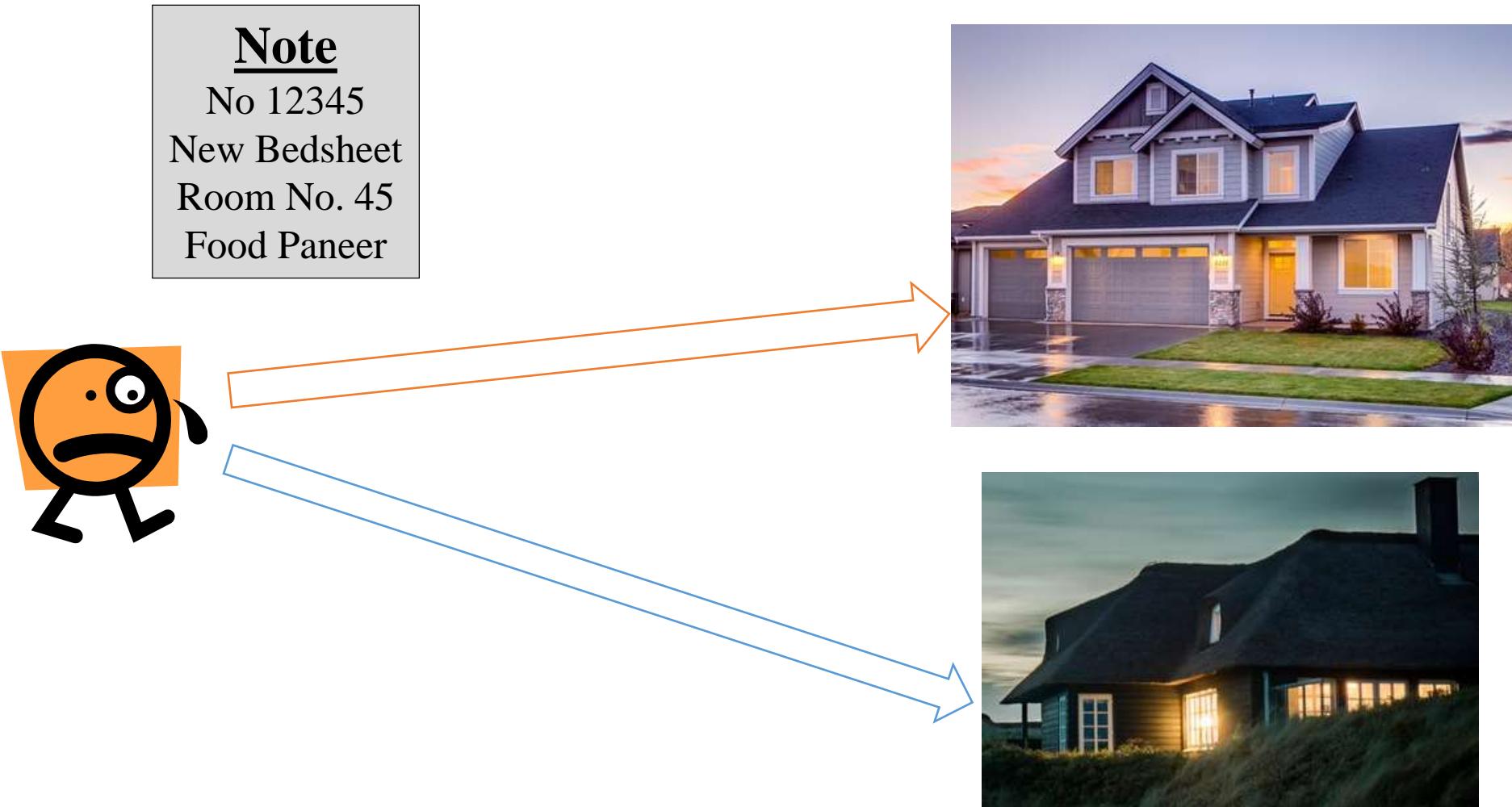
```
{% if perms.enroll.delete_blog %}  
<input type="button" value="Delete"><br><br>%  
{ % endif %}
```

```
{% if perms.enroll %}  
<input type="button" value="Delete"><br><br>%  
{ % endif %}
```

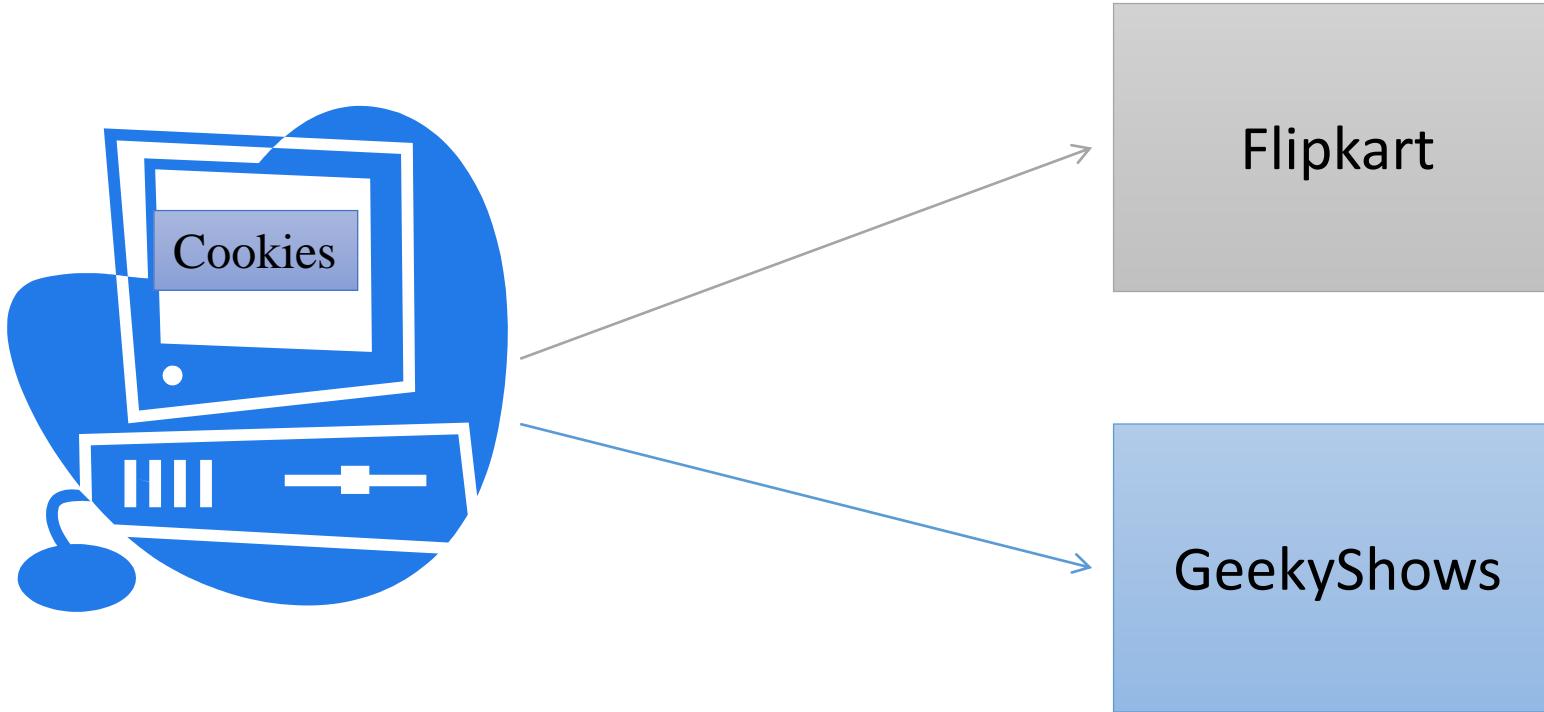
# 41-Cookies

A cookie is a small piece of text data set by Web server that resided on the client's machine. Once it's been set, the client automatically returns the cookie to the web server with each request that it makes. This allows the server to place value it wishes to 'remember' in the cookie, and have access to them when creating a response.

# How Cookie Works



# How Cookie Works



# Django Cookies

HttpRequest.COOKIES - A dictionary containing all cookies. Keys and values are strings.

# Creating Cookies

**set\_cookie()** – set\_cookie() is used to set/create/sent cookies.

Syntax: - `HttpServletResponse.set_cookie(key, value="", max_age=None, expires=None, path="/", domain=None, secure=False, httponly=False, samesite=None)`

Example: - `set_cookie(“name”, “sonam”)`

**key** – This is the name of the cookie.

**value** – This sets the value of cookie. This value is stored on the clients computer.

*name and value are required to set cookie.*

When you omit the optional cookie fields, the browser fills them in automatically with reasonable defaults.

# Creating Cookies

max\_age - It should be a number of seconds, or None (default) if the cookie should last only as long as the client's browser session. If expires is not specified, it will be calculated.

Example:- `set_cookie("name", "sonam", max_age=60*60*24*10)` // 10 days

expires - It describes the time when cookie will be expire. It should either be a string in the format "Wdy, DD-Mon-YY HH:MM:SS GMT" or a datetime.datetime object in UTC. If expires is a datetime object, the max\_age will be calculated.

Example:- `set_cookie("name", "sonam", expires= datetime.utcnow() + timedelta(days=2))`

path - Path can be / (root) or /mydir (directory).

Example: - `set_cookie("name", "sonam", "/")`

Example: - `set_cookie("name", "sonam", "/home")`

# Creating Cookies

domain - Use domain if you want to set a cross-domain cookie. For example, domain="example.com" will set a cookie that is readable by the domains www.example.com, blog.example.com, etc. Otherwise, a cookie will only be readable by the domain that set it.

Example: - `set_cookie ("name", "sonam", domain="geekyshows.com")`

`note.geekyshows.com`

`code.geekyshows.com`

Example: - `set_cookie ("name", "sonam", "code.geekyshows.com")`

secure - Cookie to only be transmitted over secure protocol as https. When set to TRUE , the cookie will only be set if a secure connection exists.

Example: - `set_cookie( "name", "sonam", max_age=60*60*24*10, path="/ ", domain="geekyshows.com", secure=True)`

# Creating Cookies

httponly - Use `httponly=True` if you want to prevent client-side JavaScript from having access to the cookie.

`HttpOnly` is a flag included in a `Set-Cookie` HTTP response header. It's part of the RFC 6265 standard for cookies and can be a useful way to mitigate the risk of a client-side script accessing the protected cookie data.

Example: - `set_cookie("name", "sonam", max_age=60*60*24*10, httponly=True)`

samesite - Use `samesite='Strict'` or `samesite='Lax'` to tell the browser not to send this cookie when performing a cross-origin request. `SameSite` isn't supported by all browsers, so it's not a replacement for Django's CSRF protection, but rather a defense in depth measure.

RFC 6265 states that user agents should support cookies of at least 4096 bytes. For many browsers this is also the maximum size. Django will not raise an exception if there's an attempt to store a cookie of more than 4096 bytes, but many browsers will not set the cookie correctly.

Example: - `set_cookie("name", "sonam", samesite='Strict')`

# Reading/Accessing Cookie

HttpRequest.COOKIES - A dictionary containing all cookies. Keys and values are strings.

Syntax:- request.COOKIES['key'];

Example: - request.COOKIES['name'];

Syntax:- request.COOKIES.get('key', default)

Example: - request.COOKIES.get('name')

Example: - request.COOKIES.get('name', "Guest")

# Replace/Append Cookies

When we assign a new value to cookie, the current cookie are not replaced. The new cookie is parsed and its name-value pair is appended to the list. The exception is when you assign a new cookie with the same name (and same domain and path, if they exist) as a cookie that already exists. In this case the old value is replaced with the new.

Examples:-

- set\_cookie("name", "sonam")  
set\_cookie("name", "rahul")
- 
- Replace

- set\_cookie("name", "sonam")  
set\_cookie("lname", "jha")
- 
- Append

# Deleting Cookies

HttpResponse.delete\_cookie(key, path='/', domain=None) – This method is used to delete the cookie based on given key with same domain and path, if they were set, otherwise the cookie may not be deleted.

Example: - delete\_cookie('name')

# Creating Signed Cookies

HttpServletResponse.set\_signed\_cookie(key, value, salt='', max\_age=None, expires=None, path='/', domain=None, secure=False, httponly=False, samesite=None) –

It is similar to set\_cookie(), but cryptographic signing the cookie before setting it. Use in conjunction with HttpServletRequest.get\_signed\_cookie().

You can use the optional salt argument for added key strength, but you will need to remember to pass it to the corresponding HttpServletRequest.get\_signed\_cookie() call.

# Getting Signed Cookies

`HttpRequest.get_signed_cookie(key, default=RAISE_ERROR, salt='', max_age=None)` –

It returns a cookie value for a signed cookie, or raises a `django.core.signing.BadSignature` exception if the signature is no longer valid.

If you provide the `default` argument the exception will be suppressed and that default value will be returned instead.

The optional `salt` argument can be used to provide extra protection against brute force attacks on your secret key. If supplied, the `max_age` argument will be checked against the signed timestamp attached to the cookie value to ensure the cookie is not older than `max_age` seconds.

# Cookies Security Issues

- Can misuse Client Details
- Can track User
- Client Can Delete Cookies
- Client can Manipulate Cookies

# Cookies Limitation

- Each cookie can contain 4096 bytes Data
- Cookies can be stored in Browser and server
- It is sent with each request

# 42-Session

The session framework lets you store and retrieve arbitrary data on a per-site-visitor basis.

It stores data on the server side and abstracts the sending and receiving of cookies. Cookies contain a session ID not the data itself.

By default, Django stores sessions in your database.

As it stores sessions in database so it is mandatory to run makemigrations and migrate to use session. It will create required tables.

The Django sessions framework is entirely, and solely, cookie-based.

`django.contrib.sessions.middleware.SessionMiddleware`

`django.contrib.sessions`

# Session

database-backed sessions - If you want to use a database-backed session, you need to add 'django.contrib.sessions' to your INSTALLED\_APPS setting.

Once you have configured your installation, run manage.py migrate to install the single database table that stores session data.

file-based sessions - To use file-based sessions, set the SESSION\_ENGINE setting to "django.contrib.sessions.backends.file".

You might also want to set the SESSION\_FILE\_PATH setting (which defaults to output from tempfile.gettempdir(), most likely /tmp) to control where Django stores session files. Be sure to check that your Web server has permissions to read and write to this location.

cookie-based sessions - To use cookies-based sessions, set the SESSION\_ENGINE setting to "django.contrib.sessions.backends.signed\_cookies". The session data will be stored using Django's tools for cryptographic signing and the SECRET\_KEY setting.

# Session

cached sessions - For better performance, you may want to use a cache-based session backend. To store session data using Django's cache system, you'll first need to make sure you've configured your cache.

# Using sessions in views

When SessionMiddleware is activated, each HttpRequest object, the first argument to any Django view function will have a session attribute, which is a dictionary-like object.

You can read it and write to request.session at any point in your view. You can edit it multiple times.

## Set Item

```
request.session['key'] = 'value'
```

## Get Item

```
returned_value = request.session['key']
```

```
returned_value = request.session.get('key', default=None)
```

# Using sessions in views

## Delete Item

```
del request.session['key']
```

This raises `KeyError` if the given key isn't already in the session.

## Contains

```
'key' in request.session
```

# Session Methods

keys() method returns a view object that displays a list of all the keys in the dictionary.

Syntax:- dict.keys()

items() method returns the list with all dictionary keys with values.

Syntax:- dict.items()

clear() function is used to erase all the elements of list. After this operation, list becomes empty.

Syntax:- dict.clear()

setdefault() method returns the value of a key (if the key is in dictionary). If not, it inserts key with a value to the dictionary.

Syntax: dict.setdefault(key, default\_value)

# **Session Methods**

`flush()` – It deletes the current session data from the session and deletes the session cookie. This is used if you want to ensure that the previous session data can't be accessed again from the user's browser (for example, the `django.contrib.auth.logout()` function calls it).

# Session Methods

`get_session_cookie_age()` – It returns the age of session cookies, in seconds. Defaults to `SESSION_COOKIE_AGE`.

`set_expiry(value)` – It sets the expiration time for the session. You can pass a number of different values:

If value is an integer, the session will expire after that many seconds of inactivity. For example, calling `request.session.set_expiry(300)` would make the session expire in 5 minutes.

If value is a datetime or timedelta object, the session will expire at that specific date/time. Note that datetime and timedelta values are only serializable if you are using the PickleSerializer.

If value is 0, the user's session cookie will expire when the user's Web browser is closed.

If value is None, the session reverts to using the global session expiry policy.

Reading a session is not considered activity for expiration purposes. Session expiration is computed from the last time the session was modified.

# Session Methods

`get_expiry_age()` – It returns the number of seconds until this session expires. For sessions with no custom expiration (or those set to expire at browser close), this will equal `SESSION_COOKIE_AGE`.

This function accepts two optional keyword arguments:

`modification`: last modification of the session, as a datetime object. Defaults to the current time.

`expiry`: expiry information for the session, as a datetime object, an int (in seconds), or `None`. Defaults to the value stored in the session by `set_expiry()`, if there is one, or `None`.

`get_expiry_date()` – It returns the date this session will expire. For sessions with no custom expiration (or those set to expire at browser close), this will equal the date `SESSION_COOKIE_AGE` seconds from now.

This function accepts the same keyword arguments as `get_expiry_age()`.

# Session Methods

`get_expire_at_browser_close()` – It returns either True or False, depending on whether the user's session cookie will expire when the user's Web browser is closed.

`clear_expired()` – It removes expired sessions from the session store. This class method is called by `clearsessions`.

`cycle_key()` – It creates a new session key while retaining the current session data. `django.contrib.auth.login()` calls this method to mitigate against session fixation.

# Session Methods

`set_test_cookie()` – It sets a test cookie to determine whether the user’s browser supports cookies. Due to the way cookies work, you won’t be able to test this until the user’s next page request.

`test_cookie_worked()` – It returns either True or False, depending on whether the user’s browser accepted the test cookie. Due to the way cookies work, you’ll have to call `set_test_cookie()` on a previous, separate page request.

`delete_test_cookie()` – It deletes the test cookie. Use this to clean up after yourself.

# Session Settings

**SESSION\_CACHE\_ALIAS** - If you're using cache-based session storage, this selects the cache to use. Default: 'default'

**SESSION\_COOKIE\_AGE** - The age of session cookies, in seconds. Default: 1209600 (2 weeks, in seconds)

**SESSION\_COOKIE\_DOMAIN** - The domain to use for session cookies. Set this to a string such as "example.com" for cross-domain cookies, or use None for a standard domain cookie.

Be cautious when updating this setting on a production site. If you update this setting to enable cross-domain cookies on a site that previously used standard domain cookies, existing user cookies will be set to the old domain. This may result in them being unable to log in as long as these cookies persist. Default: None

# Session Settings

SESSION\_COOKIE\_HTTPONLY - Whether to use HttpOnly flag on the session cookie. If this is set to True, client-side JavaScript will not be able to access the session cookie.

HttpOnly is a flag included in a Set-Cookie HTTP response header. It's part of the RFC 6265 standard for cookies and can be a useful way to mitigate the risk of a client-side script accessing the protected cookie data.

This makes it less trivial for an attacker to escalate a cross-site scripting vulnerability into full hijacking of a user's session. There aren't many good reasons for turning this off. Your code shouldn't read session cookies from JavaScript. Default: True

SESSION\_COOKIE\_NAME - The name of the cookie to use for sessions. This can be whatever you want (as long as it's different from the other cookie names in your application). Default: 'sessionid'

# Session Settings

`SESSION_COOKIE_PATH` - The path set on the session cookie. This should either match the URL path of your Django installation or be parent of that path.

This is useful if you have multiple Django instances running under the same hostname. They can use different cookie paths, and each instance will only see its own session cookie. Default: '/'

# Session Settings

`SESSION_COOKIE_SAMESITE` - The value of the SameSite flag on the session cookie. This flag prevents the cookie from being sent in cross-site requests thus preventing CSRF attacks and making some methods of stealing session cookie impossible.

Possible values for the setting are:

'Strict': prevents the cookie from being sent by the browser to the target site in all cross-site browsing context, even when following a regular link.

For example, for a GitHub-like website this would mean that if a logged-in user follows a link to a private GitHub project posted on a corporate discussion forum or email, GitHub will not receive the session cookie and the user won't be able to access the project. A bank website, however, most likely doesn't want to allow any transactional pages to be linked from external sites so the 'Strict' flag would be appropriate.

'Lax' (default): provides a balance between security and usability for websites that want to maintain user's logged-in session after the user arrives from an external link.

In the GitHub scenario, the session cookie would be allowed when following a regular link from an external website and be blocked in CSRF-prone request methods (e.g. POST).

'None' (string): the session cookie will be sent with all same-site and cross-site requests.

False: disables the flag.

# Session Settings

SESSION\_COOKIE\_SECURE - Whether to use a secure cookie for the session cookie. If this is set to True, the cookie will be marked as “secure”, which means browsers may ensure that the cookie is only sent under an HTTPS connection.

Leaving this setting off isn’t a good idea because an attacker could capture an unencrypted session cookie with a packet sniffer and use the cookie to hijack the user’s session. Default: False

SESSION\_ENGINE – Controls where Django stores session data. Included engines are:

'django.contrib.sessions.backends.db'

'django.contrib.sessions.backends.file'

'django.contrib.sessions.backends.cache'

'django.contrib.sessions.backends.cached\_db'

'django.contrib.sessions.backends.signed\_cookies'

Default: 'django.contrib.sessions.backends.db'

# Session Settings

SESSION\_EXPIRE\_AT\_BROWSER\_CLOSE - Whether to expire the session when the user closes their browser. Default: False

SESSION\_FILE\_PATH - If you're using file-based session storage, this sets the directory in which Django will store session data. When the default value (None) is used, Django will use the standard temporary directory for the system. Default: None

SESSION\_SAVE\_EVERY\_REQUEST - Whether to save the session data on every request. If this is False (default), then the session data will only be saved if it has been modified that is, if any of its dictionary values have been assigned or deleted. Empty sessions won't be created, even if this setting is active. Default: False

# Session Settings

SESSION\_SERIALIZER - Full import path of a serializer class to use for serializing session data.  
Included serializers are:

'django.contrib.sessions.serializers.PickleSerializer'

'django.contrib.sessions.serializers.JSONSerializer'

Default: 'django.contrib.sessions.serializers.JSONSerializer'

Time Zone Link : [https://en.wikipedia.org/wiki/List\\_of\\_tz\\_database\\_time\\_zones](https://en.wikipedia.org/wiki/List_of_tz_database_time_zones)

## 43-Cache

A Cache is an information technology for the temporary storage (caching) of Web documents, such as Web pages, images, and other types of Web multimedia, to reduce server lag.

Caching is one of those methods which a website implements to become faster. It is cost efficient and saves CPU processing time.

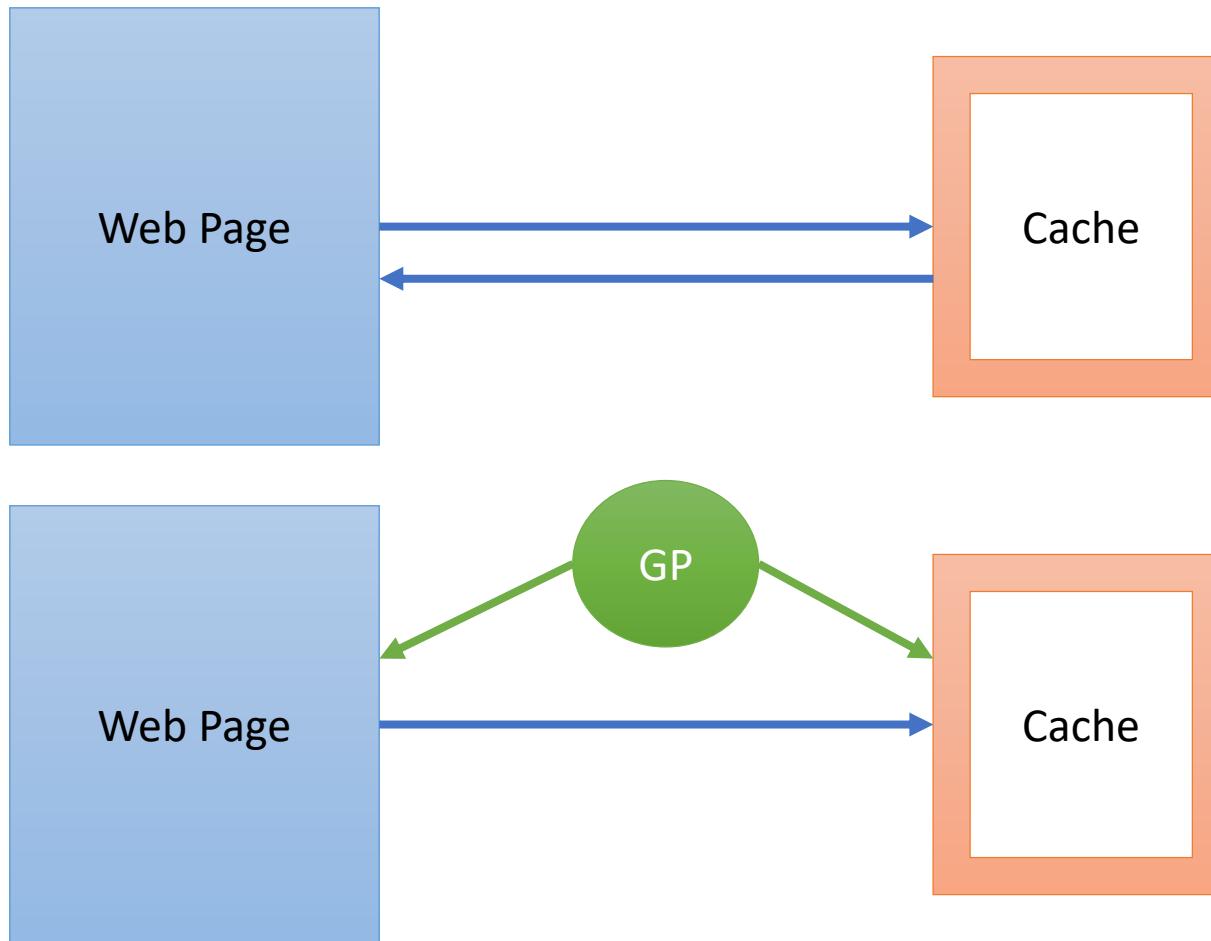
Django comes with a robust cache system that lets you save dynamic pages so they don't have to be calculated for each request.

You can cache the output of specific views, you can cache only the pieces that are difficult to produce, or you can cache your entire site.

Following are the options of caching:-

- Database Caching
- File System Caching
- Local Memory Caching

# How Cache Works



## **44-How to implement Caching**

- The per-site cache - Once the cache is set up, the simplest way to use caching is to cache your entire site.
- The per-view cache - A more granular way to use the caching framework is by caching the output of individual views.
- Template fragment caching – This gives you more control what to cache.

# The per-site cache

The per-site cache – Once the cache is set up, the simplest way to use caching is to cache your entire site.

MIDDLEWARE = [

```
'django.middleware.cache.UpdateCacheMiddleware',  
'django.middleware.common.CommonMiddleware',  
'django.middleware.cache.FetchFromCacheMiddleware',
```

]

CACHE\_MIDDLEWARE\_ALIAS – The cache alias to use for storage.

CACHE\_MIDDLEWARE\_SECONDS – The number of seconds each page should be cached.

CACHE\_MIDDLEWARE\_KEY\_PREFIX – If the cache is shared across multiple sites using the same Django installation, set this to the name of the site, or some other string that is unique to this Django instance, to prevent key collisions. Use an empty string if you don't care.

# Database Caching

Django can store its cached data in your database. This works best if you've got a fast, well-indexed database server.

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache',  
        'LOCATION': 'my_cache_table',  
    }  
}
```

The name of the database table. This name can be whatever you want, as long as it's a valid table name that's not already being used in your database.

Before using the database cache, you must create the cache table with this command:

***python manage.py createcachetable***

This creates a table in your database that is in the proper format that Django's database-cache system expects. The name of the table is taken from LOCATION.

If you are using multiple database caches, createcachetable creates one table for each cache.

# Cache Arguments

Each cache backend can be given additional arguments to control caching behavior.

**TIMEOUT:** The default timeout, in seconds, to use for the cache. This argument defaults to 300 seconds (5 minutes). You can set **TIMEOUT** to `None` so that, by default, cache keys never expire. A value of 0 causes keys to immediately expire (effectively “don’t cache”).

**OPTIONS:** Any options that should be passed to the cache backend. The list of valid options will vary with each backend, and cache backends backed by a third-party library will pass their options directly to the underlying cache library.

Cache backends that implement their own culling strategy (i.e., the locmem, filesystem and database backends) will honor the following options:

**MAX\_ENTRIES:** The maximum number of entries allowed in the cache before old values are deleted. This argument defaults to 300.

**CULL\_FREQUENCY:** The fraction of entries that are culled when **MAX\_ENTRIES** is reached. The actual ratio is  $1 / \text{CULL\_FREQUENCY}$ , so set **CULL\_FREQUENCY** to 2 to cull half the entries when **MAX\_ENTRIES** is reached. This argument should be an integer and defaults to 3.

A value of 0 for **CULL\_FREQUENCY** means that the entire cache will be dumped when **MAX\_ENTRIES** is reached. On some backends (database in particular) this makes culling much faster at the expense of more cache misses.

# Cache Arguments

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache',  
        'LOCATION': 'enroll_cache',  
        'TIMEOUT': 60,  
        'OPTIONS': {  
            'MAX_ENTRIES': 1000  
        }  
    }  
}
```

# Filesystem Caching

The file-based backend serializes and stores each cache value as a separate file.

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',  
        'LOCATION': 'c:/Django code/gs80',  
    }  
}
```

Absolute directory path where all cache file will be stored.

Make sure the directory pointed-to by this setting exists and is readable and writable by the system user under which your Web server runs.

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',  
        'LOCATION': '/var/tmp/django_cache',  
    }  
}
```

# Local Memory Caching

This is the default cache if another is not specified in your settings file. This cache is per-process and thread-safe.

Each process will have its own private cache instance, which means no cross-process caching is possible. This obviously also means the local memory cache isn't particularly memory-efficient.

It's probably not a good choice for production environments. It's nice for development.

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',  
        'LOCATION': 'unique-snowflake',  
    },  
}
```

The cache LOCATION is used to identify individual memory stores.

# Dummy Caching

Django comes with a “dummy” cache that doesn’t actually cache – it just implements the cache interface without doing anything.

This is useful if you have a production site that uses heavy-duty caching in various places but a development/test environment where you don’t want to cache and don’t want to have to change your code to special-case the latter.

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.dummy.DummyCache',  
    },  
}
```

# How to implement Caching

- The per-site cache - Once the cache is set up, the simplest way to use caching is to cache your entire site.
- The per-view cache - A more granular way to use the caching framework is by caching the output of individual views.
- Template fragment caching – This gives you more control what to cache.

# The per-view cache

The per-view cache - A more granular way to use the caching framework is by caching the output of individual views. `django.views.decorators.cache` defines a `cache_page` decorator that will automatically cache the view's response. If multiple URLs point at the same view, each URL will be cached separately.

```
from django.views.decorators.cache import cache_page  
  
@cache_page(timeout, cache, key_prefix)  
def my_view(request):  
  
    @cache_page(60, cache="some_cache", key_prefix="some_key")  
    def home(request):
```

`timeout` - The cache timeout, in seconds.

`cache` – This directs the decorator to use a specific cache (from your `CACHES` setting) when caching view results. By default, the default cache will be used.

`key_prefix` - You can also override the cache prefix on a per-view basis. It works in the same way as the `CACHE_MIDDLEWARE_KEY_PREFIX` setting for the middleware.

# The per-view cache

## Specifying per-view cache in the URLconf

```
from django.views.decorators.cache import cache_page  
urlpatterns = [  
    path('route/', cache_page(timeout, cache, key_prefix)(view_function)),
```

```
]
```

```
urlpatterns = [  
    path('home/', cache_page(60)(views.home), name="home"),
```

```
]
```

# Database Caching

Django can store its cached data in your database. This works best if you've got a fast, well-indexed database server.

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache',  
        'LOCATION': 'my_cache_table',  
    }  
}
```

The name of the database table. This name can be whatever you want, as long as it's a valid table name that's not already being used in your database.

Before using the database cache, you must create the cache table with this command:

***python manage.py createcachetable***

This creates a table in your database that is in the proper format that Django's database-cache system expects. The name of the table is taken from LOCATION.

If you are using multiple database caches, createcachetable creates one table for each cache.

# Cache Arguments

Each cache backend can be given additional arguments to control caching behavior.

**TIMEOUT:** The default timeout, in seconds, to use for the cache. This argument defaults to 300 seconds (5 minutes). You can set **TIMEOUT** to `None` so that, by default, cache keys never expire. A value of 0 causes keys to immediately expire (effectively “don’t cache”).

**OPTIONS:** Any options that should be passed to the cache backend. The list of valid options will vary with each backend, and cache backends backed by a third-party library will pass their options directly to the underlying cache library.

Cache backends that implement their own culling strategy (i.e., the locmem, filesystem and database backends) will honor the following options:

**MAX\_ENTRIES:** The maximum number of entries allowed in the cache before old values are deleted. This argument defaults to 300.

**CULL\_FREQUENCY:** The fraction of entries that are culled when **MAX\_ENTRIES** is reached. The actual ratio is  $1 / \text{CULL\_FREQUENCY}$ , so set **CULL\_FREQUENCY** to 2 to cull half the entries when **MAX\_ENTRIES** is reached. This argument should be an integer and defaults to 3.

A value of 0 for **CULL\_FREQUENCY** means that the entire cache will be dumped when **MAX\_ENTRIES** is reached. On some backends (database in particular) this makes culling much faster at the expense of more cache misses.

# Cache Arguments

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',  
        'LOCATION': 'c:/Django code/gs80',  
        'TIMEOUT': 60,  
        'OPTIONS': {  
            'MAX_ENTRIES': 1000  
        }  
    }  
}
```

# Filesystem Caching

The file-based backend serializes and stores each cache value as a separate file.

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',  
        'LOCATION': 'c:/Djangocode/gs80',  
    }  
}
```

Absolute directory path where all cache file will be stored.

Make sure the directory pointed-to by this setting exists and is readable and writable by the system user under which your Web server runs.

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',  
        'LOCATION': '/var/tmp/django_cache',  
    }  
}
```

# Local Memory Caching

This is the default cache if another is not specified in your settings file. This cache is per-process and thread-safe.

Each process will have its own private cache instance, which means no cross-process caching is possible. This obviously also means the local memory cache isn't particularly memory-efficient.

It's probably not a good choice for production environments. It's nice for development.

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',  
        'LOCATION': 'unique-snowflake',  
    },  
}
```

The cache LOCATION is used to identify individual memory stores.

# Dummy Caching

Django comes with a “dummy” cache that doesn’t actually cache – it just implements the cache interface without doing anything.

This is useful if you have a production site that uses heavy-duty caching in various places but a development/test environment where you don’t want to cache and don’t want to have to change your code to special-case the latter.

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.dummy.DummyCache',  
    },  
}
```

## **45-How to implement Caching**

- The per-site cache - Once the cache is set up, the simplest way to use caching is to cache your entire site.
- The per-view cache - A more granular way to use the caching framework is by caching the output of individual views.
- Template fragment caching – This gives you more control what to cache.

# Template Fragment Caching

This gives you more control what to cache.

{% load cache %} – This gives access to cache tag in template.

{% cache %} - This template tag caches the contents of the block for a given amount of time.

Syntax:-

```
{% cache timeout name variable using="" %} ..... {% endcache name %}
```

Timeout - The cache timeout, in seconds. The fragment is cached forever if timeout is None. It can be a template variable, as long as the template variable resolves to an integer value.

Name - The name to give the cache fragment. The name will be taken as is, do not use a variable.

Variable – This can be a variable with or without filters. This will cache multiple copies of a fragment depending on some dynamic data that appears inside the fragment.

using - The cache tag will try to use the given cache. If no such cache exists, it will fall back to using the default cache. You may select an alternate cache backend to use with the using keyword argument, which must be the last argument to the tag.

# Template Fragment Caching

Example:-

```
{% load cache %}  
{% cache 60 sidebar request.user.username using="localcache" %}  
.. sidebar for logged in user ..  
{% endcache %}
```

# Database Caching

Django can store its cached data in your database. This works best if you've got a fast, well-indexed database server.

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache',  
        'LOCATION': 'my_cache_table',  
    }  
}
```

The name of the database table. This name can be whatever you want, as long as it's a valid table name that's not already being used in your database.

Before using the database cache, you must create the cache table with this command:

***python manage.py createcachetable***

This creates a table in your database that is in the proper format that Django's database-cache system expects. The name of the table is taken from LOCATION.

If you are using multiple database caches, createcachetable creates one table for each cache.

# Cache Arguments

Each cache backend can be given additional arguments to control caching behavior.

**TIMEOUT:** The default timeout, in seconds, to use for the cache. This argument defaults to 300 seconds (5 minutes). You can set **TIMEOUT** to `None` so that, by default, cache keys never expire. A value of 0 causes keys to immediately expire (effectively “don’t cache”).

**OPTIONS:** Any options that should be passed to the cache backend. The list of valid options will vary with each backend, and cache backends backed by a third-party library will pass their options directly to the underlying cache library.

Cache backends that implement their own culling strategy (i.e., the locmem, filesystem and database backends) will honor the following options:

**MAX\_ENTRIES:** The maximum number of entries allowed in the cache before old values are deleted. This argument defaults to 300.

**CULL\_FREQUENCY:** The fraction of entries that are culled when **MAX\_ENTRIES** is reached. The actual ratio is  $1 / \text{CULL\_FREQUENCY}$ , so set **CULL\_FREQUENCY** to 2 to cull half the entries when **MAX\_ENTRIES** is reached. This argument should be an integer and defaults to 3.

A value of 0 for **CULL\_FREQUENCY** means that the entire cache will be dumped when **MAX\_ENTRIES** is reached. On some backends (database in particular) this makes culling much faster at the expense of more cache misses.

# Cache Arguments

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',  
        'LOCATION': 'c:\Djangocode\gs85\cache',  
        'TIMEOUT': 60,  
        'OPTIONS': {  
            'MAX_ENTRIES': 1000  
        }  
    }  
}
```

# Filesystem Caching

The file-based backend serializes and stores each cache value as a separate file.

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',  
        'LOCATION': 'c:\Djangocode\gs85\cache',  
    }  
}
```

Absolute directory path where all cache file will be stored.

Make sure the directory pointed-to by this setting exists and is readable and writable by the system user under which your Web server runs.

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',  
        'LOCATION': '/var/tmp/django_cache',  
    }  
}
```

# Local Memory Caching

This is the default cache if another is not specified in your settings file. This cache is per-process and thread-safe.

Each process will have its own private cache instance, which means no cross-process caching is possible. This obviously also means the local memory cache isn't particularly memory-efficient.

It's probably not a good choice for production environments. It's nice for development.

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',  
        'LOCATION': 'unique-snowflake',  
    },  
}
```

The cache LOCATION is used to identify individual memory stores.

# Dummy Caching

Django comes with a “dummy” cache that doesn’t actually cache – it just implements the cache interface without doing anything.

This is useful if you have a production site that uses heavy-duty caching in various places but a development/test environment where you don’t want to cache and don’t want to have to change your code to special-case the latter.

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.dummy.DummyCache',  
    },  
}
```

# **46-Low-Level Cache API**

Sometimes, caching an entire rendered page doesn't gain you very much and is, in fact, inconvenient overkill.

Perhaps, for instance, your site includes a view whose results depend on several expensive queries, the results of which change at different intervals. In this case, it would not be ideal to use the full-page caching that the per-site or per-view cache strategies offer, because you wouldn't want to cache the entire result (since some of the data changes often), but you'd still want to cache the results that rarely change.

For cases like this, Django exposes a low-level cache API. You can use this API to store objects in the cache with any level of granularity you like. You can cache any Python object that can be pickled safely: strings, dictionaries, lists of model objects, and so forth.

`django.core.cache.cache`

# Low-Level Cache API

```
from django.core.cache import cache
```

cache.set(key, value, timeout=DEFAULT\_TIMEOUT, version=None) – This method is used to set cache.

Where,

key – It should be str.

value – It can be any pickleable Python object.

timeout – It is number of seconds the value should be stored in the cache. Timeout of None will cache the value forever. A timeout of 0 won't cache the value.

version – It is an int. You can set cache with same key but different version.

cache.get(key, default=None, version=None) – This method is used to get cache. If the key doesn't exists in the cache, it returns None.

Where,

default – This specifies which value to return if the object doesn't exist in the cache.

# Low-Level Cache API

```
from django.core.cache import cache
```

`cache.add(key, value, timeout=DEFAULT_TIMEOUT, version=None)` – This method is used to add a key only if it doesn't already exist. It takes the same parameters as `set()`, but it will not attempt to update the cache if the key specified is already present. If you need to know whether `add()` stored a value in the cache, you can check the return value. It will return `True` if the value was stored, `False` otherwise.

`cache.get_or_set(key, default, timeout=DEFAULT_TIMEOUT, version=None)` – This method is used to get a key's value or set a value if the key isn't in the cache. It takes the same parameters as `get()` but the default is set as the new cache value for that key, rather than returned. You can also pass any callable as a default value.

`cache.set_many(dict, timeout)` – This method is used to set multiple values more efficiently, use `set_many()` to pass a dictionary of key-value pairs.

`cache.get_many(keys, version=None)` - There's also a `get_many()` interface that only hits the cache once. `get_many()` returns a dictionary with all the keys you asked for that actually exist in the cache (and haven't expired).

# Low-Level Cache API

```
from django.core.cache import cache
```

cache.delete(key, version=None) – This method is used to delete keys explicitly to clear the cache for a particular object.

cache.delete\_many(keys, version=None) – This method is used to clear a bunch of keys at once. It can take a list of keys to be cleared.

cache.clear() – This method is used to delete all the keys in the cache. Be careful with this; clear() will remove everything from the cache, not just the keys set by your application.

cache.touch(key, timeout=DEFAULT\_TIMEOUT, version=None) – This method is used to set a new expiration for a key. touch() returns True if the key was successfully touched, False otherwise.

# Low-Level Cache API

```
from django.core.cache import cache  
cache.incr(key, delta=1, version=None)  
cache.decr(key, delta=1, version=None)
```

You can also increment or decrement a key that already exists using the `incr()` or `decr()` methods, respectively. By default, the existing cache value will be incremented or decremented by 1. Other increment/decrement values can be specified by providing an argument to the increment/decrement call. A `ValueError` will be raised if you attempt to increment or decrement a nonexistent cache key.

# Low-Level Cache API

```
from django.core.cache import cache
```

cache.close() - You can close the connection to your cache with close() if implemented by the cache backend.

# 47-Signals

The signals are utilities that allow us to associate events with actions.

Signals allow certain senders to notify a set of receivers that some action has taken place.

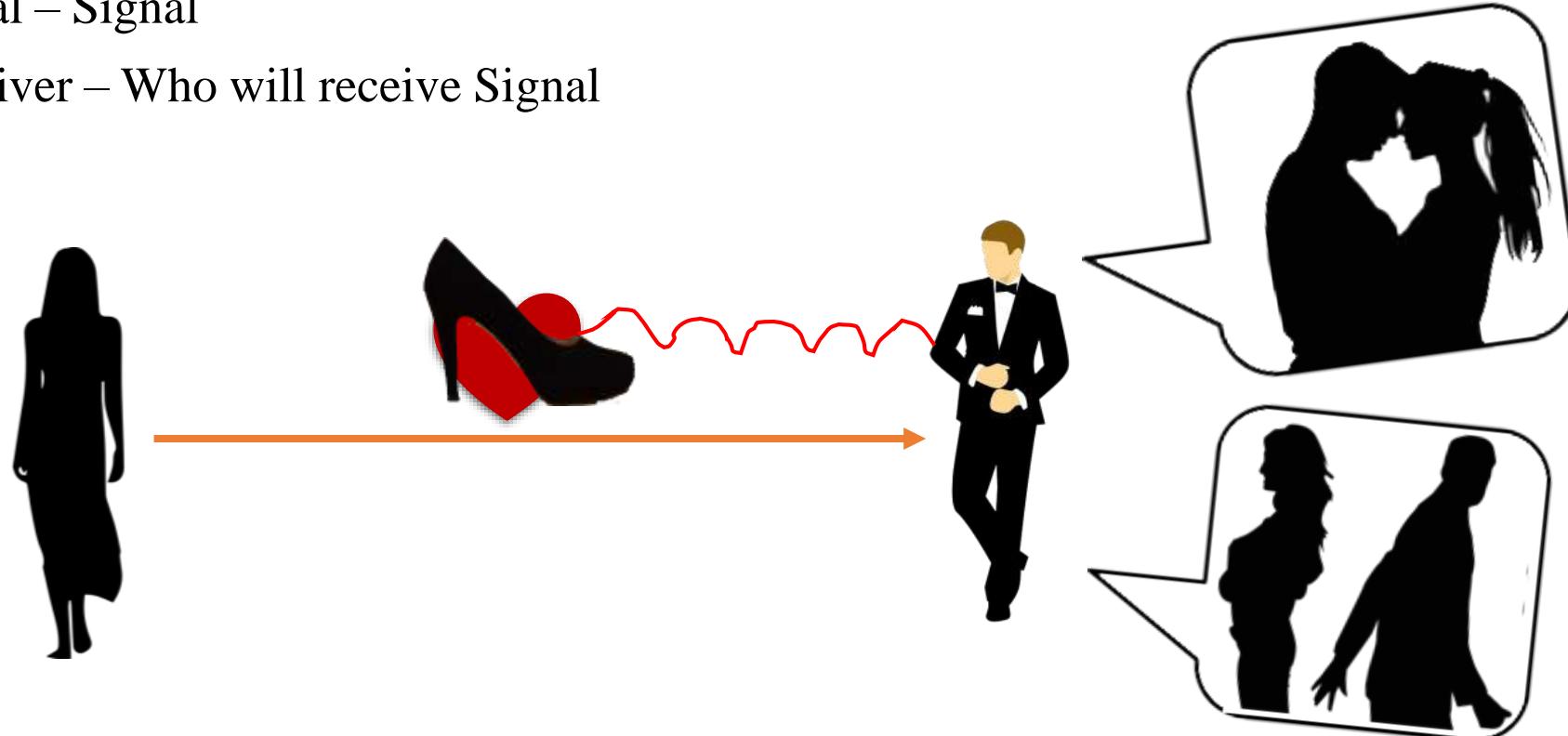
- Login and Logout Signals
- Model Signals
- Management Signals
- Request/Response Signals
- Test Signals
- Database Wrappers

# Signals

Sender – Who will send Signal

Signal – Signal

Receiver – Who will receive Signal



# Signals

Receiver Function - This function takes a sender argument, along with wildcard keyword arguments (\*\*kwargs); all signal handlers must take these arguments. A receiver can be any Python function or method.

```
def receiver_func(sender, request, user, **kwargs):  
    pass
```

Connecting/Registering Receiver Function - There are two ways you can connect a receiver to a signal:-

- Manual Connect Route
- Decorator

# Signals

Manual Connect Route - To receive a signal, register a receiver function using the Signal.connect() method. The receiver function is called when the signal is sent. All of the signal's receiver functions are called one at a time, in the order they were registered.

Signal.connect(receiver\_func, sender=None, weak=True, dispatch\_uid=None)

Where,

receiver\_func – The callback function which will be connected to signal.

sender – Specifies a particular sender to receive signals from.

weak – Django stores signal handlers as weak references by default. Thus, if your receiver is a local function, it may be garbage collected. To prevent this, pass weak=False when you call the signal's connect() method.

dispatch\_uid – A unique identifier for a signal receiver in cases where duplicate signals may be sent.

Decorator - @receiver(signal or list of signal, sender)

# Built-in Signals

Django provides a set of built-in signals that let user code get notified by Django itself of certain actions.

**Login and Logout Signals** - The auth framework uses the following signals that can be used for notification when a user logs in or out.

## **`django.contrib.auth.signals`**

`user_logged_in(sender, request, user)` - Sent when a user logs in successfully.

`sender` - The class of the user that just logged in.

`request` - The current HttpRequest instance.

`user` - The user instance that just logged in.

`user_logged_out(sender, request, user)` - Sent when the logout method is called.

`sender` - The class of the user that just logged out or `None` if the user was not authenticated.

`request` - The current HttpRequest instance.

`user` - The user instance that just logged out or `None` if the user was not authenticated.

# Built-in Signals

`user_login_failed(sender, credentials, request)` - Sent when the user failed to login successfully.

`sender` - The name of the module used for authentication.

`credentials` - A dictionary of keyword arguments containing the user credentials that were passed to `authenticate()` or your own custom authentication backend. Credentials matching a set of ‘sensitive’ patterns, (including password) will not be sent in the clear as part of the signal.

`request` - The `HttpRequest` object, if one was provided to `authenticate()`

# Built-in Signals

**Model signals** - A set of signals sent by the model system.

## **django.db.models.signals**

`pre_init` (`sender, args, kwargs`) - Whenever you instantiate a Django model, this signal is sent at the beginning of the model's `__init__()` method.

`sender` - The model class that just had an instance created.

`args` - A list of positional arguments passed to `__init__()`.

`kwargs` - A dictionary of keyword arguments passed to `__init__()`.

`post_init` (`sender, instance`) - Like `pre_init`, but this one is sent when the `__init__()` method finishes.

`sender` - The model class that just had an instance created.

`instance` - The actual instance of the model that's just been created.

# Built-in Signals

`pre_save(sender, instance, raw, using, update_fields)` - This is sent at the beginning of a model's `save()` method.

`sender` - The model class.

`instance` - The actual instance being saved.

`raw` - A boolean; True if the model is saved exactly as presented (i.e. when loading a fixture). One should not query/modify other records in the database as the database might not be in a consistent state yet.

`using` - The database alias being used.

`update_fields` - The set of fields to update as passed to `Model.save()`, or `None` if `update_fields` wasn't passed to `save()`.

# Built-in Signals

`post_save(sender, instance, created, raw, using, update_fields)` - Like `pre_save`, but sent at the end of the `save()` method.

`sender` - The model class.

`instance` - The actual instance being saved.

`created` - A boolean; True if a new record was created.

`raw` - A boolean; True if the model is saved exactly as presented (i.e. when loading a fixture). One should not query/modify other records in the database as the database might not be in a consistent state yet.

`using` - The database alias being used.

`update_fields` - The set of fields to update as passed to `Model.save()`, or `None` if `update_fields` wasn't passed to `save()`.

# Built-in Signals

`pre_delete(sender, instance, using)` - Sent at the beginning of a model's `delete()` method and a queryset's `delete()` method.

`sender` - The model class.

`instance` - The actual instance being deleted.

`using` - The database alias being used.

`post_delete(sender, instance, using)` - Like `pre_delete`, but sent at the end of a model's `delete()` method and a queryset's `delete()` method.

`sender` - The model class.

`instance` - The actual instance being deleted.

Note that the object will no longer be in the database, so be very careful what you do with this instance.

`using` - The database alias being used.

# Built-in Signals

m2m\_changed(sender, instance, action, reverse, model, pk\_set, using) - Sent when a ManyToManyField is changed on a model instance. Strictly speaking, this is not a model signal since it is sent by the ManyToManyField, but since it complements the pre\_save/post\_save and pre\_delete/post\_delete when it comes to tracking changes to models, it is included here.

sender - The intermediate model class describing the ManyToManyField. This class is automatically created when a many-to-many field is defined; you can access it using the through attribute on the many-to-many field.

instance - The instance whose many-to-many relation is updated. This can be an instance of the sender, or of the class the ManyToManyField is related to.

action - A string indicating the type of update that is done on the relation. This can be one of the following:

"pre\_add" - Sent before one or more objects are added to the relation.

"post\_add" - Sent after one or more objects are added to the relation.

"pre\_remove" - Sent before one or more objects are removed from the relation.

"post\_remove" - Sent after one or more objects are removed from the relation.

"pre\_clear" - Sent before the relation is cleared.

"post\_clear" - Sent after the relation is cleared.

# Built-in Signals

reverse - Indicates which side of the relation is updated (i.e., if it is the forward or reverse relation that is being modified).

model - The class of the objects that are added to, removed from or cleared from the relation.

pk\_set - For the pre\_add and post\_add actions, this is a set of primary key values that will be, or have been, added to the relation. This may be a subset of the values submitted to be added, since inserts must filter existing values in order to avoid a database IntegrityError.

For the pre\_remove and post\_remove actions, this is a set of primary key values that was submitted to be removed from the relation. This is not dependent on whether the values actually will be, or have been, removed. In particular, non-existent values may be submitted, and will appear in pk\_set, even though they have no effect on the database.

For the pre\_clear and post\_clear actions, this is None.

using - The database alias being used.

# Built-in Signals

`class_prepared(sender)` - Sent whenever a model class has been “prepared” – that is, once model has been defined and registered with Django’s model system. Django uses this signal internally; it’s not generally used in third-party applications.

Since this signal is sent during the app registry population process, and  `AppConfig.ready()` runs after the app registry is fully populated, receivers cannot be connected in that method. One possibility is to connect them  `AppConfig.__init__()` instead, taking care not to import models or trigger calls to the app registry.

`sender` - The model class which was just prepared.

# Built-in Signals

**Request/Response Signals** - Signals sent by the core framework when processing a request.

## **django.core.signals**

`request_started(sender, environ)` - Sent when Django begins processing an HTTP request.

sender - The handler class – e.g. `django.core.handlers.wsgi.WsgiHandler` – that handled the request.

environ - The `environ` dictionary provided to the request.

`request_finished(sender)` - Sent when Django finishes delivering an HTTP response to the client.

sender - The handler class.

`got_request_exception(sender, request)` - This signal is sent whenever Django encounters an exception while processing an incoming HTTP request.

sender - Unused (always `None`).

request - The `HttpRequest` object.

# Built-in Signals

Management signals – Signals sent by Django-admin

## **django.db.models.signals**

`pre_migrate(sender, app_config, verbosity, interactive, using, plan, apps)` - Sent by the `migrate` command before it starts to install an application. It's not emitted for applications that lack a `models` module.

`sender` - An  `AppConfig` instance for the application about to be migrated-synced.

`app_config` - Same as `sender`.

`verbosity` - Indicates how much information `manage.py` is printing on screen.

Functions which listen for `pre_migrate` should adjust what they output to the screen based on the value of this argument.

`interactive` - If `interactive` is `True`, it's safe to prompt the user to input things on the command line. If `interactive` is `False`, functions which listen for this signal should not try to prompt for anything.

For example, the `django.contrib.auth` app only prompts to create a superuser when `interactive` is `True`.

`using` - The alias of database on which a command will operate.

# Built-in Signals

plan - The migration plan that is going to be used for the migration run. While the plan is not public API, this allows for the rare cases when it is necessary to know the plan. A plan is a list of two-tuples with the first item being the instance of a migration class and the second item showing if the migration was rolled back (True) or applied (False).

apps - An instance of Apps containing the state of the project before the migration run. It should be used instead of the global apps registry to retrieve the models you want to perform operations on

# Built-in Signals

`post_migrate(sender, app_config, verbosity, interactive, using, plan, apps)` - Sent at the end of the `migrate` (even if no migrations are run) and `flush` commands. It's not emitted for applications that lack a `models` module.

Handlers of this signal must not perform database schema alterations as doing so may cause the `flush` command to fail if it runs during the `migrate` command.

`sender` - An  `AppConfig` instance for the application that was just installed.

`app_config` - Same as `sender`.

`verbosity` - Indicates how much information `manage.py` is printing on screen.

Functions which listen for `post_migrate` should adjust what they output to the screen based on the value of this argument.

`interactive` - If `interactive` is `True`, it's safe to prompt the user to input things on the command line. If `interactive` is `False`, functions which listen for this signal should not try to prompt for anything.

For example, the `django.contrib.auth` app only prompts to create a superuser when `interactive` is `True`.

# Built-in Signals

using - The database alias used for synchronization. Defaults to the default database.

plan - The migration plan that was used for the migration run. While the plan is not public API, this allows for the rare cases when it is necessary to know the plan. A plan is a list of two-tuples with the first item being the instance of a migration class and the second item showing if the migration was rolled back (True) or applied (False).

apps - An instance of Apps containing the state of the project after the migration run. It should be used instead of the global apps registry to retrieve the models you want to perform operations on.

# Built-in Signals

Test Signals - Signals only sent when running tests.

## `django.test.signals`

`setting_changed(sender, setting, value, enter)` - This signal is sent when the value of a setting is changed through the `django.test.TestCase.settings()` context manager or the `django.test.override_settings()` decorator/context manager.

It's actually sent twice: when the new value is applied ("setup") and when the original value is restored ("teardown"). Use the `enter` argument to distinguish between the two.

You can also import this signal from `django.core.signals` to avoid importing from `django.test` in non-test situations.

`sender` - The settings handler.

`setting` - The name of the setting.

`value` - The value of the setting after the change. For settings that initially don't exist, in the "teardown" phase, `value` is `None`.

`enter` - A boolean; `True` if the setting is applied, `False` if restored.

# Built-in Signals

template\_rendered(sender, template, context) - Sent when the test system renders a template. This signal is not emitted during normal operation of a Django server – it is only available during testing.

sender - The Template object which was rendered.

template - Same as sender

context - The Context with which the template was rendered.

# Built-in Signals

**Database Wrappers** - Signals sent by the database wrapper when a database connection is initiated.

## **django.db.backends.signals**

`connection_created` - Sent when the database wrapper makes the initial connection to the database. This is particularly useful if you'd like to send any post connection commands to the SQL backend.

`sender` - The database wrapper class – i.e. `django.db.backends.postgresql.DatabaseWrapper` or `django.db.backends.mysql.DatabaseWrapper`, etc.

`connection` - The database connection that was opened. This can be used in a multiple-database configuration to differentiate connection signals from different databases.

# 48-Defining Custom Signals

All signals are django.dispatch.Signal instances.

```
class Signal():
```

# Sending signals

There are two ways to send signals in Django.

- `Signal.send(sender, **kwargs)` – This is used to send a signal, all built-in signals use this to send signals. You must provide the `sender` argument which is a class most of the time and may provide as many other keyword arguments as you like. It returns a list of tuple pairs `[(receiver, response), ... ]`, representing the list of called receiver functions and their response values.
- `Signal.send_robust(sender, **kwargs)` – This is used to send a signal. You must provide the `sender` argument which is a class most of the time and may provide as many other keyword arguments as you like. It returns a list of tuple pairs `[(receiver, response), ... ]`, representing the list of called receiver functions and their response values.

# **Sending signals**

## **Difference between send () and send\_robust()**

- `send()` does not catch any exceptions raised by receivers; it simply allows errors to propagate. Thus not all receivers may be notified of a signal in the face of an error.
- `send_robust()` catches all errors derived from Python's `Exception` class, and ensures all receivers are notified of the signal. If an error occurs, the error instance is returned in the tuple pair for the receiver that raised the error.

# Disconnecting Signals

`Signal.disconnect(receiver=None, sender=None, dispatch_uid=None)` – This is used to disconnect a receiver from a signal. The arguments are as described in `Signal.connect()`. The method returns True if a receiver was disconnected and False if not.

The receiver argument indicates the registered receiver to disconnect. It may be None if `dispatch_uid` is used to identify the receiver.

# 49-Middleware

Middleware is a framework of hooks into Django's request/response processing.

It's a light, low-level "plugin" system for globally altering Django's input or output.  
Each middleware component is responsible for doing some specific function.

- Built in Middleware
- Custom Middleware

```
MIDDLEWARE = [  
    'django.middleware.security.SecurityMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    'django.middleware.csrf.CsrfViewMiddleware',  
    'django.contrib.auth.middleware.AuthenticationMiddleware',  
    'django.contrib.messages.middleware.MessageMiddleware',  
    'django.middleware.clickjacking.XFrameOptionsMiddleware',  
]
```

# How Middleware Works



# How Middleware Works



Banda



Bhai



Baap



Bandi



# How Middleware Works



Banda



Bhai



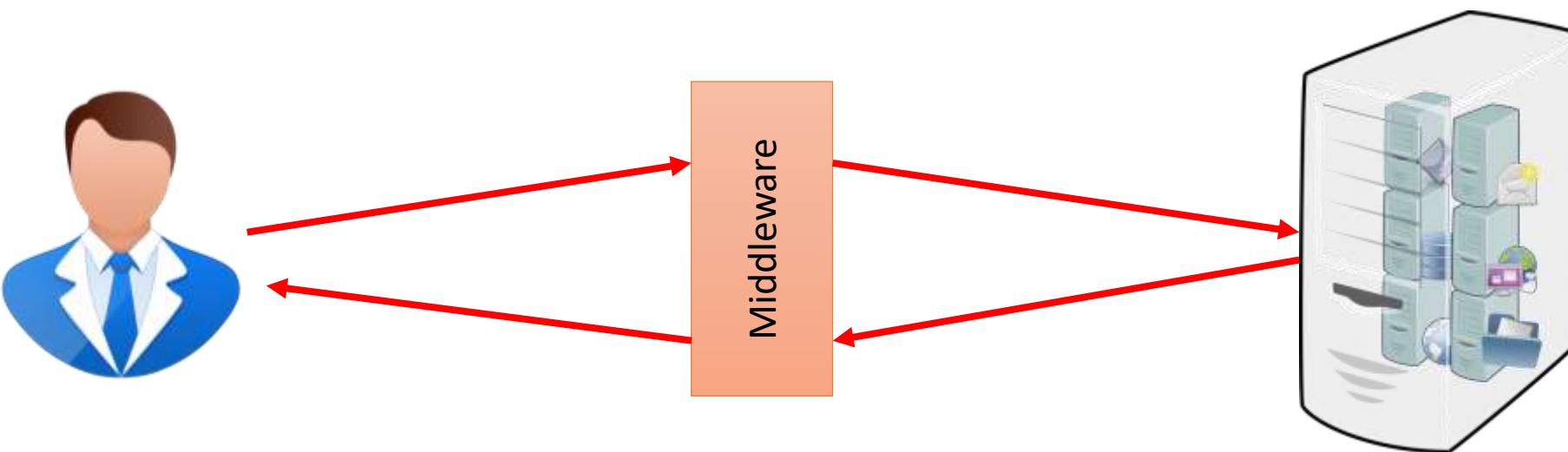
Baap



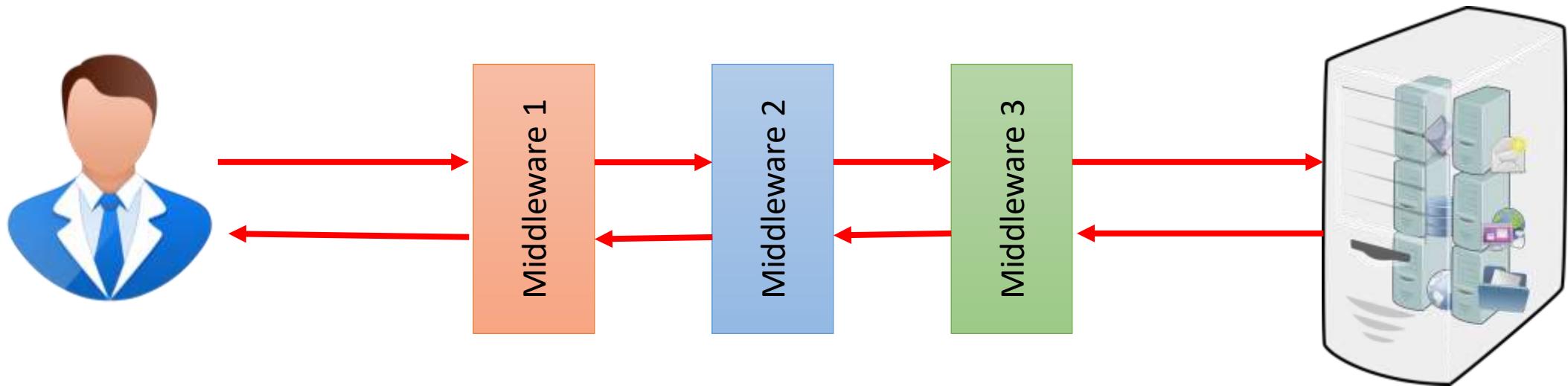
Bandi



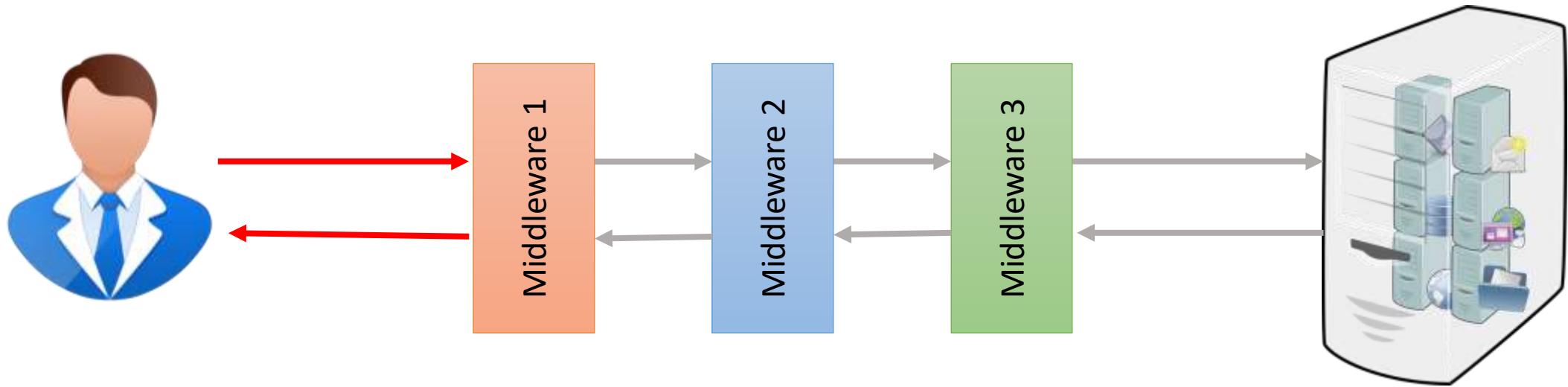
# How Middleware Works



# How Middleware Works



# How Middleware Works



# Function based Middleware

A middleware factory is a callable that takes a `get_response` callable and returns a middleware.

A middleware is a callable that takes a request and returns a response, just like a view.

```
def my_middleware(get_response):
    # One-time configuration and initialization.

    def my_function(request):
        # Code to be executed for each request before the view are called.

        response = get_response(request)

        # Code to be executed for each request/response after the view is called.

        return response

    return my_function
```

# **get\_response ( )**

The `get_response` callable provided by Django might be the actual view (if this is the last listed middleware) or it might be the next middleware in the chain.

The current middleware doesn't need to know or care what exactly it is, just that it represents whatever comes next.

The `get_response` callable for the last middleware in the chain won't be the actual view but rather a wrapper method from the handler which takes care of applying view middleware, calling the view with appropriate URL arguments, and applying template-response and exception middleware.

Middleware can live anywhere on your Python path.

# Activating Middleware

To activate a middleware component, add it to the MIDDLEWARE list in your Django settings.

In MIDDLEWARE, each middleware component is represented by a string: the full Python path to the middleware factory's class or function name. The order in MIDDLEWARE matters because a middleware can depend on other middleware. For instance, AuthenticationMiddleware stores the authenticated user in the session; therefore, it must run after SessionMiddleware.

Eg. -

```
MIDDLEWARE = [  
    'django.middleware.security.SecurityMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'blog.middlewares.my_middleware'  
]
```

# Class based Middleware

```
class MyMiddleware:  
    def __init__(self, get_response):  
        self.get_response = get_response  
        # One-time configuration and initialization.  
    def __call__(self, request):  
        # Code to be executed for each request before the view (and later middleware) are called.  
        response = self.get_response(request)  
        # Code to be executed for each request/response after the view is called.  
        return response
```

# `__init__(get_response)`

`__init__(get_response)` - Middleware factories must accept a `get_response` argument. You can also initialize some global state for the middleware. Keep in mind a couple of caveats:

- Django initializes your middleware with only the `get_response` argument, so you can't define `__init__()` as requiring any other arguments.
- Unlike the `__call__()` method which is called once per request, `__init__()` is called only once, when the Web server starts.

# Activating Middleware

To activate a middleware component, add it to the MIDDLEWARE list in your Django settings.

In MIDDLEWARE, each middleware component is represented by a string: the full Python path to the middleware factory's class or function name. The order in MIDDLEWARE matters because a middleware can depend on other middleware. For instance, AuthenticationMiddleware stores the authenticated user in the session; therefore, it must run after SessionMiddleware.

Eg. -

```
MIDDLEWARE = [  
    'django.middleware.security.SecurityMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'blog.middlewares.MyMiddleware'  
]
```

# Middleware Hooks

Following are special methods to class-based middleware:

`process_view(request, view_func, view_args, view_kwargs)` - It is called just before Django calls the view.

It should return either `None` or an `HttpResponse` object.

If it returns `None`, Django will continue processing this request, executing any other `process_view()` middleware and, then, the appropriate view.

If it returns an `HttpResponse` object, Django won't bother calling the appropriate view; it'll apply response middleware to that `HttpResponse` and return the result.

# Middleware Hooks

process\_view(request, view\_func, view\_args, view\_kwargs)

Where,

Request - It is an HttpRequest object.

view\_func – It is the Python function that Django is about to use. (It's the actual function object, not the name of the function as a string.)

view\_args – It is a list of positional arguments that will be passed to the view.

view\_kwargs – It is a dictionary of keyword arguments that will be passed to the view.

Neither view\_args nor view\_kwargs include the first view argument (request).

# Middleware Hooks

`process_exception(request, exception)` - Django calls `process_exception()` when a view raises an exception.

It should return either `None` or an `HttpResponse` object.

If it returns an `HttpResponse` object, the template response and response middleware will be applied and the resulting response returned to the browser. Otherwise, default exception handling kicks in.

Where,

`Request` – It is an `HttpRequest` object.

`Exception` – It is an `Exception` object raised by the view function.

Note - Middleware are run in reverse order during the response phase, which includes `process_exception`. If an exception middleware returns a response, the `process_exception` methods of the middleware classes above that middleware won't be called at all.

# Middleware Hooks

`process_template_response(request, response)` – This method is called just after the view has finished executing, if the response instance has a `render()` method, indicating that it is a `TemplateResponse` or equivalent.

It must return a response object that implements a `render` method.

It could alter the given response by changing `response.template_name` and `response.context_data`, or it could create and return a brand-new `TemplateResponse` or equivalent.

You don't need to explicitly render responses, responses will be automatically rendered once all template response middleware has been called.

Where,

`request` – It is an `HttpRequest` object.

`response` – It is the `TemplateResponse` object (or equivalent) returned by a Django view or by a middleware.

Note - Middleware are run in reverse order during the response phase, which includes `process_template_response()`.

# TemplateResponse

TemplateResponse - TemplateResponse is a subclass of SimpleTemplateResponse that knows about the current HttpRequest.

A TemplateResponse object can be used anywhere that a normal django.http.HttpResponse can be used. It can also be used as an alternative to calling render().

## Method

\_\_init\_\_(request, template, context=None, content\_type=None, status=None, charset=None, using=None) - It instantiates a TemplateResponse object with the given request, template, context, content type, HTTP status, and charset.

Where,

request - An HttpRequest instance.

template - A backend-dependent template object (such as those returned by get\_template()), the name of a template, or a list of template names.

# TemplateResponse

context - A dict of values to add to the template context. By default, this is an empty dictionary.

content\_type - The value included in the HTTP Content-Type header, including the MIME type specification and the character set encoding. If content\_type is specified, then its value is used. Otherwise, 'text/html' is used.

status - The HTTP status code for the response.

charset - The charset in which the response will be encoded. If not given it will be extracted from content\_type, and if that is unsuccessful, the DEFAULT\_CHARSET setting will be used.

using - The NAME of a template engine to use for loading the template.

# TemplateResponse

There are three circumstances under which a TemplateResponse will be rendered:

When the TemplateResponse instance is explicitly rendered, using the SimpleTemplateResponse.render() method.

When the content of the response is explicitly set by assigning response.content.

After passing through template response middleware, but before passing through response middleware.

Note –

A TemplateResponse can only be rendered once.

# Built-in Middleware

SecurityMiddleware - The `django.middleware.security.SecurityMiddleware` provides several security enhancements to the request/response cycle.

Each one can be independently enabled or disabled with a setting.

`SECURE_BROWSER_XSS_FILTER`

`SECURE_CONTENT_TYPE_NOSNIFF`

`SECURE_HSTS_INCLUDE_SUBDOMAINS`

`SECURE_HSTS_PRELOAD`

`SECURE_HSTS_SECONDS`

`SECURE_REDIRECT_EXEMPT`

`SECURE_REFERRER_POLICY`

`SECURE_SSL_HOST`

`SECURE_SSL_REDIRECT`

# Built-in Middleware

`SECURE_BROWSER_XSS_FILTER` - If True, the SecurityMiddleware sets the X-XSS-Protection: 1; mode=block header on all responses that do not already have it.

Modern browsers don't honor X-XSS-Protection HTTP header anymore. Although the setting offers little practical benefit, you may still want to set the header if you support older browsers. Default is False

`SECURE_CONTENT_TYPE_NOSNIFF` - If True, the SecurityMiddleware sets the X-Content-Type-Options: nosniff header on all responses that do not already have it. Default is True

`SECURE_HSTS_INCLUDE_SUBDOMAINS` - If True, the SecurityMiddleware adds the includeSubDomains directive to the HTTP Strict Transport Security header. It has no effect unless `SECURE_HSTS_SECONDS` is set to a non-zero value. Default is False

# Built-in Middleware

`SECURE_HSTS_PRELOAD` - If True, the SecurityMiddleware adds the preload directive to the HTTP Strict Transport Security header. It has no effect unless `SECURE_HSTS_SECONDS` is set to a non-zero value. Default is False

`SECURE_HSTS_SECONDS` – If set to a non-zero integer value, the SecurityMiddleware sets the HTTP Strict Transport Security header on all responses that do not already have it. Default is 0

`SECURE_REDIRECT_EXEMPT` – If a URL path matches a regular expression in this list, the request will not be redirected to HTTPS. The SecurityMiddleware strips leading slashes from URL paths, so patterns shouldn't include them, e.g. `SECURE_REDIRECT_EXEMPT = [r'^no-ssl/$', ...]`. If `SECURE_SSL_REDIRECT` is False, this setting has no effect. Default is [] empty list

`SECURE_REFERRER_POLICY` - If configured, the SecurityMiddleware sets the Referrer Policy header on all responses that do not already have it to the value provided. Default is None

# Built-in Middleware

`SECURE_SSL_HOST` - If a string (e.g. `secure.example.com`), all SSL redirects will be directed to this host rather than the originally-requested host (e.g. `www.example.com`). If `SECURE_SSL_REDIRECT` is `False`, this setting has no effect. Default is `None`

`SECURE_SSL_REDIRECT` - If `True`, the `SecurityMiddleware` redirects all non-HTTPS requests to HTTPS (except for those URLs matching a regular expression listed in `SECURE_REDIRECT_EXEMPT`). Default is `False`

# Built-in Middleware

CommonMiddleware - Adds a few conveniences for perfectionists:

Forbids access to user agents in the `DISALLOWED_USER_AGENTS` setting, which should be a list of compiled regular expression objects.

Performs URL rewriting based on the `APPEND_SLASH` and `PREPEND_WWW` settings.

If `APPEND_SLASH` is True and the initial URL doesn't end with a slash, and it is not found in the URLconf, then a new URL is formed by appending a slash at the end. If this new URL is found in the URLconf, then Django redirects the request to this new URL. Otherwise, the initial URL is processed as usual.

For example, `geekyshows.com/home` will be redirected to `geekyshows.com/home/` if you don't have a valid URL pattern for `geekyshows.com/home` but do have a valid pattern for `geekyshows.com/home/`.

If `PREPEND_WWW` is True, URLs that lack a leading "www." will be redirected to the same URL with a leading "www."

Both of these options are meant to normalize URLs. The philosophy is that each URL should exist in one, and only one, place. Technically a URL `geekyshows.com/home` is distinct from `geekyshows.com/home/` a search-engine indexer would treat them as separate URLs – so it's best practice to normalize URLs.

Sets the Content-Length header for non-streaming responses.

# Built-in Middleware

UpdateCacheMiddleware and FetchFromCacheMiddleware - These middleware belongs to cache middleware. It enables the site-wide cache. If these are enabled, each Django-powered page will be cached for as long as the CACHE\_MIDDLEWARE\_SECONDS setting defines.

MessageMiddleware - Enables cookie- and session-based message support.

SessionMiddleware - Enables session support.

AuthenticationMiddleware - It adds the user attribute, representing the currently-logged-in user, to every incoming HttpRequest object.

CsrfViewMiddleware - It adds protection against Cross Site Request Forgeries by adding hidden form fields to POST forms and checking requests for the correct value.

# Built-in Middleware

XFrameOptionsMiddleware - Simple clickjacking protection via the X-Frame-Options header.

FlatpageFallbackMiddleware - Should be near the bottom as it's a last-resort type of middleware.

RedirectFallbackMiddleware - Should be near the bottom as it's a last-resort type of middleware.

LocaleMiddleware - One of the topmost, after SessionMiddleware (uses session data) and UpdateCacheMiddleware (modifies Vary header).

ConditionalGetMiddleware - Before any middleware that may change the response (it sets the ETag header). After GZipMiddleware so it won't calculate an ETag header on gzipped contents.

# Built-in Middleware

GZipMiddleware - Before any middleware that may change or use the response body. After  
UpdateCacheMiddleware: Modifies Vary header.

# 50-QuerySet API

A QuerySet can be defined as a list containing all those objects we have created using the Django model.

QuerySets allow you to read the data from the database, filter it and order it.

query property – This property is used to get sql query of query set.

Syntax:- `queryset.query`

# Methods that return new QuerySets

## Retrieving all objects

all ( ) - This method is used to retrieve all objects. This returns a copy of current QuerySet.

Example:- Student.objects.all()

## Retrieving specific objects

- filter (\*\*kwargs) - It returns a new QuerySet containing objects that match the given *lookup parameters*. filter() will always give you a QuerySet, even if only a single object matches the query.

Example:- Student.objects.filter(marks=70)

- exclude(\*\*kwargs) - It returns a new QuerySet containing objects that do not match the given *lookup parameters*.

Example:- Student.objects.exclude(marks=70)

# Methods that return new QuerySets

order\_by(\*fields) – It orders the fields.

- ‘field’ – Asc order
- ‘-field’ – Desc Order
- ‘?’ – Randomly

reverse() – This works only when there is ordering in queryset.

values(\*fields, \*\*expressions) - It returns a QuerySet that returns dictionaries, rather than model instances, when used as an iterable. Each of those dictionaries represents an object, with the keys corresponding to the attribute names of model objects.

distinct(\*fields) - This eliminates duplicate rows from the query results.

# Methods that return new QuerySets

`values_list(*fields, flat=False, named=False)` - This is similar to `values()` except that instead of returning dictionaries, it returns tuples when iterated over.

- If you don't pass any values to `values_list()`, it will return all the fields in the model, in the order they were declared.
- If you only pass in a single field, you can also pass in the `flat` parameter. If `True`, this will mean the returned results are single values, rather than one-tuples.
- You can pass `named=True` to get results as a namedtuple.

`using(alias)` - This method is for controlling which database the QuerySet will be evaluated against if you are using more than one database. The only argument this method takes is the alias of a database, as defined in `DATABASES`.

Example:- `student_data = Student.objects.using('default')`

# Methods that return new QuerySets

`dates(field, kind, order='ASC')` - It returns a QuerySet that evaluates to a list of `datetime.date` objects representing all available dates of a particular kind within the contents of the QuerySet.

Where,

`field` – It should be the name of a DateField of your model.

`kind` – It should be either "year", "month", "week", or "day".

"year" returns a list of all distinct year values for the field.

"month" returns a list of all distinct year/month values for the field.

"week" returns a list of all distinct year/week values for the field. All dates will be a Monday.

"day" returns a list of all distinct year/month/day values for the field.

`order` – It should be either 'ASC' or 'DESC'. This specifies how to order the results. defaults to 'ASC'.

Each `datetime.date` object in the result list is “truncated” to the given type.

# Methods that return new QuerySets

`datetimes(field_name, kind, order='ASC', tzinfo=None)` – It returns a QuerySet that evaluates to a list of `datetime.datetime` objects representing all available dates of a particular kind within the contents of the QuerySet.

`field_name` – It should be the name of a `DateTimeField` of your model.

`Kind` - It should be either "year", "month", "week", or "day".

"year" returns a list of all distinct year values for the field.

"month" returns a list of all distinct year/month values for the field.

"week" returns a list of all distinct year/week values for the field. All dates will be a Monday.

"day" returns a list of all distinct year/month/day values for the field.

`order` – It should be either 'ASC' or 'DESC'. This specifies how to order the results. defaults to 'ASC'.

`tzinfo` – It defines the time zone to which datetimes are converted prior to truncation. This parameter must be a `datetime.tzinfo` object. If it's None, Django uses the current time zone. It has no effect when `USE_TZ` is False.

Each `datetime.datetime` object in the result list is “truncated” to the given type.

# Methods that return new QuerySets

none() - Calling none() will create a queryset that never returns any objects and no query will be executed when accessing the results. A qs.none() queryset is an instance of EmptyQuerySet.

Example:- student\_data = Student.objects.none()

union(\*other\_qs, all=False) - Uses SQL's UNION operator to combine the results of two or more QuerySets. The UNION operator selects only distinct values by default. To allow duplicate values, use the all=True argument.

Example:- student\_data = qs2.union(qs1, all=True)

intersection(\*other\_qs) – Uses SQL's INTERSECT operator to return the shared elements of two or more QuerySets.

Example:- student\_data = qs1.intersection(qs2)

difference(\*other\_qs) - Uses SQL's EXCEPT operator to keep only elements present in the QuerySet but not in some other QuerySets.

Example:- student\_data = qs1.difference(qs2)

# Methods that return new QuerySets

- `select_related(*fields)`
- `defer(*fields)`
- `only(*fields)`
- `prefetch_related(*lookups)`
- `extra(select=None, where=None, params=None, tables=None, order_by=None, select_params=None)`
- `select_for_update(nowait=False, skip_locked=False, of=())`
- `raw(raw_query, params=None, translations=None)`
- `annotate(*args, **kwargs)`

# Operators that return new QuerySets

AND (&) - Combines two QuerySets using the SQL AND operator.

Example:-

```
student_data = Student.objects.filter(id=6) & Student.objects.filter(roll=106)
```

```
student_data = Student.objects.filter(id=6, roll=106)
```

```
student_data = Student.objects.filter(Q(id=6) & Q(roll=106))
```

OR (|) - Combines two QuerySets using the SQL OR operator.

Example:-

```
Student.objects.filter(id=11) | Student.objects.filter(roll=106)
```

```
Student.objects.filter(Q(id=11) | Q(roll=106))
```

# Methods that do not return new QuerySets

## Retrieving a single object

get( ) - It returns one single object. If There is no result match it will raise DoesNotExist exception. If more than one item matches the get() query. It will raise MultipleObjectsReturned.

Example:- Student.objects.get(pk=1)

first() - It returns the first object matched by the queryset, or None if there is no matching object. If the QuerySet has no ordering defined, then the queryset is automatically ordered by the primary key.

Example:- student\_data = Student.objects.first()

student\_data = Student.objects.order\_by('name').first()

last() - It returns the last object matched by the queryset, or None if there is no matching object. If the QuerySet has no ordering defined, then the queryset is automatically ordered by the primary key.

# Methods that do not return new QuerySets

latest(\*fields) - It returns the latest object in the table based on the given field(s).

Example:- `student_data = Student.objects.latest('pass_date')`

earliest(\*fields) - It returns the earliest object in the table based on the given field(s).

Example:- `student_data = Student.objects.earliest('pass_date')`

exists() - It returns True if the QuerySet contains any results, and False if not. This tries to perform the query in the simplest and fastest way possible, but it does execute nearly the same query as a normal QuerySet query.

Example:-

```
student_data = Student.objects.all()
```

```
print(student_data.exists())
```

# Methods that do not return new QuerySets

create(\*\*kwargs) - A convenience method for creating an object and saving it all in one step.

Example:-

```
s = Student(name='Sameer', roll=112, city='Bokaro', marks=60, pass_date='2020-5-4')
```

```
s.save(force_insert=True)
```

```
s = Student.objects.create(name='Sameer', roll=112, city='Bokaro', marks=60, pass_date='2020-5-4')
```

get\_or\_create(defaults=None, \*\*kwargs) - A convenience method for looking up an object with the given kwargs (may be empty if your model has defaults for all fields), creating one if necessary.

It returns a tuple of (object, created), where object is the retrieved or created object and created is a boolean specifying whether a new object was created.

Example:-

```
student_data, created = Student.objects.get_or_create(name='Sameer', roll=112, city='Bokaro',  
marks=60, pass_date='2020-5-4')
```

```
print(student_data, created)
```

# Methods that do not return new QuerySets

update(\*\*kwargs) - Performs an SQL update query for the specified fields, and returns the number of rows matched (which may not be equal to the number of rows updated if some rows already have the new value).

Example:-

```
student_data = Student.objects.filter(id=12).update(name='Kabir', marks=80)
```

```
# Update student's city Pass who has marks 60
```

```
student_data = Student.objects.filter(marks=60).update(city='Pass')
```

```
student_data = Student.objects.get(id=12).update(name='Kabir', marks=80)
```

# Methods that do not return new QuerySets

`update_or_create(defaults=None, **kwargs)` – A convenience method for updating an object with the given kwargs, creating a new one if necessary. The defaults is a dictionary of (field, value) pairs used to update the object. The values in defaults can be callables.

It returns a tuple of (object, created), where object is the created or updated object and created is a boolean specifying whether a new object was created.

The `update_or_create` method tries to fetch an object from database based on the given kwargs. If a match is found, it updates the fields passed in the defaults dictionary.

Example:-

```
student_data, created = Student.objects.update_or_create(id=14, name='Kohli',  
defaults={'name':'Sameer'})
```

# Methods that do not return new QuerySets

`bulk_create(objs, batch_size=None, ignore_conflicts=False)` – This method inserts the provided list of objects into the database in an efficient manner.

The model’s `save()` method will not be called, and the `pre_save` and `post_save` signals will not be sent.

It does not work with child models in a multi-table inheritance scenario.

If the model’s primary key is an `AutoField` it does not retrieve and set the primary key attribute, as `save()` does, unless the database backend supports it (currently PostgreSQL).

It does not work with many-to-many relationships.

It casts `objs` to a list, which fully evaluates `objs` if it’s a generator. The cast allows inspecting all objects so that any objects with a manually set primary key can be inserted first.

The `batch_size` parameter controls how many objects are created in a single query. The default is to create all objects in one batch, except for SQLite where the default is such that at most 999 variables per query are used.

On databases that support it (all but Oracle), setting the `ignore_conflicts` parameter to `True` tells the database to ignore failure to insert any rows that fail constraints such as duplicate unique values. Enabling this parameter disables setting the primary key on each model instance.

# Methods that do not return new QuerySets

Example:-

```
objs = [  
    Student(name='Sonal', roll=120, city='Dhanbad', marks=40, pass_date='2020-5-4'),  
    Student(name='Kunal', roll=121, city='Dumka', marks=50, pass_date='2020-5-7'),  
    Student(name='Anisa', roll=122, city='Giridih', marks=70, pass_date='2020-5-9') ]  
student_data = Student.objects.bulk_create(objs)
```

# Methods that do not return new QuerySets

`bulk_update(objs, fields, batch_size=None)` - This method efficiently updates the given fields on the provided model instances, generally with one query. `QuerySet.update()` is used to save the changes, so this is more efficient than iterating through the list of models and calling `save()` on each of them.

You cannot update the model's primary key.

Each model's `save()` method isn't called, and the `pre_save` and `post_save` signals aren't sent.

If updating a large number of columns in a large number of rows, the SQL generated can be very large. Avoid this by specifying a suitable `batch_size`.

Updating fields defined on multi-table inheritance ancestors will incur an extra query per ancestor.

If `objs` contains duplicates, only the first one is updated.

The `batch_size` parameter controls how many objects are saved in a single query. The default is to update all objects in one batch, except for SQLite and Oracle which have restrictions on the number of variables used in a query.

Example:-

```
all_student_data = Student.objects.all()
```

```
for stu in all_student_data:
```

```
    stu.city = 'Bhel'
```

```
student_data = Student.objects.bulk_update(all_student_data, ['city'])
```

# Methods that do not return new QuerySets

in\_bulk(id\_list=None, field\_name='pk') – It takes a list of field values (id\_list) and the field\_name for those values, and returns a dictionary mapping each value to an instance of the object with the given field value. If id\_list isn't provided, all objects in the queryset are returned. field\_name must be a unique field, and it defaults to the primary key.

Example:-

```
student_data = Student.objects.in_bulk([1, 2])
```

```
print(student_data[1].name)
```

```
print()
```

```
student_data1 = Student.objects.in_bulk([])
```

```
print(student_data1)
```

```
print()
```

```
student_data2 = Student.objects.in_bulk()
```

```
print(student_data2)
```

```
print()
```

# Methods that do not return new QuerySets

`delete()` - The delete method, conveniently, is named `delete()`. This method immediately deletes the object and returns the number of objects deleted and a dictionary with the number of deletions per object type.

Example:-

Delete One Record

```
student_data = Student.objects.get(pk=22)  
deleted = student_data.delete()
```

Delete in Bulk

You can also delete objects in bulk. Every QuerySet has a `delete()` method, which deletes all members of that QuerySet.

```
Example:- student_data = Student.objects.filter(marks=50).delete()
```

Delete All Records

```
Example:- student_data = Student.objects.all().delete()
```

# Methods that do not return new QuerySets

count() - It returns an integer representing the number of objects in the database matching the QuerySet. A count() call performs a SELECT COUNT(\*) behind the scenes.

Example:-

```
student_data = Student.objects.all()  
print(student_data.count())
```

explain(format=None, \*\*options) – It returns a string of the QuerySet’s execution plan, which details how the database would execute the query, including any indexes or joins that would be used. Knowing these details may help you improve the performance of slow queries. explain() is supported by all built-in database backends except Oracle because an implementation there isn’t straightforward. The format parameter changes the output format from the databases’s default, usually text-based. PostgreSQL supports ‘TEXT’, ‘JSON’, ‘YAML’, and ‘XML’. MySQL supports ‘TEXT’ (also called ‘TRADITIONAL’) and ‘JSON’.

Example:- `print(Student.objects.all().explain())`

# Methods that do not return new QuerySets

- `aggregate(*args, **kwargs)`
- `as_manager()`
- `iterator(chunk_size=2000)`

# Field Lookups

Field lookups are how you specify the meat of an SQL WHERE clause.

They're specified as keyword arguments to the QuerySet methods filter(), exclude() and get().

If you pass an invalid keyword argument, a lookup function will raise TypeError.

Syntax:- field\_\_lookuptype=value

Example:- Student.objects.filter(marks\_\_lt='50')

SELECT \* FROM myapp\_student WHERE marks < '50';

The field specified in a lookup has to be the name of a model field.

In case of a ForeignKey you can specify the field name suffixed with \_id. In this case, the value parameter is expected to contain the raw value of the foreign model's primary key.

Example:- Student.objects.filter(stu\_id=10)

# Field Lookups

exact - Exact match. If the value provided for comparison is None, it will be interpreted as an SQL NULL. This is case sensitive

Example:- `Student.objects.get(name__exact='sonam')`

iexact - Exact match. If the value provided for comparison is None, it will be interpreted as an SQL NULL. This is case insensitive

Example:- `Student.objects.get(name__iexact='sonam')`

contains - Case-sensitive containment test.

Example:- `Student.objects.get(name__contains='kumar')`

icontains - Case-insensitive containment test.

Example:- `Student.objects.get(name__icontains='kumar')`

in - In a given iterable; often a list, tuple, or queryset. It's not a common use case, but strings (being iterables) are accepted.

Example:- `Student.objects.filter(id__in=[1, 5, 7])`

# Field Lookups

gt - Greater than.

Example: - Student.objects.filter(marks\_\_gt=50)

gte - Greater than or equal to.

Example: - Student.objects.filter(marks\_\_gte=50)

lt - Less than.

Example: - Student.objects.filter(marks\_\_lt=50)

lte - Less than or equal to.

Example: - Student.objects.filter(marks\_\_lte=50)

# Field Lookups

startswith - Case-sensitive starts-with.

Example: - Student.objects.filter(name\_\_startswith='r')

istartswith - Case-insensitive starts-with.

Example: - Student.objects.filter(name\_\_istartswith='r')

endswith - Case-sensitive ends-with.

Example:- Student.objects.filter(name\_\_endswith='j')

iendswith - Case-insensitive ends-with.

Example:- Student.objects.filter(name\_\_iendswith='j')

# Field Lookups

range - Range test (inclusive).

Example:- `Student.objects.filter(passdate__range=('2020-04-01', '2020-05-05'))`

SQL: - `SELECT ... WHERE admission_date BETWEEN '2020-04-01' and '2020-05-05';`

You can use range anywhere you can use BETWEEN in SQL — for dates, numbers and even characters.

date - For datetime fields, casts the value as date. Allows chaining additional field lookups. Takes a date value.

Example:-

`Student.objects.filter(admdatetime__date=date(2020, 6, 5))`

`Student.objects.filter(admdatetime__date__gt=date(2020, 6, 5))`

year - For date and datetime fields, an exact year match. Allows chaining additional field lookups. Takes an integer year.

Example:-

`Student.objects.filter(passdate__year=2020)`

`Student.objects.filter(passdate__year__gt=2019)`

# Field Lookups

month - For date and datetime fields, an exact month match. Allows chaining additional field lookups. Takes an integer 1 (January) through 12 (December).

Example:-

```
Student.objects.filter(passdate__month=6)
```

```
Student.objects.filter(passdate__month__gt=5)
```

day - For date and datetime fields, an exact day match. Allows chaining additional field lookups. Takes an integer day.

Example:-

```
Student.objects.filter(passdate__day=5)
```

```
Student.objects.filter(passdate__day__gt=3)
```

This will match any record with a pub\_date on the third day of the month, such as January 3, July 3, etc.

# Field Lookups

week - For date and datetime fields, return the week number (1-52 or 53) according to ISO-8601, i.e., weeks start on a Monday and the first week contains the year's first Thursday.

Example:-

```
Student.objects.filter(passdate__week=23)
```

```
Student.objects.filter(passdate__week__gt=22)
```

week\_day - For date and datetime fields, a ‘day of the week’ match. Allows chaining additional field lookups.

Takes an integer value representing the day of week from 1 (Sunday) to 7 (Saturday).

Example:-

```
Student.objects.filter(passdate__week_day=6)
```

```
Student.objects.filter(passdate__week_day__gt=5)
```

This will match any record with a admission\_date that falls on a Monday (day 2 of the week), regardless of the month or year in which it occurs. Week days are indexed with day 1 being Sunday and day 7 being Saturday.

# Field Lookups

quarter - For date and datetime fields, a ‘quarter of the year’ match. Allows chaining additional field lookups. Takes an integer value between 1 and 4 representing the quarter of the year.

Example to retrieve entries in the second quarter (April 1 to June 30):

```
Student.objects.filter(passdate__quarter=2)
```

time - For datetime fields, casts the value as time. Allows chaining additional field lookups. Takes a datetime.time value.

Example:- `Student.objects.filter(admdatetime__time__gt=time(6,00))`

hour - For datetime and time fields, an exact hour match. Allows chaining additional field lookups. Takes an integer between 0 and 23.

Example:- `Student.objects.filter(admdatetime__hour__gt=5)`

# Field Lookups

minute - For datetime and time fields, an exact minute match. Allows chaining additional field lookups. Takes an integer between 0 and 59.

Example:- `Student.objects.filter(admdatetime__minute__gt=50)`

second - For datetime and time fields, an exact second match. Allows chaining additional field lookups. Takes an integer between 0 and 59.

Example:- `Student.objects.filter(admdatetime__second__gt=30)`

isnull - Takes either True or False, which correspond to SQL queries of IS NULL and IS NOT NULL, respectively.

Example:- `Student.objects.filter(roll__isnull=False)`

regex

iregex

# Aggregation

sometimes you will need to retrieve values that are derived by summarizing or aggregating a collection of objects.

aggregate() - It is a terminal clause for a QuerySet that, when invoked, returns a dictionary of name-value pairs. The name is an identifier for the aggregate value; the value is the computed aggregate. The name is automatically generated from the name of the field and the aggregate function.

Syntax:- `aggregate(name=agg_function('field'), name=agg_function('field'),)`

field - It describes the aggregate value that we want to compute.

name - If you want to manually specify a name for the aggregate value, you can do so by providing that name when you specify the aggregate clause.

annotate() - Per-object summaries can be generated using the annotate() clause. When an annotate() clause is specified, each object in the QuerySet will be annotated with the specified values. The output of the annotate() clause is a QuerySet; this QuerySet can be modified using any other QuerySet operation, including filter(), order\_by(), or even additional calls to annotate().

Syntax:- `annotate(name=agg_function('field'), name=agg_function('field'),)`

# Aggregation Functions

Django provides the following aggregation functions in the **django.db.models** module.

**Avg(expression, output\_field=None, distinct=False, filter=None, \*\*extra)** - It returns the mean value of the given expression, which must be numeric unless you specify a different output\_field.

Default alias: <field>\_\_avg

Return type: float if input is int, otherwise same as input field, or output\_field if supplied

Has one optional argument:

**distinct** - If distinct=True, Avg returns the mean value of unique values. This is the SQL equivalent of AVG(DISTINCT <field>). The default value is False.

**Count(expression, distinct=False, filter=None, \*\*extra)** - It returns the number of objects that are related through the provided expression.

Default alias: <field>\_\_count

Return type: int

Has one optional argument:

**distinct** - If distinct=True, the count will only include unique instances. This is the SQL equivalent of COUNT(DISTINCT <field>). The default value is False.

# Aggregation Functions

Max(expression, output\_field=None, filter=None, \*\*extra)- It returns the maximum value of the given expression.

Default alias: <field>\_max

Return type: same as input field, or output\_field if supplied

Min(expression, output\_field=None, filter=None, \*\*extra) - It returns the minimum value of the given expression.

Default alias: <field>\_min

Return type: same as input field, or output\_field if supplied

Sum(expression, output\_field=None, distinct=False, filter=None, \*\*extra) - It computes the sum of all values of the given expression.

Default alias: <field>\_sum

Return type: same as input field, or output\_field if supplied

Has one optional argument:

distinct - If distinct=True, Sum returns the sum of unique values. This is the SQL equivalent of SUM(DISTINCT <field>). The default value is False.

# Aggregation Functions

`StdDev(expression, output_field=None, sample=False, filter=None, **extra)` - It returns the standard deviation of the data in the provided expression.

Default alias: `<field>_stddev`

Return type: float if input is int, otherwise same as input field, or `output_field` if supplied

Has one optional argument:

`sample` - By default, `StdDev` returns the population standard deviation. However, if `sample=True`, the return value will be the sample standard deviation.

`Variance(expression, output_field=None, sample=False, filter=None, **extra)` - It returns the variance of the data in the provided expression.

Default alias: `<field>_variance`

Return type: float if input is int, otherwise same as input field, or `output_field` if supplied

Has one optional argument:

`sample` - By default, `Variance` returns the population variance. However, if `sample=True`, the return value will be the sample variance.

# Q Objects

Q object is an object used to encapsulate a collection of keyword arguments. These keyword arguments are specified as in “Field lookups” .

If you need to execute more complex queries, you can use Q objects.

Q objects can be combined using the & and | operators. When an operator is used on two Q objects, it yields a new Q object.

**from django.db.models import Q**

& (AND) Operator

Example:- Student.objects.filter(Q(id=6) & Q(roll=106))

| (OR) Operator

Example:- Student.objects.filter(Q(id=6) | Q(roll=108))

~ Negation Operator

Example:- Student.objects.filter(~Q(id=6))

# Limiting QuerySets

Use a subset of Python's array-slicing syntax to limit your QuerySet to a certain number of results. This is the equivalent of SQL's LIMIT and OFFSET clauses.

`Student.objects.all()[:5]` - This returns First 5 objects

`Student.objects.all()[5:10]` - This returns sixth through tenth objects

`Student.objects.all()[-1]` - This is not valid.

`Student.objects.all():[10:2]` - This returns a list of every second object of the first 10.

# 51-Model Inheritance

Model inheritance in Django works almost identically to the way normal class inheritance works in Python. The base class should subclass `django.db.models.Model`.

- Abstract Base Classes
- Multi-table Inheritance
- Proxy Models

# Abstract Base Classes

Abstract base classes are useful when you want to put some common information into a number of other models.

You write your base class and put *abstract=True* in the *Meta* class.

This model will then not be used to create any database table. Instead, when it is used as a base class for other models, its fields will be added to those of the child class.

It does not generate a database table or have a manager, and cannot be instantiated or saved directly.

Fields inherited from abstract base classes can be overridden with another field or value, or be removed with None.

# Abstract Base Classes

```
from django.db import models

class CommonInfo(models.Model):
    name = models.CharField(max_length=70)
    age = models.IntegerField()

    class Meta:
        abstract = True

class Student(CommonInfo):
    fees = models.IntegerField()

class Teacher(CommonInfo):
    salary = models.IntegerField()
```

# Abstract Base Classes

Meta Inheritance - When an abstract base class is created, Django makes any Meta inner class you declared in the base class available as an attribute.

If a child class does not declare its own Meta class, it will inherit the parent's Meta.

If the child wants to extend the parent's Meta class, it can subclass it.

Django does make one adjustment to the Meta class of an abstract base class: before installing the Meta attribute, it sets abstract=False.

This means that children of abstract base classes don't automatically become abstract classes themselves.

You can make an abstract base class that inherits from another abstract base class. You just need to remember to explicitly set abstract=True each time.

# Abstract Base Classes

```
from django.db import models

class CommonInfo(models.Model):
    name = models.CharField(max_length=70)
    age = models.IntegerField()

    class Meta:
        abstract = True

class Student(CommonInfo):
    fees = models.IntegerField()

    class Meta:
        abstract = False
```

# Abstract Base Classes

When you are using related\_name or related\_query\_name in an abstract base class (only), part of the value should contain '%(app\_label)s' and '%(class)s'.

- '%(class)s' is replaced by the lowercased name of the child class that the field is used in.
- '%(app\_label)s' is replaced by the lowercased name of the app the child class is contained within. Each installed application name must be unique and the model class names within each app must also be unique, therefore the resulting name will end up being different.

```
from django.db import models

class Base(models.Model):
    m2m = models.ManyToManyField(
        OtherModel,
        related_name="%(app_label)s_%(class)s_related",
        related_query_name="%(app_label)s_%(class)ss",
    )
    class Meta:
        abstract = True
```

```
class ChildA(Base):
    pass
```

```
class ChildB(Base):
    pass
```

# Multi-table Inheritance

In this inheritance each model have their own database table, which means base class and child class will have their own table.

The inheritance relationship introduces links between the child model and each of its parents (via an automatically-created OneToOneField).

```
from django.db import models

class ExamCenter(models.Model):
    cname = models.CharField(max_length=70)
    city = models.CharField(max_length=70)
```

```
class Student(ExamCenter):
    name = models.CharField(max_length=70)
    roll = models.IntegerField()
```

All of the fields of ExamCenter will also be available in Student, although the data will reside in a different database table.

# Proxy Model

Sometimes, however, you only want to change the Python behavior of a model – perhaps to change the default manager, or add a new method.

This is what proxy model inheritance is for: creating a proxy for the original model. You can create, delete and update instances of the proxy model and all the data will be saved as if you were using the original (non-proxied) model. The difference is that you can change things like the default model ordering or the default manager in the proxy, without having to alter the original.

Proxy models are declared like normal models. You tell Django that it's a proxy model by setting the `proxy` attribute of the `Meta` class to `True`.

# Proxy Model

```
from django.db import models
```

```
class ExamCenter(models.Model):
```

```
    cname = models.CharField(max_length=70)
```

```
    city = models.CharField(max_length=70)
```

```
class MyExamCenter(ExamCenter):
```

```
    class Meta:
```

```
        proxy = True
```

```
        ordering = ['city']
```

```
    def do_something(self):
```

```
        pass
```

```
class MySome(ExamCenter):
```

```
    objects = NewManager()
```

```
    class Meta:
```

```
        proxy = True
```

# Proxy Model

- A proxy model must inherit from exactly one non-abstract model class.
- You can't inherit from multiple non-abstract models as the proxy model doesn't provide any connection between the rows in the different database tables.
- A proxy model can inherit from any number of abstract model classes, providing they do not define any model fields.
- A proxy model may also inherit from any number of proxy models that share a common non-abstract parent class.
- If you don't specify any model managers on a proxy model, it inherits the managers from its model parents.
- If you define a manager on the proxy model, it will become the default, although any managers defined on the parent classes will still be available.

# 52-Model Manager

A Manager is the interface through which database query operations are provided to Django models. At least one Manager exists for every model in a Django application.

Model manager is used to interact with database.

By default, Django adds a Manager with the name objects to every Django model class.

`django.db.models.manager.Manager`

# Change Manager Name

By default, Django adds a Manager with the name objects to every Django model class. However, if you want to use objects as a field name, or if you want to use a name other than objects for the Manager, you can rename it on a per-model basis.

To rename the Manager for a given class, define a class attribute of type `models.Manager()` on that model.

```
from django.db import models
```

```
class Student(models.Model):
```

```
    name = models.CharField(max_length=70)
```

```
    roll = models.IntegerField()
```

```
students = models.Manager()
```

```
student_data = Student.students.all()
```

# Custom Model Manager

You can use a custom Manager in a particular model by extending the base Manager class and instantiating your custom Manager in your model.

A custom Manager method can return anything you want. It doesn't have to return a QuerySet.

- to modify the initial QuerySet the Manager returns
- to add extra Manager methods

# Modify the initial QuerySet

A Manager's base QuerySet returns all objects in the system. You can override a Manager's base QuerySet by overriding the Manager.get\_queryset() method. get\_queryset() should return a QuerySet with the properties you require.

## Write Model Manager

```
class CustomManager(models.Manager):
    def get_queryset(self):      # overriding Built-in method called when we call all()
        return super().get_queryset().order_by('name')
```

## Associate Manager with Model

```
Class Student(models.Model):
```

```
    objects = models.Manager()
    students = CustomManager()
```

```
views.py
```

```
Student_data = Student.objects.all()
Student_data = Student.students.all()
```

```
# Default Manager
# Custom Manager
```



You can associate more than one manager in one Model

```
# Work as per default Manager
# Work as per Custom manager
```

# Add extra Manager methods

Adding extra Manager methods is the preferred way to add “table-level” functionality to your models.

## Write Model Manager

```
class CustomManager(models.Manager):  
    def get_stu_roll_range(self, r1, r2):  
        return super().get_queryset().filter(roll__range=(r1, r2))
```

## Associate Manager with Model

```
Class Student(models.Model):  
    objects = models.Manager()  
    students = CustomManager()
```

views.py

```
Student_data = Student.objects.all()  
Student_data = Student.students.get_stu_roll_range(101, 103)
```

# **53-Model Relationship**

Django offers ways to define the three most common types of database relationships

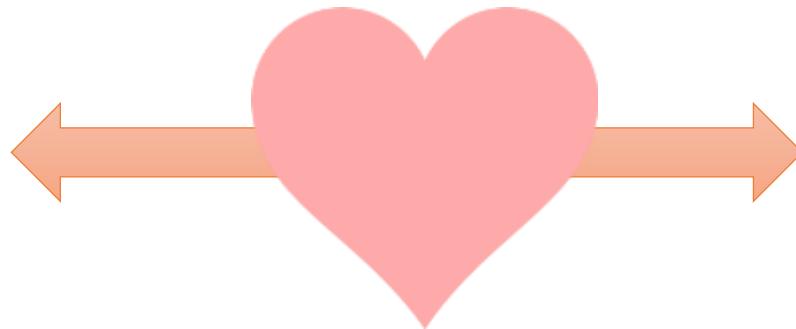
- One to One Relationship
- Many to One Relationship
- Many to Many Relationships

# One to One Relationship

When one row of table A can be linked to one row of table B.



Husband



Wife

# One to One

## User

ID	Username	Password
1	Rahul	rahul12
2	Sonam	sonam12
3	Kunal	kunal12

## Page

ID	Page Name	Page Cat	Page Publish Date
1	Geekyshows	Programming	12-12-2000
2	World News	News	11-09-2003
3	DjangoLove	Programming	07-01-2001

# One to One Relationship

One to One Relationship - To define a one-to-one relationship, use **OneToOneField**. You use it just like any other Field type by including it as a class attribute of your model.

OneToOneField requires a positional argument, the class to which the model is related.

Syntax:- `OneToOneField(to, on_delete, parent_link=False, **options)`

Where,

`to` - The class to which the model is related.

`on_delete` - When an object referenced by a ForeignKey is deleted, Django will emulate the behavior of the SQL constraint specified by the `on_delete` argument. `on_delete` doesn't create an SQL constraint in the database.

`parent_link` - When True and used in a model which inherits from another concrete model, indicates that this field should be used as the link back to the parent class, rather than the extra OneToOneField which would normally be implicitly created by subclassing.

`limit_choices_to` - Sets a limit to the available choices for this field when this field is rendered using a ModelForm or the admin (by default, all objects in the queryset are available to choose). Either a dictionary, a Q object, or a callable returning a dictionary or Q object can be used.

# One to One Relationship

`related_name` - The name to use for the relation from the related object back to this one. It's also the default value for `related_query_name` (the name to use for the reverse filter name from the target model).

If you'd prefer Django not to create a backwards relation, set `related_name` to '+' or end it with '+'.

`related_query_name` - The name to use for the reverse filter name from the target model. It defaults to the value of `related_name` or `default_related_name` if set, otherwise it defaults to the name of the model.

`to_field` - The field on the related object that the relation is to. By default, Django uses the primary key of the related object. If you reference a different field, that field must have `unique=True`.

`swappable` - Controls the migration framework's reaction if this `ForeignKey` is pointing at a swappable model. If it is `True` - the default - then if the `ForeignKey` is pointing at a model which matches the current value of `settings.AUTH_USER_MODEL` (or another swappable model setting) the relationship will be stored in the migration using a reference to the setting, not to the model directly.

# One to One Relationship

`db_constraint` - Controls whether or not a constraint should be created in the database for this foreign key. The default is `True`, and that's almost certainly what you want; setting this to `False` can be very bad for data integrity. That said, here are some scenarios where you might want to do this:

You have legacy data that is not valid.

You're sharding your database.

If this is set to `False`, accessing a related object that doesn't exist will raise its `DoesNotExist` exception.

# on\_delete

on\_delete - When an object referenced by a ForeignKey is deleted, Django will emulate the behavior of the SQL constraint specified by the on\_delete argument. on\_delete doesn't create an SQL constraint in the database.

The possible values for on\_delete are found in django.db.models:

- CASCADE - Cascade deletes. Django emulates the behavior of the SQL constraint ON DELETE CASCADE and also deletes the object containing the ForeignKey.
- PROTECT - Prevent deletion of the referenced object by raising ProtectedError, a subclass of django.db.IntegrityError.
- SET\_NULL - Set the ForeignKey null; this is only possible if null is True.
- SET\_DEFAULT - Set the ForeignKey to its default value; a default for the ForeignKey must be set.
- SET() - Set the ForeignKey to the value passed to SET(), or if a callable is passed in, the result of calling it.
- DO\_NOTHING - Take no action. If your database backend enforces referential integrity, this will cause an IntegrityError unless you manually add an SQL ON DELETE constraint to the database field.

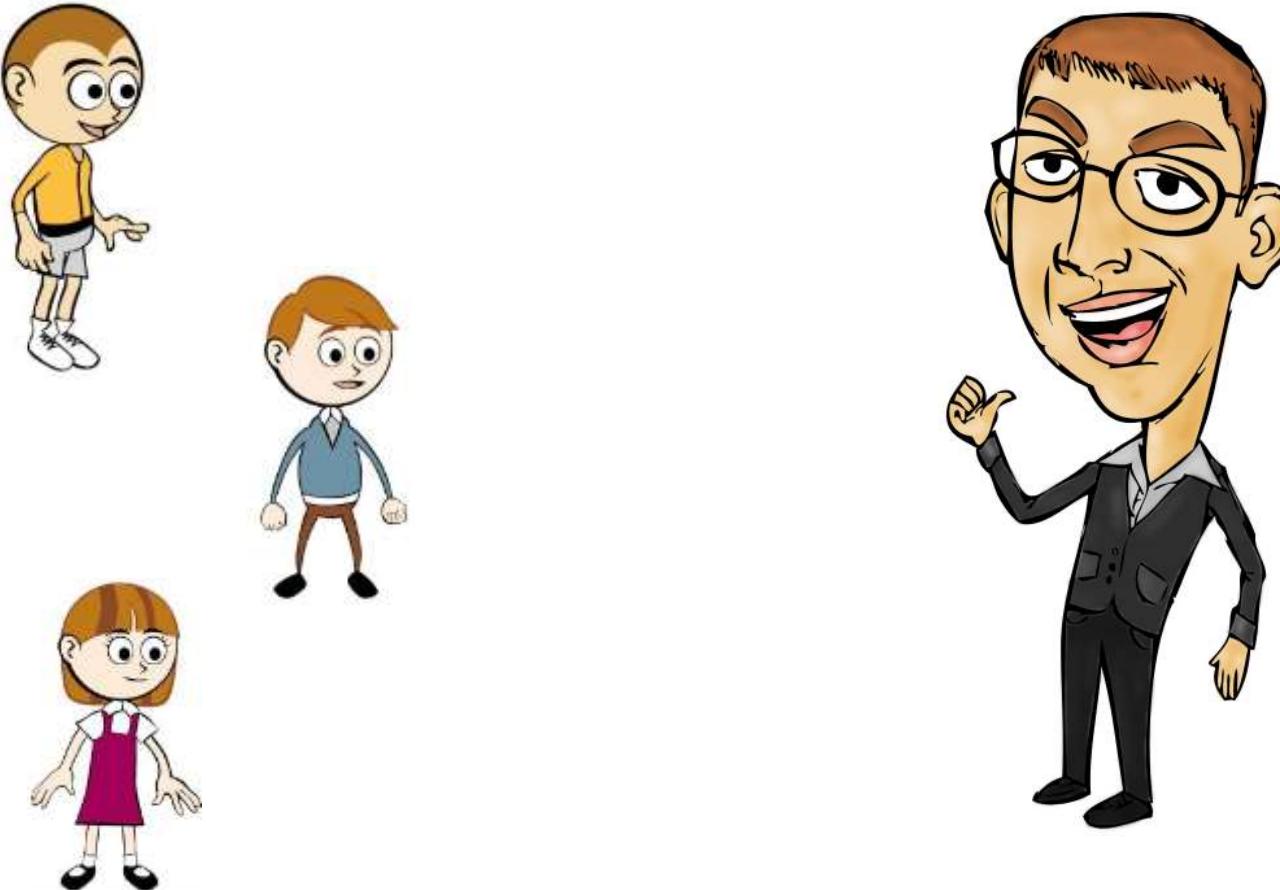
# One to One Relationship

```
class User(models.Model):
    user_name = models.CharField(max_length=70)
    password = models.CharField(max_length=70)

class Page(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    page_name = models.CharField(max_length=70)
    page_cat = models.CharField(max_length=70)
    page_publish_date = models.DateField()
```

# Many to One Relationship

When one or more row of table B can be linked to one row of table A.



# Many to One Relationship

**Post**

<b>ID</b>	<b>Post Title</b>	<b>Post Cat</b>	<b>Post Publish Date</b>	<b>User_id</b>
1	Title 1	django	12-12-2000	1
2	Title 2	django	11-09-2003	1
3	Title 3	python	07-01-2001	2

**User**

<b>ID</b>	<b>Username</b>	<b>Password</b>
1	Rahul	rahul12
2	Sonam	sonam12
3	Kunal	kunal12

# Many to One Relationship

Many to One Relationship - To define a many-to-one relationship, use **ForeignKey**. You use it just like any other Field type: by including it as a class attribute of your model.

A many-to-one relationship requires two positional arguments: the class to which the model is related and the `on_delete` option.

Syntax:- `ForeignKey(to, on_delete, **options)`

`to` - The class to which the model is related.

`on_delete` - When an object referenced by a `ForeignKey` is deleted, Django will emulate the behavior of the SQL constraint specified by the `on_delete` argument. `on_delete` doesn't create an SQL constraint in the database.

`limit_choices_to` - Sets a limit to the available choices for this field when this field is rendered using a `ModelForm` or the admin (by default, all objects in the queryset are available to choose). Either a dictionary, a `Q` object, or a callable returning a dictionary or `Q` object can be used.

# Many to One Relationship

`related_name` - The name to use for the relation from the related object back to this one. It's also the default value for `related_query_name` (the name to use for the reverse filter name from the target model).

If you'd prefer Django not to create a backwards relation, set `related_name` to '+' or end it with '+'.

`related_query_name` - The name to use for the reverse filter name from the target model. It defaults to the value of `related_name` or `default_related_name` if set, otherwise it defaults to the name of the model.

`to_field` - The field on the related object that the relation is to. By default, Django uses the primary key of the related object. If you reference a different field, that field must have `unique=True`.

`swappable` - Controls the migration framework's reaction if this `ForeignKey` is pointing at a swappable model. If it is `True` - the default - then if the `ForeignKey` is pointing at a model which matches the current value of `settings.AUTH_USER_MODEL` (or another swappable model setting) the relationship will be stored in the migration using a reference to the setting, not to the model directly.

# Many to One Relationship

db\_constraint - Controls whether or not a constraint should be created in the database for this foreign key. The default is True, and that's almost certainly what you want; setting this to False can be very bad for data integrity. That said, here are some scenarios where you might want to do this:

You have legacy data that is not valid.

You're sharding your database.

If this is set to False, accessing a related object that doesn't exist will raise its DoesNotExist exception.

# on\_delete

on\_delete - When an object referenced by a ForeignKey is deleted, Django will emulate the behavior of the SQL constraint specified by the on\_delete argument. on\_delete doesn't create an SQL constraint in the database.

The possible values for on\_delete are found in django.db.models:

- CASCADE - Cascade deletes. Django emulates the behavior of the SQL constraint ON DELETE CASCADE and also deletes the object containing the ForeignKey.
- PROTECT - Prevent deletion of the referenced object by raising ProtectedError, a subclass of django.db.IntegrityError.
- SET\_NULL - Set the ForeignKey null; this is only possible if null is True.
- SET\_DEFAULT - Set the ForeignKey to its default value; a default for the ForeignKey must be set.
- SET() - Set the ForeignKey to the value passed to SET(), or if a callable is passed in, the result of calling it.
- DO\_NOTHING - Take no action. If your database backend enforces referential integrity, this will cause an IntegrityError unless you manually add an SQL ON DELETE constraint to the database field.

# Many to One Relationship

```
class User(models.Model):
    user_name = models.CharField(max_length=70)
    password = models.CharField(max_length=70)

class Post(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    post_title = models.CharField(max_length=70)
    post_cat = models.CharField(max_length=70)
    post_publish_date = models.DateField()
```

# Many to Many Relationship

When one row of table A can be linked to one or more rows of table B, and vice-versa.



# Many to Many Relationship

User

ID	Username	Password
1	Rahul	rahul12
2	Sonam	sonam12
3	Kunal	kunal12

Song

ID	Song Name	Song Duration
1	Tum hi ho	5
2	Kuch Kuch	7
3	Dil to hai dil	8

song\_user

ID	Song_id	User_id
1	1	1
2	1	2
3	2	3
4	2	1

# Many to Many Relationship

Many to Many Relationships - To define a many-to-many relationship, use `ManyToManyField`. You use it just like any other Field type: by including it as a class attribute of your model.

`ManyToManyField` requires a positional argument: the class to which the model is related.

Syntax:- `ManyToManyField(to, **options)`

Where,

`to` - The class to which the model is related.

`related_name` - The name to use for the relation from the related object back to this one. It's also the default value for `related_query_name` (the name to use for the reverse filter name from the target model).

If you'd prefer Django not to create a backwards relation, set `related_name` to '+' or end it with '+'.

`related_query_name` - The name to use for the reverse filter name from the target model. It defaults to the value of `related_name` or `default_related_name` if set, otherwise it defaults to the name of the model.

# Many to Many Relationship

`limit_choices_to` - Sets a limit to the available choices for this field when this field is rendered using a ModelForm or the admin (by default, all objects in the queryset are available to choose). Either a dictionary, a `Q` object, or a callable returning a dictionary or `Q` object can be used.

`symmetrical` - If you do not want symmetry in many-to-many relationships with self, set `symmetrical` to `False`. This will force Django to add the descriptor for the reverse relationship, allowing `ManyToManyField` relationships to be non-symmetrical. Only used in the definition of `ManyToManyFields` on self.

`through` - Django will automatically generate a table to manage many-to-many relationships. However, if you want to manually specify the intermediary table, you can use the `through` option to specify the Django model that represents the intermediate table that you want to use.

`through_fields` - Only used when a custom intermediary model is specified. Django will normally determine which fields of the intermediary model to use in order to establish a many-to-many relationship automatically. `through_fields` accepts a 2-tuple ('`field1`', '`field2`'), where `field1` is the name of the foreign key to the model the `ManyToManyField` is defined on , and `field2` the name of the foreign key to the target model.

# Many to Many Relationship

`db_table` - The name of the table to create for storing the many-to-many data. If this is not provided, Django will assume a default name based upon the names of: the table for the model defining the relationship and the name of the field itself.

`db_constraint` - Controls whether or not constraints should be created in the database for the foreign keys in the intermediary table. The default is `True`, and that's almost certainly what you want; setting this to `False` can be very bad for data integrity. That said, here are some scenarios where you might want to do this:

You have legacy data that is not valid.

You're sharding your database.

It is an error to pass both `db_constraint` and `through`.

`swappable` - Controls the migration framework's reaction if this `ManyToManyField` is pointing at a swappable model. If it is `True` - the default - then if the `ManyToManyField` is pointing at a model which matches the current value of `settings.AUTH_USER_MODEL` (or another swappable model setting) the relationship will be stored in the migration using a reference to the setting, not to the model directly.

# Many to Many Relationship

Example:-

```
class User(models.Model):
    user_name = models.CharField(max_length=70)
    password = models.CharField(max_length=70)
```

```
class Song(models.Model):
    user = models.ManyToManyField(User)
    song_name = models.CharField(max_length=70)
    song_duration = models.IntegerField()
```

## **54-Type of Views**

- Function Based View
- Class Based View

# Class Based View

Class-based views provide an alternative way to implement views as Python objects instead of functions.

They do not replace function-based views.

- Base Class-Based Views / Base View
- Generic Class-Based Views / Generic View

## Advantages:-

- Organization of code related to specific HTTP methods (GET, POST, etc.) can be addressed by separate methods instead of conditional branching.
- Object oriented techniques such as mixins (multiple inheritance) can be used to factor code into reusable components.

# **Base Class-Based View**

Base class-based views can be thought of as parent views, which can be used by themselves or inherited from. They may not provide all the capabilities required for projects, in which case there are Mixins which extend what base views can do.

- View
- TemplateView
- RedirectView

# View

`django.views.generic.base.View`

The master class-based base view. All other class-based views inherit from this base class. It isn't strictly a generic view and thus can also be imported from `django.views`.

C:\Users\GS\AppData\Local\Programs\Python\Python38-32\Lib\site-packages\django\views\generic

Attribute:-

`http_method_names = ['get', 'post', 'put', 'patch', 'delete', 'head', 'options', 'trace']`

The list of HTTP method names that this view will accept.

# View

## Methods:-

`setup(self, request, *args, **kwargs)` - It initializes view instance attributes: `self.request`, `self.args`, and `self.kwargs` prior to `dispatch()`

`dispatch(self, request, *args, **kwargs)` - The view part of the view – the method that accepts a request argument plus arguments, and returns a HTTP response.

The default implementation will inspect the HTTP method and attempt to delegate to a method that matches the HTTP method; a GET will be delegated to `get()`, a POST to `post()`, and so on.

By default, a HEAD request will be delegated to `get()`. If you need to handle HEAD requests in a different way than GET, you can override the `head()` method.

`http_method_not_allowed(self, request, *args, **kwargs)` - If the view was called with a HTTP method it doesn't support, this method is called instead.

The default implementation returns `HttpResponseNotAllowed` with a list of allowed methods in plain text.

# View

## Methods:-

options(self, request, \*args, \*\*kwargs) – It handles responding to requests for the OPTIONS HTTP verb. Returns a response with the Allow header containing a list of the view's allowed HTTP method names.

as\_view(cls, \*\*initkwargs) - It returns a callable view that takes a request and returns a response.

\_allowed\_methods(self)

# View

## views.py

### Function Based View:-

```
from django.http import HttpResponse  
  
def myview(request):  
  
    return HttpResponse('<h1>Function Based View</h1>')
```

### Class Based View:-

```
from django.views import View  
  
class MyView(View):  
  
    def get(self, request):  
  
        return HttpResponse('<h1>Class Based View</h1>')
```

# View

## urls.py

### Function Based View:-

```
from django.urls import path
from school import views
urlpatterns = [    path('func/', views.myview, name='func'),    ]
```

### Class Based View:-

```
from django.urls import path
from school import views
urlpatterns = [
    path('cl/', views.MyView.as_view(), name='cl'),
]
```

# View

Django's URL resolver expects to send the request and associated arguments to a callable function, not a class, class-based views have an `as_view()` class method which returns a function that can be called when a request arrives for a URL matching the associated pattern.

The function creates an instance of the class, calls `setup()` to initialize its attributes, and then calls its `dispatch()` method.

`dispatch` looks at the request to determine whether it is a GET, POST, etc, and relays the request to a matching method if one is defined,

or raises `HttpResponseNotAllowed` if not

# TemplateView

`django.views.generic.base.TemplateView`

It renders a given template.

This view inherits methods and attributes from the following views:

- `django.views.generic.base.TemplateResponseMixin`
- `django.views.generic.base.ContextMixin`
- `django.views.generic.base.View`

```
class TemplateView(TemplateResponseMixin, ContextMixin, View):
```

# TemplateResponseMixin

It provides a mechanism to construct a TemplateResponse, given suitable context. The template to use is configurable and can be further customized by subclasses.

## Attributes:-

template\_name - The full name of a template to use as defined by a string. Not defining a template\_name will raise a django.core.exceptions.ImproperlyConfigured exception.

template\_engine - The NAME of a template engine to use for loading the template. template\_engine is passed as the using keyword argument to response\_class. Default is None, which tells Django to search for the template in all configured engines.

response\_class - The response class to be returned by render\_to\_response method. Default is TemplateResponse. The template and context of TemplateResponse instances can be altered later (e.g. in template response middleware).

If you need custom template loading or custom context object instantiation, create a TemplateResponse subclass and assign it to response\_class.

# TemplateResponseMixin

## Attributes:-

content\_type - The content type to use for the response. content\_type is passed as a keyword argument to response\_class. Default is None – meaning that Django uses 'text/html'.

## Methods:-

render\_to\_response(context, \*\*response\_kwargs)- It returns a self.response\_class instance.

If any keyword arguments are provided, they will be passed to the constructor of the response class.

Calls get\_template\_names() to obtain the list of template names that will be searched looking for an existent template.

get\_template\_names() - It returns a list of template names to search for when rendering the template. The first template that is found will be used.

If template\_name is specified, the default implementation will return a list containing template\_name (if it is specified).

# ContextMixin

A default context mixin that passes the keyword arguments received by get\_context\_data() as the template context.

## Attribute:-

extra\_context - A dictionary to include in the context. This is a convenient way of specifying some context in as\_view().

## Method:-

get\_context\_data(\*\*kwargs)- It returns a dictionary representing the template context. The keyword arguments provided will make up the returned context.

# TemplateView

## views.py

```
from django.views.generic.base import TemplateView

class HomeView(TemplateView):
    template_name = 'school/home.html'
```

## urls.py

```
from school import views

urlpatterns = [
    path('home/', views.HomeView.as_view(), name='home'),
]
```

# TemplateView With Context

## views.py

```
from django.views.generic.base import TemplateView

class HomeView(TemplateView):
    template_name = 'school/home.html'

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['name'] = 'Sonam'
        context['roll'] = 101
        return context
```

## urls.py

```
urlpatterns = [    path('home/', views.HomeView.as_view(), name='home'), ]
```

# TemplateView With Extra Context

## views.py

```
from django.views.generic.base import TemplateView

class HomeView(TemplateView):
    template_name = 'school/home.html'

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['name'] = 'Sonam'
        context['roll'] = 101
        return context
```

## urls.py

```
urlpatterns = [ path('home/', views.HomeView.as_view(extra_context={'course':'python'}), name='home'), ]
```

# RedirectView

`django.views.generic.base.RedirectView`

It redirects to a given URL.

The given URL may contain dictionary-style string formatting, which will be interpolated against the parameters captured in the URL. Because keyword interpolation is always done (even if no arguments are passed in), any "%" characters in the URL must be written as "%%" so that Python will convert them to a single percent sign on output.

If the given URL is `None`, Django will return an `HttpResponseGone` (410).

This view inherits methods and attributes from the following view:

- `django.views.generic.base.View`

# RedirectView

## Attributes:-

url - The URL to redirect to, as a string. Or None to raise a 410 (Gone) HTTP error.

pattern\_name - The name of the URL pattern to redirect to. Reversing will be done using the same args and kwargs as are passed in for this view.

permanent - Whether the redirect should be permanent. The only difference here is the HTTP status code returned. If True, then the redirect will use status code 301. If False, then the redirect will use status code 302. By default, permanent is False.

query\_string - Whether to pass along the GET query string to the new location. If True, then the query string is appended to the URL. If False, then the query string is discarded. By default, query\_string is False.

# RedirectView

## Methods:-

`get_redirect_url(*args, **kwargs)` - It constructs the target URL for redirection.

The args and kwargs arguments are positional and/or keyword arguments captured from the URL pattern, respectively.

The default implementation uses url as a starting string and performs expansion of % named parameters in that string using the named groups captured in the URL.

If url is not set, `get_redirect_url()` tries to reverse the pattern\_name using what was captured in the URL (both named and unnamed groups are used).

If requested by query\_string, it will also append the query string to the generated URL. Subclasses may implement any behavior they wish, as long as the method returns a redirect-ready URL string.

# RedirectView

## views.py

```
from django.shortcuts import render
from django.views.generic.base import RedirectView, TemplateView

class GeekyShowsRedirectView(RedirectView):
    url='https://geekyshows.com'

class GeekRedirectView(RedirectView):
    pattern_name = 'mindex'
    permanent = False
    query_string = False

    def get_redirect_url(self, *args, **kwargs):
        print(kwargs)
        kwargs['pk'] = 16
        return super().get_redirect_url(*args, **kwargs)
```

# RedirectView

## urls.py

```
from django.urls import path
from django.views.generic.base import RedirectView, TemplateView
from school import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path("", views.TemplateView.as_view(template_name='school/home.html'), name='blankindex'),
    path('home/', views.RedirectView.as_view(url='/'), name='home'),
    path('index/', views.RedirectView.as_view(pattern_name='home'), name='index'),
    path('geekyshows/', views.RedirectView.as_view(url='https://geekyshows.com'), name='go-to-
geekyshows'),
    path('geekyshows/', views.GeekyShowsRedirectView.as_view(), name='go-to-geekyshows'),
    path('home/<int:pk>/', views.GeoKRedirectView.as_view(), name='geek'),
    path('<int:pk>/', views.TemplateView.as_view(template_name='school/home.html'), name='mindex'),
    # path('home/<slug:post>/', views.GeoKRedirectView.as_view(), name='geek'),
    # path('<slug:post>/', views.TemplateView.as_view(template_name='school/home.html'), name='index'),
]
```

## **55-Type of Views**

- Function Based View
- Class Based View

# Class Based View

Class-based views provide an alternative way to implement views as Python objects instead of functions.

They do not replace function-based views.

- Base Class-Based Views / Base View
- Generic Class-Based Views / Generic View

## Advantages:-

- Organization of code related to specific HTTP methods (GET, POST, etc.) can be addressed by separate methods instead of conditional branching.
- Object oriented techniques such as mixins (multiple inheritance) can be used to factor code into reusable components.

# **Base Class-Based View**

Base class-based views can be thought of as parent views, which can be used by themselves or inherited from. They may not provide all the capabilities required for projects, in which case there are Mixins which extend what base views can do.

- View
- TemplateView
- RedirectView

# Generic Class Based View

Django's generic views are built off of those base views, and were developed as a shortcut for common usage patterns such as displaying the details of an object.

They take certain common idioms and patterns found in view development and abstract them so that you can quickly write common views of data without having to repeat yourself.

Most generic views require the `queryset` key, which is a `QuerySet` instance.

- Display View – `ListView`, `DetailView`
- Editing View – `FormView`, `CreateView`, `UpdateView`, `DeleteView`
- Date Views – `ArchiveIndexView`, `YearArchiveView`, `MonthArchiveView`, `WeekArchiveView`,  
`DayArchiveView`, `TodayArchiveView`, `DateDetailView`

# Generic Display View

The two following generic class-based views are designed to display data.

- ListView
- DetailView

# ListView

`django.views.generic.list.ListView`

A page representing a list of objects.

While this view is executing, `self.object_list` will contain the list of objects (usually, but not necessarily a queryset) that the view is operating upon.

This view inherits methods and attributes from the following views:

- `django.views.generic.list.MultipleObjectTemplateResponseMixin`
- `django.views.generic.base.TemplateResponseMixin`
- `django.views.generic.list.BaseListView`
- `django.views.generic.list.MultipleObjectMixin`
- `django.views.generic.base.View`

# MultipleObjectTemplateResponseMixin

A mixin class that performs template-based response rendering for views that operate upon a list of object instances. Requires that the view it is mixed with provides self.object\_list, the list of object instances that the view is operating on. self.object\_list may be, but is not required to be, a QuerySet.

This inherits methods and attributes from the following views:

- django.views.generic.base.TemplateResponseMixin

Attribute:-

template\_name\_suffix - The suffix to append to the auto-generated candidate template name. Default suffix is \_list.

Method:-

get\_template\_names() - It returns a list of candidate template names.

# BaseListView

A base view for displaying a list of objects. It is not intended to be used directly, but rather as a parent class of the `django.views.generic.list.ListView` or other views representing lists of objects.

This view inherits methods and attributes from the following views:

- `django.views.generic.list.MultipleObjectMixin`
- `django.views.generic.base.View`

Methods:-

`get(request, *args, **kwargs)` - It adds `object_list` to the context. If `allow_empty` is `True` then display an empty list. If `allow_empty` is `False` then raise a `404` error.

# MultipleObjectMixin

`django.views.generic.list.MultipleObjectMixin`

A mixin that can be used to display a list of objects.

If `paginate_by` is specified, Django will paginate the results returned by this. You can specify the page number in the URL in one of two ways:

- Use the `page` parameter in the URLconf.
- Pass the page number via the `page` query-string parameter.

These values and lists are 1-based, not 0-based, so the first page would be represented as page 1.

As a special case, you are also permitted to use `last` as a value for `page`.

This allows you to access the final page of results without first having to determine how many pages there are.

Note that `page` must be either a valid page number or the value `last`; any other value for `page` will result in a 404 error.

# MultipleObjectMixin

Attribute:-

allow\_empty - A boolean specifying whether to display the page if no objects are available. If this is False and no objects are available, the view will raise a 404 instead of displaying an empty page. By default, this is True.

model - The model that this view will display data for. Specifying model = Student is effectively the same as specifying queryset = Student.objects.all(), where objects stands for Student's default manager.

queryset - A QuerySet that represents the objects. If provided, the value of queryset supersedes the value provided for model.

ordering - A string or list of strings specifying the ordering to apply to the queryset. Valid values are the same as those for order\_by().

# MultipleObjectMixin

Attributes:-

`paginate_by` - An integer specifying how many objects should be displayed per page. If this is given, the view will paginate objects with `paginate_by` objects per page. The view will expect either a page query string parameter (via `request.GET`) or a page variable specified in the URLconf.

`paginate_orphans` - An integer specifying the number of “overflow” objects the last page can contain. This extends the `paginate_by` limit on the last page by up to `paginate_orphans`, in order to keep the last page from having a very small number of objects.

`page_kwarg` - A string specifying the name to use for the page parameter. The view will expect this parameter to be available either as a query string parameter (via `request.GET`) or as a kwarg variable specified in the URLconf. Defaults to `page`.

# MultipleObjectMixin

Attributes:-

`paginator_class` - The paginator class to be used for pagination. By default, `django.core.paginator.Paginator` is used. If the custom paginator class doesn't have the same constructor interface as `django.core.paginator.Paginator`, you will also need to provide an implementation for `get_paginator()`.

`context_object_name` - Designates the name of the variable to use in the context.

# MultipleObjectMixin

Methods:-

`get_queryset()` - Get the list of items for this view. This must be an iterable and may be a queryset (in which queryset-specific behavior will be enabled).

`get_ordering()` - Returns a string (or iterable of strings) that defines the ordering that will be applied to the queryset.

Returns ordering by default.

`paginate_queryset(queryset, page_size)` - Returns a 4-tuple containing (paginator, page, object\_list, is\_paginated).

Constructed by paginating queryset into pages of size `page_size`. If the request contains a `page` argument, either as a captured URL argument or as a GET argument, `object_list` will correspond to the objects from that page.

# MultipleObjectMixin

Methods:-

`get_paginate_by(queryset)` - Returns the number of items to paginate by, or `None` for no pagination. By default this returns the value of `paginate_by`.

`get Paginator(queryset, per_page, orphans=0, allow_empty_first_page=True)` - Returns an instance of the paginator to use for this view. By default, instantiates an instance of `paginator_class`.

`get_paginate_orphans()` - An integer specifying the number of “overflow” objects the last page can contain. By default this returns the value of `paginate_orphans`.

`get_allow_empty()` - Return a boolean specifying whether to display the page if no objects are available. If this method returns `False` and no objects are available, the view will raise a `404` instead of displaying an empty page. By default, this is `True`.

# MultipleObjectMixin

Methods:-

`get_context_object_name(object_list)` - Return the context variable name that will be used to contain the list of data that this view is manipulating. If `object_list` is a queryset of Django objects and `context_object_name` is not set, the context name will be the `model_name` of the model that the queryset is composed from, with postfix '`_list`' appended. For example, the model Article would have a context object named `article_list`.

`get_context_data(**kwargs)` - Returns context data for displaying the list of objects.

Context

`object_list`: The list of objects that this view is displaying. If `context_object_name` is specified, that variable will also be set in the context, with the same value as `object_list`.

`is_paginated`: A boolean representing whether the results are paginated. Specifically, this is set to `False` if no page size has been specified, or if the available objects do not span multiple pages.

`paginator`: An instance of `django.core.paginator.Paginator`. If the page is not paginated, this context variable will be `None`.

`page_obj`: An instance of `django.core.paginator.Page`. If the page is not paginated, this context variable will be `None`.

# ListView with Default Template and Context

## views.py

```
from django.views.generic.list import ListView  
from .models import Student  
class StudentListView(ListView):  
    model = Student
```

## urls.py

```
urlpatterns = [ path('student/', views.StudentListView.as_view(), name='student'), ]
```

### Default Template

Syntax:- AppName/ModelClassName\_list.html

Example:- school/student\_list.html

### Default Context

Syntax:- ModelClassName\_list

Example:- student\_list

We can also use **object\_list**

# ListView with Custom Template and Context

## views.py

```
from django.views.generic.list import ListView  
from .models import Student  
  
class StudentListView(ListView):  
    model = Student  
    template_name = 'school/student.html'  
    context_object_name = 'students'
```

Custom Template Name

Custom Context Name

## urls.py

```
urlpatterns = [  
    path('student/', views.StudentListView.as_view(), name='student'),  
]
```

Note - *school/students.html*, *school/student\_list.html* These both will work

# DetailView

`django.views.generic.detail.DetailView`

While this view is executing, `self.object` will contain the object that the view is operating upon.

This view inherits methods and attributes from the following views:

- `django.views.generic.detail.SingleObjectTemplateResponseMixin`
- `django.views.generic.base.TemplateResponseMixin`
- `django.views.generic.detail.BaseDetailView`
- `django.views.generic.detail.SingleObjectMixin`
- `django.views.generic.base.View`

# SingleObjectTemplateResponseMixin

`django.views.generic.detail.SingleObjectTemplateResponseMixin`

A mixin class that performs template-based response rendering for views that operate upon a single object instance. Requires that the view it is mixed with provides `self.object`, the object instance that the view is operating on. `self.object` will usually be, but is not required to be, an instance of a Django model. It may be `None` if the view is in the process of constructing a new instance.

This view inherits methods and attributes from the following views:

- `django.views.generic.base.TemplateResponseMixin`

# SingleObjectTemplateResponseMixin

Attribute:-

template\_name\_field - The field on the current object instance that can be used to determine the name of a candidate template. If either template\_name\_field itself or the value of the template\_name\_field on the current object instance is None, the object will not be used for a candidate template name.

template\_name\_suffix - The suffix to append to the auto-generated candidate template name. Default suffix is \_detail.

Method:-

get\_template\_names()- Returns a list of candidate template names. Returns the following list:  
the value of template\_name on the view (if provided)  
the contents of the template\_name\_field field on the object instance that the view is operating upon (if available)

<app\_label>/<model\_name><template\_name\_suffix>.html

# SingleObjectMixin

`django.views.generic.detail.SingleObjectMixin`

Provides a mechanism for looking up an object associated with the current HTTP request.

Attribute:-

`model` - The model that this view will display data for. Specifying `model = Student` is effectively the same as specifying `queryset = Student.objects.all()`, where `objects` stands for `Student`'s default manager.

`queryset` - A `QuerySet` that represents the objects. If provided, the value of `queryset` supersedes the value provided for `model`.

`slug_field` - The name of the field on the model that contains the slug. By default, `slug_field` is '`slug`'.

`slug_url_kwarg` - The name of the URLConf keyword argument that contains the slug. By default, `slug_url_kwarg` is '`slug`'.

# SingleObjectMixin

Attribute:-

`pk_url_kwarg` - The name of the URLConf keyword argument that contains the primary key. By default, `pk_url_kwarg` is '`pk`'.

`context_object_name` - Designates the name of the variable to use in the context.

`query_pk_and_slug` - If True, causes `get_object()` to perform its lookup using both the primary key and the slug. Defaults to False.

# SingleObjectMixin

Methods:-

`get_object(queryset=None)` - Returns the single object that this view will display. If `queryset` is provided, that `queryset` will be used as the source of objects; otherwise, `get_queryset()` will be used. `get_object()` looks for a `pk_url_kwarg` argument in the arguments to the view; if this argument is found, this method performs a primary-key based lookup using that value. If this argument is not found, it looks for a `slug_url_kwarg` argument, and performs a slug lookup using the `slug_field`.

When `query_pk_and_slug` is `True`, `get_object()` will perform its lookup using both the primary key and the slug.

`get_queryset()` - Returns the `queryset` that will be used to retrieve the object that this view will display. By default, `get_queryset()` returns the value of the `queryset` attribute if it is set, otherwise it constructs a `QuerySet` by calling the `all()` method on the model attribute's default manager.

`get_slug_field()` - Returns the name of a slug field to be used to look up by slug. By default this returns the value of `slug_field`.

# SingleObjectMixin

Methods:-

`get_context_object_name(obj)` - Return the context variable name that will be used to contain the data that this view is manipulating. If `context_object_name` is not set, the context name will be constructed from the `model_name` of the model that the queryset is composed from. For example, the model Article would have context object named 'article'.

`get_context_data(**kwargs)` - Returns context data for displaying the object.

The base implementation of this method requires that the `self.object` attribute be set by the view (even if `None`). Be sure to do this if you are using this mixin without one of the built-in views that does so.

It returns a dictionary with these contents:

`object`: The object that this view is displaying (`self.object`).

`context_object_name`: `self.object` will also be stored under the name returned by `get_context_object_name()`, which defaults to the lowercased version of the model name.

# DetailView with Default Template & Context

## views.py

```
from django.views.generic.detail import DetailView  
from .models import Student  
class StudentDetailView(DetailView):  
    model = Student
```

## urls.py

```
urlpatterns = [ path('student/<int:pk>', views.StudentDetailView.as_view(), name='student'), ]
```

### Default Template

Syntax:- AppName/ModelClassName\_detail.html

Example:- school/student\_detail.html

### Default Context

Syntax:- ModelClassName

Example:= student

# DetailView with Custom Template & Context

## views.py

```
from django.views.generic.detail import DetailView  
from .models import Student  
class StudentDetailView(DetailView):
```

    model = Student

    template\_name = 'school/student.html'

    context\_object\_name = 'student'

Custom Template Name

Custom Context Name

## urls.py

```
urlpatterns = [  
    path('student/<int:pk>', views.StudentDetailView.as_view(), name='student'),  
]
```

# **Generic Editing View**

The following views are described on this page and provide a foundation for editing content:

- FormView
- CreateView
- UpdateView
- DeleteView

# FormView

`django.views.generic.edit.FormView`

A view that displays a form. On error, redisplays the form with validation errors; on success, redirects to a new URL.

This view inherits methods and attributes from the following views:

- `django.views.generic.base.TemplateResponseMixin`
- `django.views.generic.edit.BaseFormView`
- `django.views.generic.edit.FormMixin`
- `django.views.generic.edit.ProcessFormView`
- `django.views.generic.base.View`

# FormMixin

`django.views.generic.edit.FormMixin`

A mixin class that provides facilities for creating and displaying forms.

This view inherits methods and attributes from the following views:

- `django.views.generic.base.ContextMixin`

Attributes:-

`initial` - A dictionary containing initial data for the form.

`form_class` - The form class to instantiate.

`success_url` - The URL to redirect to when the form is successfully processed.

`prefix` - The prefix for the generated form.

# FormMixin

Methods:-

`get_initial()` - Retrieve initial data for the form. By default, returns a copy of initial.

`get_form_class()` - Retrieve the form class to instantiate. By default `form_class`.

`get_form(form_class=None)` - Instantiate an instance of `form_class` using `get_form_kwargs()`. If `form_class` isn't provided `get_form_class()` will be used.

`get_form_kwargs()` - Build the keyword arguments required to instantiate the form.

The `initial` argument is set to `get_initial()`. If the request is a POST or PUT, the request data (`request.POST` and `request.FILES`) will also be provided.

# FormMixin

Methods:-

`get_prefix()` - Determine the prefix for the generated form. Returns prefix by default.

`get_success_url()` - Determine the URL to redirect to when the form is successfully validated. Returns `success_url` by default.

`form_valid(form)` - Redirects to `get_success_url()`.

`form_invalid(form)` - Renders a response, providing the invalid form as context.

`get_context_data(**kwargs)` - Calls `get_form()` and adds the result to the context data with the name 'form'.

# ProcessFormView

django.views.generic.edit.ProcessFormView

A mixin that provides basic HTTP GET and POST workflow.

Extends

django.views.generic.base.View

Methods:-

get(request, \*args, \*\*kwargs) - Renders a response using a context created with get\_context\_data().

post(request, \*args, \*\*kwargs) - Constructs a form, checks the form for validity, and handles it accordingly.

put(\*args, \*\*kwargs) - The PUT action is also handled and passes all parameters through to post().

# FormView

## **forms.py**

```
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField()
    email = forms.EmailField()
    msg = forms.CharField(widget=forms.Textarea)
```

## **urls.py**

```
from school import views
urlpatterns = [
    path('contact/', views.ContactFormView.as_view(),
         name='contact'),
    path('thankyou/', views.ThanksTemplateView.as_view(),
         name='thankyou'),
]
```

## **views.py**

```
from django.views.generic.edit import FormView
class ContactFormView(FormView):
    template_name = 'school/contact.html'
    form_class = ContactForm
    success_url = '/thankyou/'

    def form_valid(self, form):
        print(form)
        print(form.cleaned_data['name'])
        return HttpResponseRedirect('Msg Sent')
```

```
class ThanksTemplateView(TemplateView):
    template_name = 'school/thankyou.html'
```

# CreateView

`django.views.generic.edit.CreateView`

A view that displays a form for creating an object, redisplaying the form with validation errors (if there are any) and saving the object.

This view inherits methods and attributes from the following views:

`django.views.generic.detail.SingleObjectTemplateResponseMixin`

`django.views.generic.base.TemplateResponseMixin`

`django.views.generic.edit.BaseCreateView`

`django.views.generic.edit.ModelFormMixin`

`django.views.generic.edit.FormMixin`

`django.views.generic.detail.SingleObjectMixin`

`django.views.generic.edit.ProcessFormView`

`django.views.generic.base.View`

# CreateView

Attributes:-

template\_name\_suffix - The CreateView page displayed to a GET request uses a template\_name\_suffix of '\_form'.

object - When using CreateView you have access to self.object, which is the object being created. If the object hasn't been created yet, the value will be None.

# ModelFormMixin

A form mixin that works on ModelForms, rather than a standalone form.

Since this is a subclass of `SingleObjectMixin`, instances of this mixin have access to the `model` and `queryset` attributes, describing the type of object that the `ModelForm` is manipulating.

If you specify both the `fields` and `form_class` attributes, an `ImproperlyConfigured` exception will be raised.

Mixins

`django.views.generic.edit.FormMixin`

`django.views.generic.detail.SingleObjectMixin`

# ModelFormMixin

## Attributes

model - A model class. Can be explicitly provided, otherwise will be determined by examining self.object or queryset.

fields - A list of names of fields. This is interpreted the same way as the Meta.fields attribute of ModelForm.

This is a required attribute if you are generating the form class automatically (e.g. using model). Omitting this attribute will result in an ImproperlyConfigured exception.

success\_url - The URL to redirect to when the form is successfully processed.

success\_url may contain dictionary string formatting, which will be interpolated against the object's field attributes. For example, you could use success\_url="/polls/{slug}/" to redirect to a URL composed out of the slug field on a model.

# ModelFormMixin

Methods:-

`get_form_class()` - Retrieve the form class to instantiate. If `form_class` is provided, that class will be used. Otherwise, a `ModelForm` will be instantiated using the model associated with the queryset, or with the model, depending on which attribute is provided.

`get_form_kwargs()` - Add the current instance (`self.object`) to the standard `get_form_kwargs()`.

`get_success_url()` - Determine the URL to redirect to when the form is successfully validated. Returns `django.views.generic.edit.ModelFormMixin.success_url` if it is provided; otherwise, attempts to use the `get_absolute_url()` of the object.

`form_valid(form)` - Saves the form instance, sets the current object for the view, and redirects to `get_success_url()`.

# ModelFormMixin

Methods:-

form\_invalid(form) - Renders a response, providing the invalid form as context.

# Creating Model for CreateView

## models.py

```
from django.db import models  
from django.urls import reverse  
  
class Student(models.Model):  
    name = models.CharField(max_length=70)  
    roll = models.IntegerField()  
  
    def get_absolute_url(self):  
        return reverse("thankyou")  
  
    # return reverse("studentdetail", kwargs={"pk": self.pk})
```

## urls.py

```
urlpatterns = [    path('student/', views.StudentCreateView.as_view(), name='studentform'),  
    # path('model_detail/<int:pk>/', views.StudentDetailView.as_view(), name='studentdetail'),  
]
```

# CreateView with Default Template

## views.py

```
from django.views.generic.edit import CreateView  
from .models import Student  
class StudentCreateView(CreateView):  
    model = Student  
    fields = ('name', 'roll')      # fields = '__all__'
```

## urls.py

```
urlpatterns = [    path('student/', views.StudentCreateView.as_view(), name='studentform'),  
    # path('model_detail/<int:pk>/', views.StudentDetailView.as_view(), name='studentdetail'),  
]
```

Default Template should be: student\_form.html

# CreateView with Custom Template

## views.py

```
from django.views.generic.edit import CreateView  
from .models import Student  
class StudentCreateView(CreateView):  
    model = Student  
    fields = ('name', 'roll')  
    template_name = 'school/student.html'
```

## urls.py

```
urlpatterns = [    path('student/', views.StudentCreateView.as_view(), name='studentform'),  
    # path('model_detail/<int:pk>/', views.StudentDetailView.as_view(), name='studentdetail'),  
]
```

# UpdateView

`django.views.generic.edit.UpdateView`

A view that displays a form for editing an existing object, redisplaying the form with validation errors (if there are any) and saving changes to the object. This uses a form automatically generated from the object's model class (unless a form class is manually specified).

This view inherits methods and attributes from the following views:

- `django.views.generic.detail.SingleObjectTemplateResponseMixin`
- `django.views.generic.base.TemplateResponseMixin`
- `django.views.generic.edit.BaseUpdateView`
- `django.views.generic.edit.ModelFormMixin`
- `django.views.generic.edit.FormMixin`
- `django.views.generic.detail.SingleObjectMixin`
- `django.views.generic.edit.ProcessFormView`
- `django.views.generic.base.View`

# UpdateView

Attributes:-

template\_name\_suffix - The UpdateView page displayed to a GET request uses a template\_name\_suffix of '\_form'.

object - When using UpdateView you have access to self.object, which is the object being updated.

# UpdateView with Default Template

## views.py

```
from django.views.generic.edit import UpdateView  
from .models import Student  
class StudentUpdateView(UpdateView):  
    model = Student  
    fields = ('name', 'roll')
```

## urls.py

```
urlpatterns = [  
    path('updatestudent/<int:pk>', views.StudentUpdateView.as_view(), name='update_student'),  
]
```

Default Template should be: student\_form.html This is same as create template file.

# UpdateView with Custom Template

## views.py

```
from django.views.generic import UpdateView  
from .models import Student  
class StudentUpdateView(UpdateView):  
    model = Student  
    fields = ['name', 'roll']  
    template_name = 'school/student.html'
```

## urls.py

```
urlpatterns = [  
    path('updatestudent/<int:pk>', views.StudentUpdateView.as_view(), name='update_student'),  
]
```

Note The Create View Template File must be same

# DeleteView

`django.views.generic.edit.DeleteView`

A view that displays a confirmation page and deletes an existing object. The given object will only be deleted if the request method is POST. If this view is fetched via GET, it will display a confirmation page that should contain a form that POSTs to the same URL.

This view inherits methods and attributes from the following views:

- `django.views.generic.detail.SingleObjectTemplateResponseMixin`
- `django.views.generic.base.TemplateResponseMixin`
- `django.views.generic.edit.BaseDeleteView`
- `django.views.generic.edit.DeletionMixin`
- `django.views.generic.detail.BaseDetailView`
- `django.views.generic.detail.SingleObjectMixin`
- `django.views.generic.base.View`

# DeleteView

Attribute:-

template\_name\_suffix - The DeleteView page displayed to a GET request uses a template\_name\_suffix of '\_confirm\_delete'.

# DeletionMixin

`django.views.generic.edit.DeletionMixin`

Enables handling of the DELETE http action.

Attributes:-

`success_url` - The url to redirect to when the nominated object has been successfully deleted.

`success_url` may contain dictionary string formatting, which will be interpolated against the object's field attributes. For example, you could use `success_url="/parent/{parent_id}/"` to redirect to a URL composed out of the `parent_id` field on a model.

Methods:-

`delete(request, *args, **kwargs)` - Retrieves the target object and calls its `delete()` method, then redirects to the success URL.

`get_success_url()` - Returns the url to redirect to when the nominated object has been successfully deleted. Returns `success_url` by default.

# DeleteView with Default Template

## views.py

```
from django.views.generic.edit import DeleteView
From Django.urls import reverse_lazy
from .models import Student
class StudentDeleteView(DeleteView):
    model = Student
    success_url = reverse_lazy('add_student')          # After Delete Redirect to detail page
```

## urls.py

```
urlpatterns =
path('deletetestudent/<int:pk>', views.StudentDeleteView.as_view(), name='delete_student'),
]
```

Default Template should be: student\_confirm\_delete.html

# DeleteView with Default Template

## **Student\_confirm\_delete.py**

```
<body>
    <h1>Are you Sure ?</h1>
    <form action="" method="post">
        {% csrf_token %}
        <input type="submit" value="Delete">
        <a href="{% url 'add_student' %}">Cancel</a>
    </form>
</body>
```

# DeleteView with Custom Template

## views.py

```
from django.views.generic.edit import DeleteView
from django.urls import reverse_lazy
from .models import Student
class StudentDeleteView(DeleteView):
    model = Student
    success_url = reverse_lazy('add_student')          # After Delete Redirect to detail page
    template_name = 'school/studentdeleteconfirmation.html'
```

## urls.py

```
urlpatterns =
path('deletetestudent/<int:pk>', views.StudentDeleteView.as_view(), name='delete_student'),
```

# CBV and FBV

CBV

It is not powerful.

Its Wrapper for FBV to hide complexity

This is used in most generic view

Flow of execution is unexpected the flow is defined internally

For Get Request define get() and for POST define post ()

Its easy to use and have reusability.

# Model Related Class Based View

ListView – To List all Record. This will look for default template file named ModelClassName\_list.html and default context object named ModelClassName\_list. We can also specific our own template and context file using template\_name = student.html as well context by context\_object\_name = list\_of\_student

DetailView – To get details of a particular Record. This will look for default template file named ModelClassName\_detail.html and default context object named ModelClassName or object. We can also specific our own template and context file using template\_name = student.html as well context by context\_object\_name = list\_of\_student

CreateView – To Create Record

This will look for default template file named ModelClassName\_form.html. We can also specific our own template and context file using template\_name = studentform.html. We also need to define get\_absolute\_url so it will redirected after insertion.

# Model Related Class Based View

UpdateView – To Update Record. This is very similar to create infact it will use CreateView default template file. This will look for default template file named ModelClassName\_detail.html . We can also specific our own template and context file using template\_name = studentform.html. We also need to define get\_absolute\_url so it will redirected after insertion.

DeleteView – To Delete Record

This will look for default template file named ModelClassName\_confirm\_delete.html and default context object named ModelClassName or object. We can also specific our own template file using template\_name = studentdeleteconfirmation.html

# 56-Authentication Views

Django provides several views that you can use for handling login, logout, and password management.

These make use of the stock auth forms but you can pass in your own forms as well.

Django provides no default template for the authentication views.

You should create your own templates for the views you want to use.

## urls.py

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

These all url will be available:-

accounts/login/ [name='login']

accounts/logout/ [name='logout']

accounts/password\_change/ [name='password\_change']

accounts/password\_change/done/ [name='password\_change\_done']

accounts/password\_reset/ [name='password\_reset']

accounts/password\_reset/done/ [name='password\_reset\_done']

accounts/reset/<uidb64>/<token>/ [name='password\_reset\_confirm']

accounts/reset/done/ [name='password\_reset\_complete']

```
=urlpatterns = [
    path('login/', views.LoginView.as_view(), name='login'),
    path('logout/', views.LogoutView.as_view(), name='logout'),

    path('password_change/', views.PasswordChangeView.as_view(), name='password_change'),
    path('password_change/done/', views.PasswordChangeDoneView.as_view(), name=
        'password_change_done'),

    path('password_reset/', views.PasswordResetView.as_view(), name='password_reset'),
    path('password_reset/done/', views.PasswordResetDoneView.as_view(), name=
        'password_reset_done'),
    path('reset/<uidb64>/<token>/', views.PasswordResetConfirmView.as_view(), name=
        'password_reset_confirm'),
    path('reset/done/', views.PasswordResetCompleteView.as_view(), name='password_reset_complete'
),
]
```

# Authentication Views

```
urlpatterns = [
    path('login/', views.LoginView.as_view(), name='login'),
    path('logout/', views.LogoutView.as_view(), name='logout'),

    path('password_change/', views.PasswordChangeView.as_view(), name='password_change'),
    path('password_change/done/', views.PasswordChangeDoneView.as_view(), name=
        'password_change_done'),

    path('password_reset/', views.PasswordResetView.as_view(), name='password_reset'),
    path('password_reset/done/', views.PasswordResetDoneView.as_view(), name=
        'password_reset_done'),
    path('reset/<uidb64>/<token>/', views.PasswordResetConfirmView.as_view(), name=
        'password_reset_confirm'),
    path('reset/done/', views.PasswordResetCompleteView.as_view(), name='password_reset_complete'
        ),
]
```

# login\_required Decorator

`login_required(redirect_field_name='next', login_url=None)`

If the user is logged in, execute the view normally. The view code is free to assume the user is logged in.

If the user isn't logged in, redirect to `settings.LOGIN_URL`, passing the current absolute path in the query string. Example: `/accounts/login/?next=/accounts/profile/`

`django.contrib.auth.decorators.login_required`

Where,

`redirect_field_name` - If you would prefer to use a different name for this parameter, `login_required()` takes an optional `redirect_field_name` parameter. If you provide a value to `redirect_field_name`, you will most likely need to customize your login template as well, since the template context variable which stores the redirect path will use the value of `redirect_field_name` as its key rather than "next" (the default).

`login_url` - If you don't specify the `login_url` parameter, you'll need to ensure that the `settings.LOGIN_URL` and your login view are properly associated.

The `settings.LOGIN_URL` also accepts view function names and named URL patterns. This allows you to freely remap your login view within your URLconf without having to update the setting.

# **login\_required Decorator**

Example:-

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

# **staff\_member\_required decorator**

```
staff_member_required(redirect_field_name='next', login_url='admin:login')
```

This decorator is used on the admin views that require authorization. A view decorated with this function will have the following behavior:

- If the user is logged in, is a staff member (`User.is_staff=True`), and is active (`User.is_active=True`), execute the view normally.
- Otherwise, the request will be redirected to the URL specified by the `login_url` parameter, with the originally requested path in a query string variable specified by `redirect_field_name`.

For example: /admin/login/?next=/profile/

Example:-

```
from django.contrib.admin.views.decorators import staff_member_required  
  
@staff_member_required  
  
def profile(request):  
  
    return render(request, 'registration/profile.html')
```

# permission\_required Decorator

```
permission_required(perm, login_url=None, raise_exception=False)
```

It's a relatively common task to check whether a user has a particular permission. For that reason, Django provides a shortcut for that case: the `permission_required()` decorator.

Just like the `has_perm()` method, permission names take the form “<app label>.<permission codename>”

Example:-

```
from django.contrib.auth.decorators import permission_required

@permission_required('blog.can_add')
def profile(request):
    return render(request, 'registration/profile.html')
```

# Decorating Class-Based View

## Decorating in urls.py or URLconf

The simplest way of decorating class-based views is to decorate the result of the `as_view()` method. The easiest place to do this is in the URLconf where you deploy your view:

```
from django.urls import path
from django.views.generic import TemplateView
from registration.views import ProfileTemplateView
from django.contrib.auth.decorators import login_required

urlpatterns = [
    path('dashboard/', login_required(TemplateView.as_view(template_name='blog/dash.html')), name='dash'),
    path('profile/', login_required(ProfileTemplateView.as_view(template_name='registration/profile.html')), name='profile'),
    path('blogpost/', permission_required('blog.can_add')(BlogPostView.as_view())),
]
```

# method\_decorator

The `method_decorator` decorator transforms a function decorator into a method decorator so that it can be used on an instance method.

A method on a class isn't quite the same as a standalone function, so you can't just apply a function decorator to the method you need to transform it into a method decorator first.

```
@method_decorator(*args, **kwargs)
```

# Decorating Class-Based View

## Decorating in the Class

To decorate every instance of a class-based view, you need to decorate the class definition itself. To do this you apply the decorator to the `dispatch()` method of the class.

```
from django.utils.decorators import method_decorator
```

```
from django.views.generic import TemplateView
```

```
class ProfileTemplateView(TemplateView):
```

```
    template_name = 'registration/profile.html'
```

```
    @method_decorator(login_required)
```

```
    def dispatch(self, *args, **kwargs):
```

```
        return super().dispatch(*args, **kwargs)
```

# Decorating Class-Based View

## Decorating in the Class

You can decorate the class instead and pass the name of the method to be decorated as the keyword argument name:

```
@method_decorator(login_required, name='dispatch')  
class ProfileTemplateView(TemplateView):  
    template_name = 'registration/profile.html'
```

# Decorating Class-Based View

## Decorating in the Class

If you have a set of common decorators used in several places, you can define a list or tuple of decorators and use this instead of invoking method\_decorator() multiple times.

```
@method_decorator(never_cache, name='dispatch')  
@method_decorator(login_required, name='dispatch')  
  
class ProfileTemplateView(TemplateView):  
    template_name = 'registration/profile.html'  
  
  
decorators = [never_cache, login_required]  
@method_decorator(decorators, name='dispatch')  
  
class ProfileTemplateView(TemplateView):  
    template_name = 'registration/profile.html'
```

The decorators will process a request in the order they are passed to the decorator. In the example, never\_cache() will process the request before login\_required().

# Authentication Views

This is a list with all the views `django.contrib.auth` provides.

- `LoginView`
- `LogoutView`
- `PasswordChangeView`
- `PasswordChangeDoneView`
- `PasswordResetView`
- `PasswordResetDoneView`
- `PasswordResetConfirmView`
- `PasswordResetCompleteView`

# LoginView

*django.contrib.auth.views.LoginView*

```
template_name = 'registration/login.html'  
path('login/', views.LoginView.as_view(), name='login')
```

It's your responsibility to provide the html for the login template , called registration/login.html by default.

- If called via GET, it displays a login form that POSTs to the same URL.
- If called via POST with user submitted credentials, it tries to log the user in.
- If login is successful, the view redirects to the URL specified in next.
- If next isn't provided, it redirects to settings.LOGIN\_REDIRECT\_URL (which defaults to /accounts/profile/).
- If login isn't successful, it redisplays the login form.

# LoginView

Default Template:-

registration/login.html

This template gets passed four template context variables:

- form: A Form object representing the AuthenticationForm.
- next: The URL to redirect to after successful login. This may contain a query string, too.
- site: The current Site, according to the SITE\_ID setting. If you don't have the site framework installed, this will be set to an instance of RequestSite, which derives the site name and domain from the current HttpRequest.
- site\_name: An alias for site.name. If you don't have the site framework installed, this will be set to the value of request.META['SERVER\_NAME'].

Custom Template:-

```
path('login/', views.LoginView.as_view(template_name='myapp/login.html'), name='login')
```

# LoginView

## Attributes:-

template\_name: The name of a template to display for the view used to log the user in. Defaults to registration/login.html.

redirect\_field\_name: The name of a GET field containing the URL to redirect to after login. Defaults to *next*.

authentication\_form: A callable (typically a form class) to use for authentication. Defaults to AuthenticationForm.

extra\_context: A dictionary of context data that will be added to the default context data passed to the template.

redirect\_authenticated\_user: A boolean that controls whether or not authenticated users accessing the login page will be redirected as if they had just successfully logged in. Defaults to False.

success\_url\_allowed\_hosts: A set of hosts, in addition to request.get\_host(), that are safe for redirecting after login. Defaults to an empty set.

# LogoutView

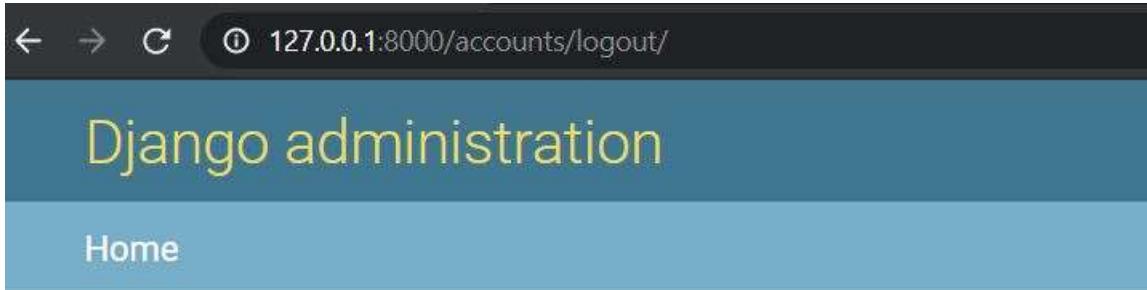
*django.contrib.auth.views.LogoutView*

template\_name = 'registration/logout\_out.html'

path('logout/', views.LogoutView.as\_view(), name='logout')

logged\_out.html template is already available and used by admin logout template file.

You can create your own custom logout template by defining template\_name attribute.



Logged out

Thanks for spending some quality time with the Web site today.

[Log in again](#)

# LogoutView

## Default Template:-

registration/logged\_out.html

This template gets passed three template context variables:

- title: The string “Logged out”, localized.
- site: The current Site, according to the SITE\_ID setting. If you don’t have the site framework installed, this will be set to an instance of RequestSite, which derives the site name and domain from the current HttpRequest.
- site\_name: An alias for site.name. If you don’t have the site framework installed, this will be set to the value of request.META['SERVER\_NAME'].

## Custom Template:-

```
path('logout/', views.LogoutView.as_view(template_name='myapp/logout.html'), name='logout')
```

# LogoutView

## Attributes:-

next\_page: The URL to redirect to after logout. Defaults to settings.LOGOUT\_REDIRECT\_URL.

template\_name: The full name of a template to display after logging the user out. Defaults to registration/logged\_out.html.

redirect\_field\_name: The name of a GET field containing the URL to redirect to after log out. Defaults to next. Overrides the next\_page URL if the given GET parameter is passed.

extra\_context: A dictionary of context data that will be added to the default context data passed to the template.

success\_url\_allowed\_hosts: A set of hosts, in addition to request.get\_host(), that are safe for redirecting after logout. Defaults to an empty set.

# PasswordChangeView

*django.contrib.auth.views.PasswordChangeView*

template\_name = 'registration/password\_change\_form.html'

path('password\_change/', views.PasswordChangeView.as\_view(), name='password\_change'),

registration/password\_change\_form.html template is already available and used by admin change password template file.

You can create your own custom change password template by defining template\_name attribute.

127.0.0.1:8000/accounts/password\_change/

Django administration

Home > Password change

Password change

Please enter your old password, for security's sake, and then enter your new password twice so we can verify you typed it in correctly.

Old password:

New password:

Your password can't be too similar to your other personal information.  
Your password must contain at least 8 characters.  
Your password can't be a commonly used password.  
Your password can't be entirely numeric.

New password confirmation:

CHANGE MY PASSWORD

# PasswordChangeView

## Default Template:-

registration/password\_change\_form.html

This template gets passed following template context variables:

- form: The password change form

## Custom Template:-

```
path('password_change/',
      views.PasswordChangeView.as_view(template_name='myapp/changepass.html'),
      name='password_change'),
```

# PasswordChangeView

## Attributes:-

template\_name: The full name of a template to use for displaying the password change form. Defaults to registration/password\_change\_form.html if not supplied.

success\_url: The URL to redirect to after a successful password change. Defaults to 'password\_change\_done'.

form\_class: A custom “change password” form which must accept a user keyword argument. The form is responsible for actually changing the user’s password. Defaults to PasswordChangeForm.

extra\_context: A dictionary of context data that will be added to the default context data passed to the template.

# PasswordChangeDoneView

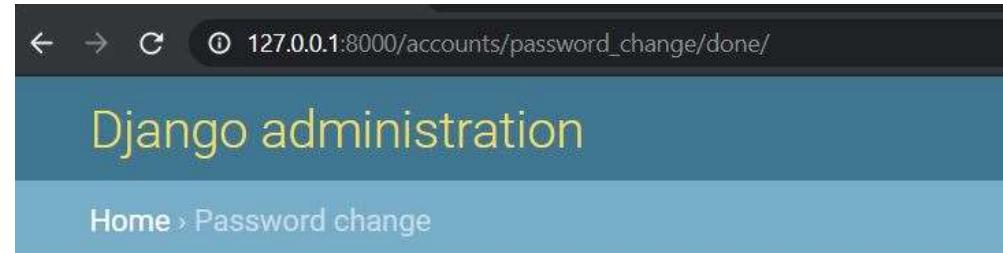
*django.contrib.auth.views.PasswordChangeDoneView*

template\_name = 'registration/password\_change\_done.html'

path('password\_change/done/', views.PasswordChangeDoneView.as\_view(),  
name='password\_change\_done'),

registration/password\_change\_done.html template is already available and used by admin password change done template file.

You can create your own custom password change done template by defining template\_name attribute.



Password change successful

Your password was changed.

# PasswordChangeDoneView

Default Template:-

registration/password\_change\_done.html

Custom Template:-

```
path('password_change/done/',
      views.PasswordChangeDoneView.as_view(template_name='myapp/changepassdone.html'),
      name='password_change_done'),
```

# PasswordChangeDoneView

## Attributes:-

template\_name: The full name of a template to use. Defaults to registration/password\_change\_done.html if not supplied.

extra\_context: A dictionary of context data that will be added to the default context data passed to the template.

# PasswordResetView

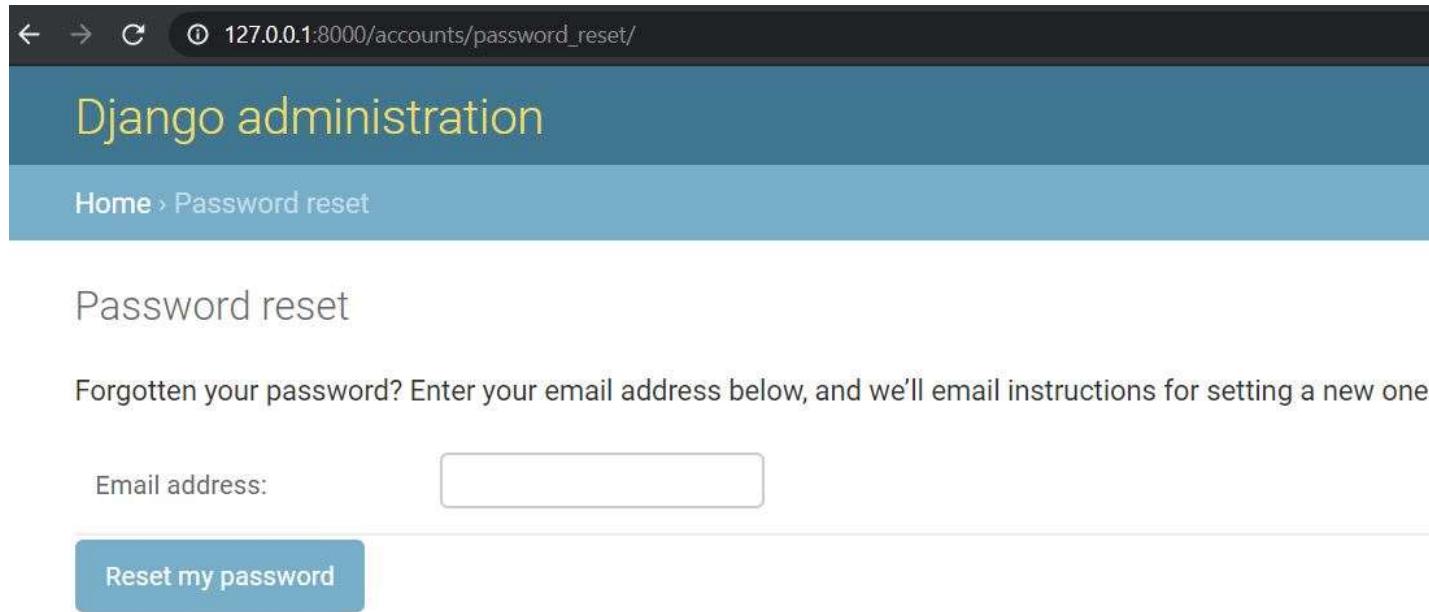
*django.contrib.auth.views.PasswordResetView*

template\_name = 'registration/password\_reset\_form.html'

path('password\_reset/', views.PasswordResetView.as\_view(), name='password\_reset'),

registration/password\_reset\_form.html template is already available and used by admin password reset template file.

You can create your own custom password reset template by defining template\_name attribute.



# PasswordResetView

## Default Template:-

registration/password\_reset\_form.html

This template gets passed following template context variables:

- form: The form for resetting the user's password.

## Custom Template:-

```
path('password_reset/', views.PasswordResetView.as_view(template_name='myapp/resetpass.html'),  
      name='password_reset'),
```

# PasswordResetView

This Email template gets passed following email template context variables:

- email: An alias for user.email
- user: The current User, according to the email form field. Only active users are able to reset their passwords (User.is\_active is True).
- site\_name: An alias for site.name. If you don't have the site framework installed, this will be set to the value of request.META['SERVER\_NAME']. For more on sites, see The “sites” framework.
- domain: An alias for site.domain. If you don't have the site framework installed, this will be set to the value of request.get\_host().
- protocol: http or https
- uid: The user's primary key encoded in base 64.
- token: Token to check that the reset link is valid.

# PasswordResetView

## Attributes:-

template\_name: The full name of a template to use for displaying the password reset form. Defaults to registration/password\_reset\_form.html if not supplied.

form\_class: Form that will be used to get the email of the user to reset the password for. Defaults to PasswordResetForm.

email\_template\_name: The full name of a template to use for generating the email with the reset password link. Defaults to registration/password\_reset\_email.html if not supplied.

subject\_template\_name: The full name of a template to use for the subject of the email with the reset password link. Defaults to registration/password\_reset\_subject.txt if not supplied.

token\_generator: Instance of the class to check the one time link. This will default to default\_token\_generator, it's an instance of django.contrib.auth.tokens.PasswordResetTokenGenerator.

success\_url: The URL to redirect to after a successful password reset request. Defaults to 'password\_reset\_done'.

# PasswordResetView

## Attributes:-

from\_email: A valid email address. By default Django uses the DEFAULT\_FROM\_EMAIL.

extra\_context: A dictionary of context data that will be added to the default context data passed to the template.

html\_email\_template\_name: The full name of a template to use for generating a text/html multipart email with the password reset link. By default, HTML email is not sent.

extra\_email\_context: A dictionary of context data that will be available in the email template. It can be used to override default template context values listed below e.g. domain.

# PasswordResetDoneView

*django.contrib.auth.views.PasswordResetDoneView*

The page shown after a user has been emailed a link to reset their password. This view is called by default if the PasswordResetView doesn't have an explicit success\_url URL set.

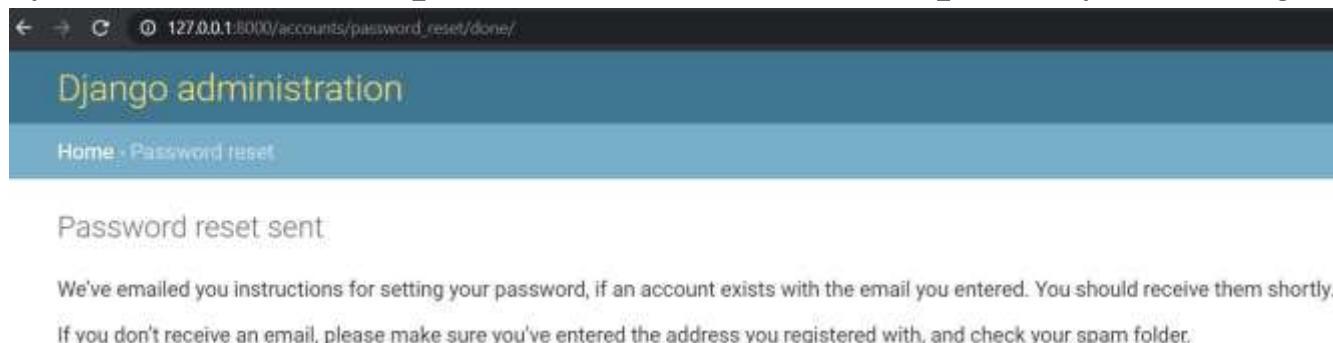
If the email address provided does not exist in the system, the user is inactive, or has an unusable password, the user will still be redirected to this view but no email will be sent.

template\_name = 'registration/password\_reset\_done.html'

```
path('password_reset/done/', views.PasswordResetDoneView.as_view(),  
      name='password_reset_done'),
```

registration/password\_reset\_done.html template is already available and used by admin password reset done template file.

You can create your own custom password reset done template by defining template\_name attribute.



# **PasswordResetDoneView**

Default Template:-

registration/password\_reset\_done.html

Custom Template:-

```
path('password_reset/done/',
views.PasswordResetDoneView.as_view(template_name='myapp/resetpassdone.html'),
name='password_reset_done'),
```

# **PasswordResetDoneView**

## Attributes:-

template\_name: The full name of a template to use. Defaults to registration/password\_reset\_done.html if not supplied.

extra\_context: A dictionary of context data that will be added to the default context data passed to the template.

# PasswordResetConfirmView

*django.contrib.auth.views.PasswordResetConfirmView*

It presents a form for entering a new password.

Keyword arguments from the URL:

- uidb64: The user's id encoded in base 64.
- token: Token to check that the password is valid.

template\_name = 'registration/password\_reset\_confirm.html'

path('reset/<uidb64>/<token>', views.PasswordResetConfirmView.as\_view(),  
name='password\_reset\_confirm'),

registration/password\_reset\_confirm.html template is already available and used by admin password  
reset confirm template file.

You can create your own custom password reset confirm template by defining template\_name  
attribute.

# PasswordResetConfirmView

## Default Template:-

registration/password\_reset\_confirm.html

This template gets passed following template context variables:

- form: The form for setting the new user's password.
- validlink: Boolean, True if the link (combination of uidb64 and token) is valid or unused yet.

## Custom Template:-

```
path('reset/<uidb64>/<token>/',  
views.PasswordResetConfirmView.as_view(template_name='myapp/resetpassconfirm.html'),  
name='password_reset_confirm'),
```

# PasswordResetConfirmView

## Attributes:-

template\_name: The full name of a template to display the confirm password view. Default value is registration/password\_reset\_confirm.html.

token\_generator: Instance of the class to check the password. This will default to default\_token\_generator, it's an instance of django.contrib.auth.tokens.PasswordResetTokenGenerator.

post\_reset\_login: A boolean indicating if the user should be automatically authenticated after a successful password reset. Defaults to False.

post\_reset\_login\_backend: A dotted path to the authentication backend to use when authenticating a user if post\_reset\_login is True. Required only if you have multiple AUTHENTICATION\_BACKENDS configured. Defaults to None.

form\_class: Form that will be used to set the password. Defaults to SetPasswordForm.

success\_url: URL to redirect after the password reset done. Defaults to 'password\_reset\_complete'.

extra\_context: A dictionary of context data that will be added to the default context data passed to the template.

# **PasswordResetConfirmView**

## Attributes:-

reset\_url\_token: Token parameter displayed as a component of password reset URLs. Defaults to 'set-password'.

# PasswordResetCompleteView

*django.contrib.auth.views.PasswordResetCompleteView*

It presents a view which informs the user that the password has been successfully changed.

template\_name = 'registration/password\_reset\_complete.html'

path('reset/done/', views.PasswordResetCompleteView.as\_view(), name='password\_reset\_complete'),

registration/password\_reset\_complete.html template is already available and used as admin password reset complete template file.

You can create your own custom password reset complete template by defining template\_name attribute.

# **PasswordResetCompleteView**

Default Template:-

registration/password\_reset\_complete.html

Custom Template:-

```
path('reset/done/',
      views.PasswordResetCompleteView.as_view(template_name='myapp/resetpasscomp.html'),
      name='password_reset_complete'),
```

# **PasswordResetCompleteView**

## Attributes:-

template\_name: The full name of a template to display the view. Defaults to registration/password\_reset\_complete.html.

extra\_context: A dictionary of context data that will be added to the default context data passed to the template.

# Built-in Forms

- AdminPasswordChangeForm - A form used in the admin interface to change a user's password. It takes the user as the first positional argument.
- AuthenticationForm - A form for logging a user in. It takes request as its first positional argument, which is stored on the form instance for use by sub-classes.
- PasswordChangeForm - A form for allowing a user to change their password.
- PasswordResetForm - A form for generating and emailing a one-time use link to reset a user's password.
- SetPasswordForm - A form that lets a user change their password without entering the old password.
- UserChangeForm - A form used in the admin interface to change a user's information and permissions.
- UserCreationForm - A ModelForm for creating a new user.

# 57-Auth Settings

Settings for django.contrib.auth.

## AUTHENTICATION\_BACKENDS

A list of authentication backend classes (as strings) to use when attempting to authenticate a user.

Default: ['django.contrib.auth.backends.ModelBackend']

## AUTH\_USER\_MODEL

The model to use to represent a User.

Default: 'auth.User'

## LOGIN\_REDIRECT\_URL

The URL or named URL pattern where requests are redirected after login when the LoginView doesn't get a next GET parameter.

Default: '/accounts/profile/'

# Auth Settings

## `LOGIN_URL`

The URL or named URL pattern where requests are redirected for login when using the `login_required()` decorator, `LoginRequiredMixin`, or `AccessMixin`.

Default: `'/accounts/login/'`

## `LOGOUT_REDIRECT_URL`

The URL or named URL pattern where requests are redirected after logout if `LogoutView` doesn't have a `next_page` attribute.

If `None`, no redirect will be performed and the logout view will be rendered.

Default: `None`

# Auth Settings

## PASSWORD\_RESET\_TIMEOUT

The number of seconds a password reset link is valid for. Depending on when the link is generated, it will be valid for up to a day longer.

Used by the PasswordResetConfirmView.

Default: 259200 (3 days, in seconds)

## PASSWORD\_HASHERS

Default: [

```
'django.contrib.auth.hashers.PBKDF2PasswordHasher',
'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
'django.contrib.auth.hashers.Argon2PasswordHasher',
'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
```

]

# Auth Settings

## AUTH\_PASSWORD\_VALIDATORS

The list of validators that are used to check the strength of user's passwords.

Default: [ ] (Empty list)

# 58-SQLite3

Open settings.py

```
DATABASE = {  
    'default': {  
        'ENGINE': 'django.db.backends.sqlite3',  
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),  
    },  
}
```

# MySQL Configuration

- Install mysql in your system.
- You have to create your own database and user to config MySQL with Django.
- Open settings.py

```
DATABASE = {  
    'default': {  
        'ENGINE': 'django.db.backends.mysql',  
        'NAME': 'database_name',  
        'USER': 'user_name',  
        'PASSWORD': 'password',  
        'HOST': 'localhost',  
        'PORT': port_number    } } } }
```

```
DATABASE = {  
    'default': {  
        'ENGINE': 'django.db.backends.mysql',  
        'NAME': 'mydb',  
        'USER': 'root',  
        'PASSWORD': '',  
        'HOST': 'localhost',  
        'PORT': 5555    } }
```

# Oracle SQL Configuration

- Install Oracle in your system.
- You have to create your own database and user to config Oracle with Django.
- Open settings.py

```
DATABASE = {
```

```
    'default': {  
        'ENGINE': 'django.db.backends.oracle',  
        'NAME': 'database_name'  
        'USER': 'user_name',  
        'PASSWORD': 'password',  
        'HOST': 'localhost',  
        'PORT': port_number  
    }  
}
```

# Postgres Configuration

- Install Postgres in your system.
- You have to create your own database and user to config Postgres with Django.
- Open settings.py

```
DATABASE = {
```

```
    'default': {  
        'ENGINE': 'django.db.backends.postgres',  
        'NAME': 'database_name'  
        'USER': 'user_name',  
        'PASSWORD': 'password',  
        'HOST': 'localhost',  
        'PORT': port_number  
    }  
}
```

# 59-Pagination

Using pagination we can split data to several pages, with Previous/Next links.



Django provides a few classes that help you manage paginated data:-

- Paginator Class
- Page Class

# Paginator Class

```
class Paginator(object_list, per_page, orphans=0, allow_empty_first_page=True)
```

Where,

object\_list – It takes tuple, list, QuerySet or other sliceable object with a count() or \_\_len\_\_() method. It is required.

per\_page – The maximum number of items to include on a page, not including orphans. It is required.

orphans – Use this when you don't want to have a last page with very few items. If the last page would normally have a number of items less than or equal to orphans, then those items will be added to the previous page (which becomes the last page) instead of leaving the items on a page by themselves. orphans defaults to zero, which means pages are never combined and the last page may have one item. It is optional.

allow\_empty\_first\_page - Whether or not the first page is allowed to be empty. If False and object\_list is empty, then an EmptyPage error will be raised. It is optional.

# Pagination Class Attributes

- count - The total number of objects, across all pages.
- num\_pages - The total number of pages.
- page\_range - A 1-based range iterator of page numbers, e.g. yielding [1, 2, 3, 4].

# Pagination Class Methods

- `get_page(number)` – This method returns a `Page` object with the given 1-based index, while also handling out of range and invalid page numbers.  
If the page isn't a number, it returns the first page.  
If the page number is negative or greater than the number of pages, it returns the last page.  
Raises an `EmptyPage` exception only if you specify `Paginator(..., allow_empty_first_page=False)` and the `object_list` is empty.
- `page(number)` – This method returns a `Page` object with the given 1-based index.  
Raises `InvalidPage` if the given page number doesn't exist.

# Page Class

```
class Page(object_list, number, paginator)
```

A page acts like a sequence of Page.object\_list when using len() or iterating it directly.

# Page Class Attributes

- object\_list - The list of objects on this page.
- number - The 1-based page number for this page.
- paginator - The associated Paginator object.

# Page Class Methods

- `has_next()` - It returns True if there's a next page.
- `has_previous()` - It returns True if there's a previous page.
- `has_other_pages()` - It returns True if there's a next or previous page.
- `next_page_number()` - It returns the next page number. Raises `InvalidPage` if next page doesn't exist.
- `previous_page_number()` - It returns the previous page number. Raises `InvalidPage` if previous page doesn't exist.
- `start_index()` - It returns the 1-based index of the first object on the page, relative to all of the objects in the paginator's list. For example, when paginating a list of 5 objects with 2 objects per page, the second page's `start_index()` would return 3.
- `end_index()` - It returns the 1-based index of the last object on the page, relative to all of the objects in the paginator's list. For example, when paginating a list of 5 objects with 2 objects per page, the second page's `end_index()` would return 4.

# Using Pagination

- Pagination with Function Based View
- Pagination with Class Based View



# **60-Filters**

When we need to modify variable before displaying we can use filters. Pipe ‘|’ is used to apply filter.

Syntax:- { {variable | filter} }

Example:- { {name|upper} }

- Built-in Filters – upper, truncateword:40,
- Custom Filters

# Custom Filters

Custom filters are Python functions.

- Create Filter's Function
- Register Filter
- Use Filter in Template

# Geeky Steps

- Create a Folder inside Application named:- **templatetags**
- Create `__init__.py` File inside templatetags Folder to ensure the Folder is treated as a Python package.
- Create Modules as many as you need, inside templatetags package and write code for custom filter

Note - After adding the templatetags module, you will need to restart your server before you can use the tags or filters in templates.

# Custom Filters

## Creating Filter's Function:-

```
def function_name(value, arg):  
    return value.replace(arg, '')
```

## Register Filter:-

```
from django import template  
  
register = template.Library()  
  
register.filter("filter_name", filter_function)
```

## Creating Filter's Function And Registering:-

```
@register.filter(name="filter_name")  
  
def function_name(value, arg):  
    return value.replace(arg, '')
```

```
@register.filter  
  
def function_name(value, arg):  
    return value.replace(arg, '')
```

Note- If you leave off the name argument, Django will use the function's name as the filter name.

# Custom Filters

## How to use Custom Filter in Templates:-

- Load the filter module in Template using load Tag

```
{% load filter_module_name %}
```

- Use Filter

```
{{ variable|filter_name }}
```

```
{{ variable|filter_name:argument }}
```

# 61-Django Security

- Cross Site Scripting (XSS) Protection
- Cross Site Request Forgery (CSRF) Protection
- SQL Injection Protection
- ClickJacking Protection
- SSL/HTTPS
- Host header validation
- Referrer Policy
- Session security
- User-Uploaded Content

# Django Security

- Make sure that your Python code is outside of the Web server's root. This will ensure that your Python code is not accidentally served as plain text (or accidentally executed).
- Take care with any user uploaded files.
- Django does not throttle requests to authenticate users. To protect against brute-force attacks against the authentication system, you may consider deploying a Django plugin or Web server module to throttle these requests.
- Keep your SECRET\_KEY a secret.
- It is a good idea to limit the accessibility of your caching system and database using a firewall.

# 62-What Next ?

- Update Yourself by reading Django Doc and other Resources
- Build Applications
- Apply for Job or Internship
- Try to use various packages with Django - [www.pypi.org](http://www.pypi.org)
- Learn Django REST Framework
- Learn other Web App Framework/Library – Flask, Web2Py, TurboGears Pyramid
- Job Opportunities in Django are more as compared to Flask.

# Django 3.1

- Django 3.1 supports Python 3.6, 3.7, and 3.8.
- Asynchronous views and middleware support
- JSONField for all supported database backends
- DEFAULT\_HASHING\_ALGORITHM settings
- Django 3.1 supports MariaDB 10.2 and higher.
- The admin no longer supports the legacy Internet Explorer browser.
- The admin now has a sidebar on larger screens for easier navigation.
- The settings.py generated by the startproject command now uses pathlib.Path instead of os.path for building filesystem paths.

# Features Deprecated in 3.1

- `PASSWORD_RESET_TIMEOUT_DAYS` setting is deprecated in favor of `PASSWORD_RESET_TIMEOUT`.
- The `HttpRequest.is_ajax()` method is deprecated.
- `providing_args` argument for `Signal` is deprecated.
- The passing of URL kwargs directly to the context by `TemplateView` is deprecated. Reference them in the template with `view.kwargs` instead.
- The `NullBooleanField` model field is deprecated in favor of `BooleanField(null=True)`.

# Asynchronous Views and Middleware

```
# views.py  
async def my_view(request):  
    await asyncio.sleep(0.5)  
    return HttpResponse('Hello, async world!')
```

All asynchronous features are supported whether you are running under WSGI or ASGI mode.

If you make External HTTP calls from a view, You can use `async View`.

The ORM, cache layer, and several other parts of code that involve long-running network calls do not support `async` yet.

# Pathlib

The settings.py generated by the startproject command now uses pathlib.Path instead of os.path for building filesystem paths.

```
# settings.py

import os

BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))

DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.sqlite3',

        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),

    }

}

BASE_DIR = Path(__file__).resolve(strict=True).parent.parent

'NAME': BASE_DIR / 'db.sqlite3',
```

# **The admin now has a sidebar**

- The admin now has a sidebar on larger screens for easier navigation.

# PASSWORD\_RESET\_TIMEOUT

- PASSWORD\_RESET\_TIMEOUT\_DAYS = 3
- PASSWORD\_RESET\_TIMEOUT = 5000

# Templates inside Project

## settings.py

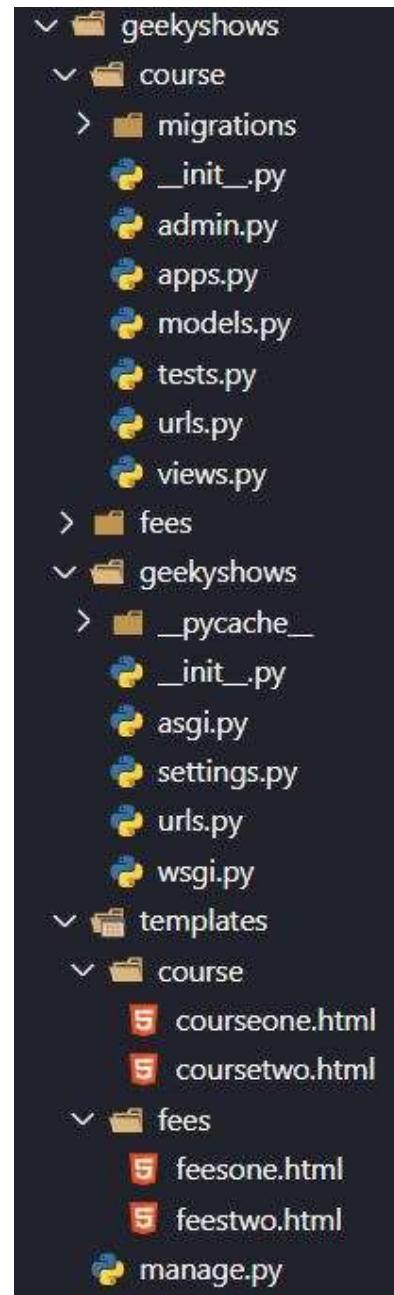
Old Django Version 3.0

```
TEMPLATES_DIR = os.path.join(BASE_DIR, 'templates')
```

```
TEMPLATES_DIR = BASE_DIR / 'templates'
```

```
INSTALLED_APPS = [
    'course',
    'fees'
]
```

```
TEMPLATES = [
    {
        'DIRS': [TEMPLATES_DIR],
        'APP_DIRS': True,
    }
]
```



# Static inside Project

## settings.py

```
TEMPLATES_DIR = BASE_DIR / 'templates'
```

Old Django Version 3.0

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

```
STATIC_DIR = BASE_DIR / 'static'
```

```
INSTALLED_APPS = [
    'course'
]
```

```
TEMPLATES = [
    {
        'DIRS': [TEMPLATES_DIR]
    }
]
```

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

