1 Table of Contents

1.1 DJANGO INTRODUCTION

Django is based on MVT architecture i.e Model-View-Template.
Model- Act as the interface of the data and helps maintaining the data.
View- User interface
Template - Contains static parts of the desired HTML output.

1.1.1 INSTALLING DJANGO

Prerequisites python installed
1.Install pip
python -m pip install -U pip

1.1.2 SETTING THE DEVELOPMENT OR VIRTUAL ENVIRONMENT

1.python3 -m venv <name>
i.e. python3 -m venv venv
2. source ./venv/bin/activate
3. pip install django

OR;

1. pip install virtualenv
2. virtualenv env
3. source env/bin/activate
4. pip install django

1.1.3 STARTING THE PROJECT

1. django-admin startproject projectName
2. cd projectName
3. python manage.py runserver
4. python manage.py help(To list all the commands that can be executed by manage.py)

1.1.4 CREATING APP

1. python manage.py startapp appname

i.e. python manage.py startapp post

And put it in the installed apps of settings.py of main app

2.Render the app using URLS we need to include the app in our main project so that URLS redirected to that app can be rendered.

In main project:

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

EXAMPLE:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
   path('admin/', admin.site.urls),
   # Enter the app name in following
   # syntax for this to work
   path('', include("projectApp.urls")),
]
```

Then create a file name urls.py in created app,

```
from django.urls import path
from . import views
        urlpatters=[
        path('',views.index)
]
```

Then in views.py of created app include:

```
from django.http import HttpResponse

def index(request):
   return HttpResponse("Hello Nischal")
```

After adding the code,settings.py of main project directory and change the value of :

```
    ROOT_URLCONF = 'app.urls'
```

Finally,We can run the server(127.0.0.1:8000)

1.2 DJANGO VIEWS (User interface)

View function is a python function that takes a web request and returns a web response i.e HTML contents of a web page,a redirect ,a 404 error,an XML document,an image or anything web can display.

1.2.1 WEB REQUEST AND WEB REPSONSE

Middleware:It is a middle ground between a request and response like a window through which data passes.

Request and response object:It helps to pass the state through the system.When a page is requested,Django creates an HttpRequest object that contains metadata about the request.Then Django loads the appropraiate view,passing the HttpRequest as the first argument to the view function .Each view is responsible for returning an HttpResponse object.

Example:In views.py

```
# importing HttResponse from library
from django.http import HttpResponse

def home(request):
    # request is handled using HttpResponse object
    return HttpResponse("Any kind of HTML Here")

    In urls.py
# importing view from views.py
from .views import home

urlpatterns = [
    path('', home),
```

]

There are many HttpRequest and HttpResponse attributes and method which can be used :
https://www.geeksforgeeks.org/django-request-and-response-cycle-httprequest-and-httpresponse-objects/

1.2.2 DJANGO CLASS BASED VIEWS AND FUNCTION BASED VIEWS

Views are divided into 2 categories:

1. Function Based Django Views

It generally a python function that takes HttpRequest object as an argument and Returns a HttpResponse object.
They are generally divided into 4 basic strategies:CRUD
CREATE,RETRIEVE,UPDATE AND DELETE .It is the base of any framework for development.

USE OF FUNCTION BASED VIEW
At first create a model which we wil be using to create instances through our view i.e app/models.py

```
# import the standard Django Model
# from built-in library
from django.db import models

# declare a new model with a name "GeeksModel"
class GeeksModel(models.Model):

    # fields of the model
    title = models.CharField(max_length = 200)
    description = models.TextField()

    # renames the instances of the model
    # with their title name
    def __str__(self):
        return self.title
```

Then,makemigrations and migrate.

If we want to see our model and its data in the admin panel,the we need to register our model .
app/admin.py

```
from django.contrib import admin
from .models import GeeksModel

admin.site.register(GeeksModel)
```

In shell ,

```
>>>python manage.py shell
>>>from app.models import GeeksModel
>>>GeeksModel.objects.create(title="title1",description="description").save()
```

Creating a view and template for the same. app/views.py

```
from django.shortcuts import render
from .models import GeeksModel

def list_view(request):
    # dictionary for initial data with
    # field names as keys
    context ={}

    # add the dictionary during initialization
    context["dataset"] = GeeksModel.objects.all()

    return render(request, "list_view.html", context)
```

Template:

```
<div class="main">

    {% for data in dataset %}.
    {{ data.title }}<br/>
    {{ data.description }}<br/>
    <hr/>

    {% endfor %}

</div>
```

ANOTHER EXAMPLE:

GET:

```
from django.shortcuts import render

def my_view(request):
    if request.method == 'GET':
        return render(request, 'mytemplate.html')
```

POST:

```
from django.shortcuts import render, redirect
from .models import MyModel

def my_view(request):
    if request.method == 'POST':
        form = MyForm(request.POST)
        if form.is_valid():
            data = form.cleaned_data
            MyModel.objects.create(**data)
            return redirect('success_page')
    else:
        form = MyForm()
    return render(request, 'mytemplate.html', {'form': form})
```

ANOTHER EXAMPLES CRUD OPERATIONS

After creating project and app,lets create a model by creating instances through our view.
app/models.py

```
from django.db import models

class GeeksModel(models.Model):
    #fields
    title=models.CharField(max_length=200)
    description=models.TextField()

    #renaming instances
    def __str__(self):
        return self.title
```

Makemigrations and migrate.

In shell ,

```
>>>python manage.py shell
>>>from app.models import GeeksModel
>>>GeeksModel.objects.create(title="title1",description="description").save()
```

Now,We will be creating a Django ModelForm:
create a file forms.py in created app,

CREATING A FORM

```
from django import forms
from .models import GeeksModel

class GeeksForm(forms.ModelForm):
    class Meta:
    model= GeeksModel          (Models to be used)
    fields=[                   (Fields to be used)
```

```
            "title",
            "description"
            ]
```

1. CREATE VIEW (logic)

-Logic to create an instance of a table in the db like taking an input from the user and storing it in a specified table.
app/views.py

```
from django.shortcuts import render

# relative import of forms
from .models import GeeksModel
from .forms import GeeksForm



def create_view(request):
    # dictionary for initial data with
    # field names as keys
    context = {}

    # add the dictionary during initialization
    form = GeeksForm(request.POST or None)
    if form.is_valid():
        form.save()

    context['form'] = form
    return render(request, "create_view.html", context)
```

In tampletes;

```
<form method="POST" enctype="multipart/form-data">

    <!-- Security token -->
    {% csrf_token %}
    <!-- Using the formset -->
    {{ form.as_p }}

    <input type="submit" value="Submit">
</form>
```

2. RETRIEVE VIEW
It is divided into two types:
-Detail View
-List View

LIST VIEW
-It is used to list all or particular instances of a table from the db in a particular order.
Used to display multiple types of data on a single page or view.
app/views.py

```
from django.shortcuts import render
from .models import GeeksModel

def lis_view(request):
      context={}
      context["dataset"]= GeeksModel.objects.all()  or .all().order_by("-id") or
.filter(title__icontains = "title")
      return render(request,"list_view.html",context)
```

Templates;

```
<div class="main">

    {% for data in dataset %}.

    {{ data.title }}<br/>
    {{ data.description }}<br/>
    <hr/>

    {% endfor %}
```

```
</div>
```

DETAIL VIEW
-It is used to display a particular instance of a table from the db with all the necessary details.example,profile of a user

app/urls.py

```
from django.urls import path
from .views import detail_view

urlpatters=[
   path('<id>',detail_view),
   ]
```

app/views.py

```
from django.shortcuts import render

# relative import of forms
from .models import GeeksModel

# pass id attribute from urls
def detail_view(request, id):
   # dictionary for initial data with
   # field names as keys
   context ={}

   # add the dictionary during initialization
   context["data"] = GeeksModel.objects.get(id = id)

   return render(request, "detail_view.html", context)
```

In templates;

```
<div class="main">
```

```
    <!-- Specify fields to be displayed -->
    {{ data.title }}<br/>
    {{ data.description }}<br/>

</div>
```

3. UPDATE VIEW

-It is a view(logic) to update a particluar instance of a table from the db with some extra details.Ex,updating an article

app/urls.py

```
from django.urls import path

# importing views from views..py
from .views import update_view, detail_view

urlpatterns = [
    path('<id>/', detail_view ),
    path('<id>/update', update_view ),
]
```

app/views.py

```
from django.shortcuts import (get_object_or_404,render,HttpResponseRedirect)

# relative import of forms
from .models import GeeksModel
from .forms import GeeksForm

# after updating it will redirect to detail_View
def detail_view(request, id):
    # dictionary for initial data with
    # field names as keys
    context ={}
    # add the dictionary during initialization
    context["data"] = GeeksModel.objects.get(id = id)

    return render(request, "detail_view.html", context)

# update view for details
def update_view(request, id):
    # dictionary for initial data with
    # field names as keys
    context ={}

    # fetch the object related to passed id
    obj = get_object_or_404(GeeksModel, id = id)

    # pass the object as instance in form
    form = GeeksForm(request.POST or None, instance = obj)

    # save the data from the form and
    # redirect to detail_view
    if form.is_valid():
        form.save()
        return HttpResponseRedirect("/"+id)

    # add form dictionary to context
    context["form"] = form

    return render(request, "update_view.html", context)
```

Templates;

1.update_view.html

```
<div class="main">
    <!-- Create a Form -->
    <form method="POST">
        <!-- Security token by Django -->
        {% csrf_token %}
        <!-- form as paragraph -->
        {{ form.as_p }}

        <input type="submit" value="Update">
    </form>

</div>
```

2.detail_view.html

```
<div class="main">
    <!-- Display attributes of instance -->
    {{ data.title }} <br/>
    {{ data.description }}
</div>
```

4. DELETE VIEW

-It is a view(logic) to delete a particular instance from the db.

app/views.py

```
from django.shortcuts import (get_object_or_404,render,HttpResponseRedirect)

from .models import GeeksModel


# delete view for details
def delete_view(request, id):
    # dictionary for initial data with
    # field names as keys
    context ={}

    # fetch the object related to passed id
    obj = get_object_or_404(GeeksModel, id = id)
    if request.method =="POST":
        # delete object
        obj.delete()
        # after deleting redirect to
        # home page
        return HttpResponseRedirect("/")

    return render(request, "delete_view.html", context)
```

app/urls.py

```
from django.urls import path

# importing views from views..py
from .views import delete_view
urlpatterns = [
    path('<id>/delete', delete_view ),
]
```

Templates;delete_view.html

```
<div class="main">
    <!-- Create a Form -->
    <form method="POST">
        <!-- Security token by Django -->
        {% csrf_token %}
        Are you want to delete this item ?
        <input type="submit" value="Yes" />
        <a href="/">Cancel </a>
    </form>
</div>
```

2. Class Based Django Views

-It is an alternative way to implement views as python objects instead of functions.

Advantages compared to function-based-views

1. Organization of code related to specific HTTP methods (GET, POST, etc.) can be addressed by separate methods instead of conditional branching.
2. Object oriented techniques such as mixins (multiple inheritance) can be used to factor code into reusable components.

How to use class based views?

app/views.py

```
from django.views.generic.list import ListView
from .models import GeeksModel

class GeeksList(ListView):

    # specify the model for list view
    model = GeeksModel
```

app/urls.py

```
from django.urls import path

# importing views from views..py
from .views import GeeksList
urlpatterns = [
    path('', GeeksList.as_view()),
]
```

Templates;

```
<ul>
    <!-- Iterate over object_list -->
    {% for object in object_list %}
    <!-- Display Objects -->
    <li>{{ object.title }}</li>
    <li>{{ object.description }}</li>

    <hr/>
    <!-- If objet_list is empty  -->
    {% empty %}
    <li>No objects yet.</li>
    {% endfor %}
</ul>
```

CRUD OPERATIONS FOR CLASS BASED VIEWS

After we have successfully started project and created an app.Creating a model of which we will be creating instances through our view.

app/models.py

```
# import the standard Django Model
# from built-in library
from django.db import models

# declare a new model with a name "GeeksModel"
class GeeksModel(models.Model):

    # fields of the model
    title = models.CharField(max_length = 200)
    description = models.TextField()

    # renames the instances of the model
    # with their title name
    def __str__(self):
        return self.title
```

Then makemigrations and migrate

And, we will create a Django ModelForm for this model.Create a file forms.py in app

```
from django import forms
from .models import GeeksModel

# creating a form
class GeeksForm(forms.ModelForm):

    # create meta class
    class Meta:
        # specify model to be used
        model = GeeksModel

        # specify fields to be used
        fields = [
            "title",
            "description",
        ]
```

app/views.py

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

class MyView(View):
    def get(self, request):
        # <view logic>
        return HttpResponse('result')
```

And,app/urls.py(Using as_view() method in class based view)

```
# urls.py
from django.urls import path
from myapp.views import MyView

urlpatterns = [
    path('about/', MyView.as_view()),

]
```

1. CREATE VIEW
- Class based views automatically setup everything from A-Z.We just need to specify which model to create View for and the fields. Then Class based CreateView will automatically try to find a template in app_name/modelname_form.html. In our case it is geeks/templates/app/geeksmodel_form.html. Let's create our class based view. In app/views.py,

app/views.py

```
from django.views.generic.edit import CreateView
from .models import GeeksModel

class GeeksCreate(CreateView):

    # specify the model for create view
    model = GeeksModel

    # specify the fields to be displayed

    fields = ['title', 'description']
```

app/urls.py

```
from django.urls import path

# importing views from views..py
from .views import GeeksCreate
urlpatterns = [
    path('', GeeksCreate.as_view() ),
]
```

Templates;

```
<form method="POST" enctype="multipart/form-data">
```

```
    <!-- Security token -->
    {% csrf_token %}

    <!-- Using the formset -->
    {{ form.as_p }}

    <input type="submit" value="Submit">
</form>
```

2. RETRIEVE VIEWS

LISTVIEW

- Class Based Views automatically setup everything from A to Z. One just needs to specify which model to create ListView for, then Class based ListView will automatically try to find a template in app_name/modelname_list.html. In our case it is app/templates/app/geeksmodel_list.html. Let's create our class based view.

In app/views.py,

```
from django.views.generic.list import ListView
from .models import GeeksModel

class GeeksList(ListView):

    # specify the model for list view
    model = GeeksModel
```

In app/urls.py

```
from django.urls import path

# importing views from views..py
from .views import GeeksList
urlpatterns = [
    path('', GeeksList.as_view()),
]
```

In templates;templates/geeks/geeksmodel_list.html

```
<ul>
   <!-- Iterate over object_list -->
   {% for object in object_list %}
   <!-- Display Objects -->
   <li>{{ object.title }}</li>
   <li>{{ object.description }}</li>

   <hr/>
   <!-- If object_list is empty  -->
   {% empty %}
   <li>No objects yet.</li>
   {% endfor %}
</ul>
```

MANIPULATE QUERYSET IN LISTVIEW

By default ListView will display all instances of a table in the order they were created. If one wants to modify the sequence of these instances or the ordering, get_queryset method need to be overridden.

In app/views.py,

```
from django.views.generic.list import ListView
from .models import GeeksModel

class GeeksList(ListView):

        # specify the model for list view
        model = GeeksModel

        def get_queryset(self, *args, **kwargs):
                qs = super(GeeksList, self).get_queryset(*args, **kwargs)
                qs = qs.order_by("-id")
                return qs
```

DETAIL VIEW

-One just needs to specify which model to create DetailView for, then Class based DetailView will automatically try to find a template in app_name/modelname_detail.html. In our case it is app/templates/app/geeksmodel_detail.html. Let's create our class based view.

1. In app/views.py,

```
from django.views.generic.detail import DetailView

from .models import GeeksModel

class GeeksDetailView(DetailView):
    # specify the model to use
    model = GeeksModel
```

2. In app/urls.py

```
from django.urls import path

# importing views from views..py
from .views import GeeksDetailView
urlpatterns = [
    # <pk> is identification for id field,
    # slug can also be used
    path('<pk>/', GeeksDetailView.as_view()),
]
```

3. Template in templates/geeks/geeksmodel_detail.html,

```
<h1>{{ object.title }}</h1>



<p>{{ object.description }}</p>
```

MANIPULATE CONTEXT DATA IN DETAILVIEW

-By default DetailView will only display fields of a table. If one wants to modify this context data before sending it to template or add some extra field, context data can be overridden.

In app/views.py,

```
from django.views.generic.detail import DetailView

from .models import GeeksModel

class GeeksDetailView(DetailView):
	# specify the model to use
	model = GeeksModel

	# override context data
	def get_context_data(self, *args, **kwargs):
		context = super(GeeksDetailView,
			self).get_context_data(*args, **kwargs)
		# add extra field
		context["category"] = "MISC"
		return context
```

Templates; geeks/templates/geeksmodel_detail.html,

```
<h1>{{ object.title }}</h1>
<p>{{ object.description }}</p>
<p>{{ category }}</p>
```

3. UPDATE VIEW

-Class Based Views automatically setup everything from A to Z. One just needs to specify which model to create UpdateView for, then Class based UpdateView will automatically try to find a template in app_name/modelname_form.html. In our case it is geeks/templates/geeks/geeksmodel_form.html. Let’s create our class based view.

In app/views.py,

```
# import generic UpdateView
from django.views.generic.edit import UpdateView

# Relative import of GeeksModel
from .models import GeeksModel
```

```
class GeeksUpdateView(UpdateView):
    # specify the model you want to use
    model = GeeksModel

    # specify the fields
    fields = [
        "title",
        "description"
    ]

    # can specify success url
    # url to redirect after successfully
    # updating details
    success_url ="/"
```

In app/urls.py

```
from django.urls import path

# importing views from views..py
from .views import GeeksUpdateView
urlpatterns = [
    # <pk> is identification for id field,
    # <slug> can also be used
    path('<pk>/update', GeeksUpdateView.as_view()),
]
```

Templates; templates/geeks/geeksmodel_form.html,

```
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Save">
</form>
```

4. DELETE VIEW
-same as create,retrieve and detail

In app/views.py

```
# import generic UpdateView
from django.views.generic.edit import DeleteView

# Relative import of GeeksModel
from .models import GeeksModel

class GeeksDeleteView(DeleteView):
    # specify the model you want to use
    model = GeeksModel

    # can specify success url
    # url to redirect after successfully
    # deleting object
    success_url ="/"

    template_name = "geeks/geeksmodel_confirm_delete.html"
```

In app/urls.py

```
from django.urls import path

# importing views from views..py
from .views import GeeksDeleteView
urlpatterns = [
    # <pk> is identification for id field,
    # slug can also be used
    path('<pk>/delete/', GeeksDeleteView.as_view()),
]
```

Templates;   templates/geeks/geeksmodel_confirm_delete.html,

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



<p>Are you sure you want to delete "{{ object }}"?</p>



    <input type="submit" value="Confirm">
</form>
```

5. FORM VIEW
-same

In app/views.py

```
# import generic FormView
from django.views.generic.edit import FormView

# Relative import of GeeksForm
from .forms import GeeksForm

class GeeksFormView(FormView):
    # specify the Form you want to use
    form_class = GeeksForm

    # specify name of template
    template_name = "geeks / geeksmodel_form.html"

    # can specify success url
    # url to redirect after successfully
    # updating details
    success_url ="/thanks/"
```

Create a template for this view in app/geeksmodel_form.html,

```
<form method="post">
   {% csrf_token %}
   {{ form.as_p }}
   <input type="submit" value="Save">
</form>
```

Map a url to this view in app/urls.py,

```
from django.urls import path

# importing views from views..py
from .views import GeeksFormView
urlpatterns = [
   path('', GeeksFormView.as_view()),
]
```

FINAL EXAMPLE OF CLASS BASED VIEW
STEP1: django-admin startproject core
STEP2: python manage.py startapp books
STEP3: core/urls.py and set path for our app,

```
     from django.contrib import admin
     from django.urls import path,include

     urlpatters=[
         path('admin/'.admin.site.urls),
         path('',include('books.urls',namespace='books'))
         ]
```

STEP4: core/settingd.py and register our app
STEP5: books/admin.py

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

     @admin.register(models.Books)
     class AuthorAdmin(admin.ModelAdmin):
         prepopulated_fields = {'slug':('title'),}
```

STEP6: create a model in books/models.py

```
      from django.db import models
      from django.template.defaultfilters import slugify

      class Books(models.Model):
      title = models.CharField(max_length=100)
      slug= models.SlugField(null=True)
      genre = models.CharField(max_lenth=100)
      author = models.CharField(max_lenth=100)
      isbn = models.IntegerField()
      count = models.IntegerField(null=True,default=0)
```

STEP7:create a form in which we will be creating CreateView.
In books/forms.py

```
from django import forms
from .models import Books

class AddForm(forms.ModelForm):
      class Meta:
      model = Books
      fields = ('title','genre','author','isbn')
      widgets = {
          'title':forms.TextInput(attrs={'class':'form-control'}),
          'genre':forms.TextInput(attrs={'class':'form-control'}),
          'author':forms.TextInput(attrs={'class':'form-control'}),
          'isbn':forms.TextInput(attrs={'class':'form-control'}),
           }
```

STEP8: lets create CreateView in app/views.py

```
      from django.views.generic.edit import CreateView
      from . forms import AddForm

      class AddBookView(CreateView):
          #model1 = Books
          form_class = AddForm
```

```
        template_name = 'add.html'
        success_url = '/books/'
```

STEP9:Map a url to this view in books/urls.py

```
    from django.urls import path
    from .views import AddBookView

    app_name = 'books'
    urlpatterns = [
        path('',AddBookView.as_view(),name = 'add'),
        ]
```

STEP10:Set Urls for books/urls.py

```
     from django.urls import path
        from .views import AddBookView

        app_name = 'books'

        urlpatterns = [
            path('', AddBookView.as_view(), name='add'),
        ]
```

STEP11:Create a template for this view in books/add.html,

```
<html lang="en">
<head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Document</title>
</head>
<body>
        <div class="container" style="max-width:600px">
                <div class="px-3 py-3 pt-md-5 pb-md-4 mx-auto text-center">
                        <h1 class="display-4">Welcome to GFG Class Based Views Django</h1>
```

```
				</div>
				<div class="py-5">
					<div class="row">
						<div class="col-12">
							<form method="post">
								{% csrf_token %}
								{{ form.as_p }}
								<input type="submit">
							</form>
						</div>
					</div>
				</div>
			</div>
		</body>
		</html>
```

STEP12:python manage.py runserver
    http://127.0.0.1:8000/

1.3 DJANGO URL DISPATCHER

In django,URL Dispatcher is used to map the url with view,which in turns helps to handle the request and produce a response.

1.3.1 METHODS OF USING DJANGO URL DISPATCHER

1. CREATING URL PATTERNS IN DJANGO
-In urls.py
To create a URL pattern :
-Open thr urls.py file
-Import path()function from django.urls
-Define list of url patterns

EXAMPLE:

```
from django.urls import path
from . import views

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

2. USING REGULAR EXPRESSION CAPTURES IN DJANGO URL PATTERNS

-If we want to extract the values from urls then pass it to views then we can do it with the help of regular expression catures.Then extracted value then can be used as according to the need in the view function.

To use:

-Define the regular expression in url pattern and add capturing groups by the help of () to colect values from the pattern.

-In the view function,include for each capturing group

-When the url pattern is matched,the matched view function is called.

EXAMPLE:

In urls.py

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.home, name='home'),
    path('blog/<int:blog_id>/', views.blog_detail, name='blog_detail'),

]
```

In views.py

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

def home(request):
    return HttpResponse("Welcome to our website!")

def blog_detail(request, blog_id):
    blog_post = {'id': blog_id, 'title': 'Sample Blog Post', 'content': 'This is the content of the blog post.'}
    context = {'blog_post': blog_post}
    return render(request, 'blog_detail.html', context)
```

3. NAMING URL PATTERNS IN DJANGO

To use:
-In the path function specify the name with a string value.
-In the templates,use the {%url ' ' %} and in the code use the reverse() function to refer to the named url pattern.

In urls.py

```
from django.urls import path
from . import views

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

In templates;
{% url 'home' %} like:

```
<!DOCTYPE html>
<html>
<head>
<title>Page Title</title>
</head>
<body>
<h2>Welcome To GFG</h2>
<a href="{% url 'home' %}">HOME</a>
```

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

4. INVERTING URL PATTERNS IN DJANGO

-In django,inverting URL PATTERNS refers to the process of reversing a url to its corresponding view or url pattern name.

Two ways:

1. Using the {% url ' ' %} template tag

```
<a href="{% url 'home' %}">Home</a>
```

2.Using the reverse() function in views

In views,models etc,we can use the reverse() function to reverse a URL based on its name.

Import it from django.urls import reverse and pass the url name as argument.

In views.py

```
from django.urls import reverse
from django.shortcuts import redirect

def my_view(request):
    return redirect(reverse('home'))
```

5. USING NAMESPACES IN DJNAGO URL PATTERNS

-In django,namespaces are used to organize url patterns and prevent naming conflicts when multiple applications are integrated into a single project.

EXAMPLE:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('myapp1/', include('myapp1.urls', namespace='myapp1')),
    path('myapp2/', include('myapp2.urls', namespace='myapp2')),
]
```

To reference URLs with namespaces in your templates or views, use the {% url %} template tag or the reverse() function with the format ‘namespace:url_name’. For example:

```
<a href="{% url 'namespace:url_name" %}">Home</a>
```

6.CLASS-BASED VIEWS IN DJANGO URL DISPATCHER

Creating class based view,

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

class ItemListView(View):
    def get(self, request):
        items = ["Item 1", "Item 2", "Item 3"]
        return HttpResponse("\n".join(items))
```

Mapping the class based view to a url,
To map your class-based view to a URL, you can use the as_view() method of your view class when defining URL patterns in your app’s urls.py file:

```
from django.urls import path
from .views import ItemListView

urlpatterns = [
    path('items/', ItemListView.as_view(), name='item-list'),
]
```

1.3.2 DJANGO URL PATTERNS

Including other URLConf modules

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
	path('admin/', admin.site.urls),
	path('', include('books.urls')),
]
```

This tells Django to search for URL Patterns in the file books/urls.py
i.e;

```
from django.urls import path
from . import views

urlpatterns = [
	path('books/<int:pk>/', views.book_detail),
	path('books/<str:genre>/', views.books_by_genre),
	path('books/', views.book_index),
]
```

PATH CONVERTORS:

1. int- Matches zero or any positive integer
2. str – Matches any non-empty string, excluding the path separator('/').
3. path – Matches any non-empty string including the path separator('/')
4. slug – Matches any slug string, i.e. a string consisting of alphabets, digits, hyphen and under score.
5. uuid – Matches a UUID(universal unique identifier).

2. GET PARAMETERS PASSED BY URLS IN DJANGO

After project setup and created app,then in models.py

```
class Article(models.Model):
	author = models.CharField(max_length = 20)
```

```
        content = models.TextField()
```

Then go to admin and add some articles for testing;

Creating url;

```
url_patterns += [
        path("articles/<id>/", views.article_detail, name ="article_detail"),
]
```

Make sure to import your views.py file here.
The <id> here will help us get and use the id parameter in our view.

In views.py;

```
def article_detail(request, id):
article = Article.objects.filter(id = id)
return render("your_template", context ={"article":article})
```

3. URL VALIDATOR IN DJANGO

URL Validation can be easily implemented using built-in tools and libraries.

To use validator:

1. Create a project folder and navigate to the project directory

```
django-admin startproject bookstore
cd bookstore
```

Activate Virtual Environment

2. Create a Django App

```
python manage.py startapp mini
```

3. Define a model

In this example, we’ll create a simple model to store the validated URLs;

```
# mini/models.py
```

```
from django.db import models

class ValidatedURL(models.Model):
    url = models.URLField(unique=True)

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

4. Create form.py

```
# mini/forms.py
from django import forms
from django.core.validators import URLValidator

class URLForm(forms.Form):
    url = forms.URLField(
        label='Enter a URL',
        validators=[URLValidator()],
        widget=forms.TextInput(attrs={'placeholder': 'https://example.com'})
    )
```

5. Generate view of the App

The index view handles URL validation and form submissions, while the success view displays the list of validated URLs.

```
# mini/views.py
from django.shortcuts import render, redirect
from .forms import URLForm
from .models import ValidatedURL

def index(request):
    if request.method == 'POST':
        form = URLForm(request.POST)
        if form.is_valid():
            url = form.cleaned_data['url']
            ValidatedURL.objects.create(url=url)
```

```
                return redirect('success')
        else:
            form = URLForm()
        return render(request, 'url_validator_app/index.html', {'form': form})

def success(request):
    validated_urls = ValidatedURL.objects.all()
    return render(request, 'url_validator_app/success.html', {'validated_urls': validated_urls})
```

6. Create the templates
Create two HTML templates: one for the form and another for displaying the validated URLs.

template/index.html:URL Validator

```
<!DOCTYPE html>
<html>
<head>
    <title>URL Validator</title>
</head>
<body>
    <h1>URL Validator</h1>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Submit</button>
    </form>
</body>
</html>
```

template/index2.html:Validated URLs

```
<!DOCTYPE html>
<html>
<head>
    <title>Validated URLs</title>
</head>
<body>
```

```
        <h1>Validated URLs</h1>
        <ul>
                {% for url in validated_urls %}
                        <li>{{ url }}</li>
                {% empty %}
                        <li>No validated URLs yet.</li>
                {% endfor %}
        </ul>
        <a href="{% url 'index' %}">Back to validation</a>
</body>
</html>
```

7. Configure URLs in the mini/urls.py

```
# mini/urls.py
from django.urls import path
from . import views

urlpatterns = [
        path('', views.index, name='index'),
        path('success/', views.success, name='success'),
]
```

8. Include app URLs in the project Urls;

```
# url_validator_project/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
        path('admin/', admin.site.urls),
        path('', include('mini.urls')),
]
```

4. URL SHORTNER WITH DJANGO

How to build a URL Shortner ?
-Views.py is basically used to connect our database,api with our frontend.

Views.py
```
from django.http import HttpResponse
def index(request):
   return HttpResponse("Hello World")
```

urls.py
```
from django.urls import path
from . import views
app_name = "url"
urlpatterns = [
   path("", views.index, name="home")
]
```

Creating Django Models;
First of all, we need a Database to store our Shorten URL's. For That, We need to create a Schema for our Database Table in models.py.

models.py
```
from django.db import models
class UrlData(models.Model):
   url = models.CharField(max_length=200)
   slug = models.CharField(max_length=15)
def __str__(self):
      return f"Short Url for: {self.url} is {self.slug}"
```

Creating a form;

forms.py
```
from django import forms
class Url(forms.Form):
   url = forms.CharField(label="URL")
```

Creating Views;

Now, We will create the Interface of our App using views.py. Let’s divide this part in Functions.

urlShort()— This Function is where our Main Algorithm works. It takes a url from form after User submits it, then it generates a Random Slug which is then stored in Database with Original Url. It is also the function which render index.html (entrypoint of our app)

views.py

```
def urlShort(request):
   if request.method == 'POST':
      form = Url(request.POST)
      if form.is_valid():
         slug = ''.join(random.choice(string.ascii_letters)
                    for x in range(10))
         url = form.cleaned_data["url"]
         new_url = UrlData(url=url, slug=slug)
         new_url.save()
         request.user.urlshort.add(new_url)
         return redirect('/')
   else:
      form = Url()
   data = UrlData.objects.all()
   context = {
      'form': form,
      'data': data
   }
   return render(request, 'index.html', context)
```

urlRedirect() — This Function tracks the slug to Original URL and redirects it to Original URL.

```
def urlRedirect(request, slugs):
   data = UrlData.objects.get(slug=slugs)
   return redirect(data.url)
```

Creating Routes;

Urls.py

```
from django.urls import path
from . import views
app_name = "url"
urlpatterns = [
    path("", views.urlShort, name="home"),
    path("u/<str:slugs>", views.urlRedirect, name="redirect")
]
```

Run the project

5.DJANGO URLRESOLVER ERROR

Url Resolver error in Django pops out when there is a mistake in your URL patterns configurations. This can be caused by incorrect URL patterns, view function mismatches, namespace conflicts, circular imports, middleware ordering, or server configuration problems.

Syntax: TypeError: 'URLResolver' object is not subscriptable

Common mistakes that can cause this error:

1. Typo error:While creating the URLs and views you may make spelling mistakes or any typo may be there.
2. Incorrect URL Path: In templates, an incorrect URL path is mentioned.
3. App specific: if you have created many apps then from the base project URL to a specific app the request not moving forward.

Solutions:

1. ADD ROOT_URLCONFG SETTING:

```
    ROOT_URLCONF = 'myproject.urls'
```

2. Add Static/Media files settings

```
  from django.conf import settings
  from django.conf.urls.static import static
  urlpatterns = [
      # ... the rest of your URLconf goes here ...
  ]
urlpattern += static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
urlpattern += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

3. Clear Cache And Restart the Server
python manage.py flush

1.4 DJANGO TEMPLATES:

Templates are the 3rd and most imp part of MVT structure and basically written in HTML,CSS AND JAVASCRIPT.

There are 2 methods of adding the templates to our website depending on our needs.

1. We can use a single template directory which will be spread over the entire project.
2. For each app,we can create a different template directory.

TEMPLATES:

1. Single directory

Configurations:

settings.py: "BASE_DIR/'templates'"

```
TEMPLATES = [
   {
      # Template backend to be used, For example Jinja
      'BACKEND': 'django.template.backends.django.DjangoTemplates',

      ## Path definition of templates folder .
      'DIRS': [BASE_DIR/'templates'],

      'APP_DIRS': True,
      # options to configure
      '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',
         ],
      },
   },
]
```

To render a template one needs to a view and a url mapped to the view.

1. app/views.py

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

# create a function
def simple_view(request):
    data = {"content": "Gfg is the best"}
    return render(request, "geeks.html", data)

def check_age(request):
    if request.method == 'POST':
      # Get the age from the form
        age = int(request.POST.get('age', 0))
        return render(request, 'check_age.html', {'age': age})
    return render(request, 'check_age.html')

def loop(request):
    data = "Gfg is the best"
    number_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    context = {
        "data": data,
        "list": number_list}

    return render(request, "loop.html", context)
```

2. app/urls.py

```
from django.urls import path

# importing views from views..py
from mini import views
```

```
urlpatterns = [
    path('simple',views.simple_view),
    path('condition', views.check_age),
    path('loop', views.loop),
]
```

3. app.html

```
<!-- app_name/templates/geeks.html  -->

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Homepage</title>
</head>
<body>
    <h1>Welcome to Geeksforgeeks.</h1>
<p> {{  data }}</p>

    <ul>
</body>
</html>
```

check_app.html

```
<!-- app_name/templates/check_age.html -->

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Age Checker</title>
</head>
```

```
<body>
   <h1>Welcome to the Age Checker</h1>

   <form method="post">
      {% csrf_token %}
      <label for="age">Enter your age:</label>
      <input type="number" id="age" name="age">
      <button type="submit">Check Age</button>
   </form>

   {% if age %}
      <p>Your age is: {{ age }}</p>
      {% if age >= 20 %}
         <p>You are an adult.</p>
      {% else %}
         <p>You are not an adult.</p>
      {% endif %}
   {% endif %}
</body>
</html>
```

loop.html

```
<!-- app_name/templates/loop.html -->

<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Even Numbers</title>
</head>
<body>
   <h1>{{ data }}</h1>
   Even Numbers
   <ul>
      {% for number in list %}
         {% if number|divisibleby:2 %}
```

```
            <li>{{ number }}</li>
          {% endif %}
        {% endfor %}
    </ul>
</body>
</html>
```

1.4.1 DJANGO TEMPLATE LANGUAGE

-Facility provided by Django templates.
The main characteristics of Django Template Language are :

1. variables
2. Tags
3. Filters
4. Comments

1. JINJA VARIABLES:

Variables output a value from the context, which is a dict-like object mapping keys to values. The context object we sent from the view can be accessed in the template using variables of Django Template.

Syntax: {{ variable_name }}

Example:

1. In app/views.py

```
# import render from django
from django.shortcuts import render

# create a function
def geeks_view(request):
        # create a dictionary
        context = {
                "first_name" : "Naveen",
                "last_name" : "Arora",
        }
        # return response
        return render(request, "geeks.html", context)
```

2. app/urls.py

```
from django.urls import path

# importing views from views..py
from .views import geeks_view

urlpatterns = [
        path('', geeks_view),
]
```

3. Templates;

```
My First Name is {{ first_name }}.
<br/>
My Last Name is {{ last_name }}.
```

2. JINJA TAGS:

-Tags provide arbitrary logic in the rendering process.
- -a tag can output content
- -serve as a control structure
- -grab content from a database
- -even enable access to other template tags.

Syntax: {% tag_name %}

a) Comment:{% comment 'comment_name' %}
        {% endcomment %}

b) Cycle: {% cycle 'value_1' 'value_2' %}(useful in loops)

c) Extends: {% extends 'template_name.html' %} (extends tag is used for inheritance of templates as well as variables in django.)

```
        {% extends "geeks.html" %}
        {% block content %}
        <h2> GeeksForGeeks is the Best
        {% endblock %}
```

Example: Assume the following directory structure:

```
dir1/
    template.html
    base2.html
    my/
        base3.html
base1.html
```

In template.html, the following paths would be valid:

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

d) If:

```
{% if variable %}
        // statements
        {% else %}
        // statements
        {% endif %}
```

If tag may use and,or,not to test a number of variables or to negate a given varia..

e) For loop:

```
{% for i in list %}
                {% endfor %}
```

```
<ul>
{% for athlete in athlete_list %}
            <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

Advanced Usage:

One can use variables too,like;

```
{% for o in some_list %}
            <tr class="{% cycle rowvalue1 rowvalue2 %}">
                        ...
            </tr>
{% endfor %}
```

One can loop over a list in reverse by using :{% for obj in list reversed %}

If you need to loop over a list of lists, you can unpack the values in each sublist into individual variables. For example, if your context contains a list of (x, y) coordinates called points, you could use the following to output the list of points:

```
{% for x, y in points %}
   There is a point at {{ x }}, {{ y }}
{% endfor %}
```

This can also be useful if you need to access the items in a dictionary. For example, if your context contained a dictionary data, the following would display the keys and values of the dictionary:

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

f) Empty tag: It can be used in for tag;like

```
 {% for i in list %}
// Do this in non - empty condition
{% empty %}
// Do this in empty condition
{% endfor %}
```

g) Boolean operators(and,or and not like the above one)

```
{% if variable boolean_operator value %}
// statements
{% endif %}
```

And,

Boolean Operators

== operator

Equality. Example:

```
{% if somevar == "x" %}
  This appears if variable somevar equals the string "x"
{% endif %}
```

!= operator

Inequality. Example:

```
{% if somevar != "x" %}
  This appears if variable somevar does not equal the string "x",
  or if somevar is not found in the context
{% endif %}
```

< operator

Less than. Example:

```
{% if somevar < 100 %}
  This appears if variable somevar is less than 100.
{% endif %}
```

> operator

Greater than. Example:

```
{% if somevar > 0 %}
  This appears if variable somevar is greater than 0.
{% endif %}
```

<= operator

Less than or equal to. Example:

```
{% if somevar <= 100 %}
  This appears if variable somevar is less than 100 or equal to 100.
{% endif %}
```

>= operator

Greater than or equal to. Example:

```
{% if somevar >= 1 %}
  This appears if variable somevar is greater than 1 or equal to 1.
{% endif %}
```

in operator

Contained within. This operator is supported by many Python containers to test whether the given value is in the container. The following are some examples of how x in y will be interpreted:

```
{% if "bc" in "abcdef" %}
  This appears since "bc" is a substring of "abcdef"
```

```
{% endif %}
{% if "hello" in greetings %}
  If greetings is a list or set, one element of which is the string
  "hello", this will appear.
{% endif %}
{% if user in users %}
  If users is a QuerySet, this will appear if user is an
  instance that belongs to the QuerySet.
{% endif %}
```

not in operator
Not contained within. This is the negation of the in operator.

is operator
Object identity. Tests if two values are the same object. Example:

```
{% if somevar is True %}
  This appears if and only if somevar is True.
{% endif %}

{% if somevar is None %}
  This appears if somevar is None, or if somevar is not found in the context.
{% endif %}
```

is not operator
Negated object identity. Tests if two values are not the same object. This is the negation of the is operator. Example:

```
{% if somevar is not True %}
  This appears if somevar is not True, or if somevar is not found in the
  context.
{% endif %}
{% if somevar is not None %}
  This appears if and only if somevar is not None.
{% endif %}
```

h) firstof: {% firstof var1 var2 var3... %}
i) include: {% include "template_name.html" %}

j) lorem: {% lorem [count] [method] [random] %}

count – A number (or variable) containing the number of paragraphs or words to generate (default is 1). {% lorem %}

method – Either w for words, p for HTML paragraphs or b for plain-text paragraph blocks (default is b). {% lorem 3 p %}

random – The word random, which if given, does not use the common paragraph ("Lorem ipsum dolor sit amet…") when generating text. {% lorem 2 w random %}

k) now: {% now "D M Y H T " %}

l) url: {% url 'some-url-name' v1 v2 %}

Advanced Usage

suppose you have a view, app_views.client, whose URLconf takes a client ID (here, client() is a method inside the views file app_views.py). The URLconf line might look like this:

```
path('client/<int:id>/', app_views.client, name='app-views-client')
```

If this app's URLconf is included into the project's URLconf under a path such as this:

```
path('clients/', include('project_name.app_name.urls'))
```

…then, in a template, you can create a link to this view like this:

```
{% url 'app-views-client' client.id %}
```

FILTERS:

It used to transform the values of variables and tag arguments.Tags can't modify the value of a variable whereas filters can be used for incrementing the value of a variable or modifying it to one's own need.

Syntax: {{ variable_name | filter_name }}

1. add: {{ value | add:"2" }} increment
2. addslashes: {{ value | addslashes }} It is used to add slashes before quotes. Useful for escaping strings in CSV.
3. capfirst: {{ value | capfirst }}
4. center: "{{ value | center:"15" }}"
5. cut: {{ value | cut:" " }}
6. date: {{ value | date:"D d M Y" }} format date
7. default: {{ value | default:"nothing" }}
8. dictsort: {{ value | dictsort:"name" }}

If value is:

```
[
    {'name': 'zed', 'age': 19},
    {'name': 'amy', 'age': 22},
    {'name': 'joe', 'age': 31},
]
```

then the output would be:

```
[
    {'name': 'amy', 'age': 22},
    {'name': 'joe', 'age': 31},
    {'name': 'zed', 'age': 19},
]
```

9. divisibleby: {{ value | divisibleby:"3" }}
10. escape: {{ title | escape }}

It is used to escape a string's HTML. Specifically, it makes these replacements:

< is converted to &lt;

> is converted to &gt;

' (single quote) is converted to &#x27;

" (double quote) is converted to &quot;

& is converted to &amp;

11. filesizeof: {{ value | filesizeformat }}
12. first: {{ value | first }} first item in the list
13. join: {{ value | join:" // " }}

It is used to join a list with a string, like Python's str.join(list)

If value is the list ['a', 'b', 'c'], the output will be the string "a // b // c".

14. last: {{ value | last }}
15. length: {{ value | length }}
16. linenumbers: {{ value | linenumbers }}

It is used to display text with line numbers

If value is:

```
one
two
three
```

the output will be:

```
1. one
```

2. two

3. three

17. lower: {{ value | lower }}}

18. make_list: {{ value | make_list }}

If value is the string “Naveen”, the output would be the list [‘N’, ‘a’, ‘v’, ‘e’, ‘e’, ‘n’]. If value is 123, the output will be the list [‘1’, ‘2’, ‘3’].

19. random: {{ value | random }} random item from given list

20. slice: {{ some_list | slice:":2" }}

21. slugify: {{ value | slugify }}

It is used to convert to ASCII. It converts spaces to hyphens and removes characters that aren’t alphanumerics, underscores, or hyphens. Converts to lowercase. Also strips leading and trailing whitespace. Example

If value is “Jai is a slug”, the output will be “jai-is-a-slug”.

22. time: {{ value | time:"H:i" }} format time

23. timesince: {{ blog_date | timesince:comment_date }}

24. title: {{ value | title }} If value is “my FIRST post”, the output will be “My First Post”.

25. unordered_list: {{ var | unordered_list }}

It is used to recursively take a self-nested list and returns an HTML unordered list – WITHOUT opening and closing <ul> tags.

if var contains [‘States’, [‘Kansas’, [‘Lawrence’, ‘Topeka’], ‘Illinois’]], then {{ var|unordered_list }} would return:

```
<li>States
<ul>
     <li>Kansas
     <ul>
          <li>Lawrence</li>
          <li>Topeka</li>
     </ul>
     </li>
     <li>Illinois</li>
</ul>
</li>
      </li>
```

26. upper: {{ value | upper }}
27. wordcount: {{ value | wordcount }}
If value is “jai is a slug”, the output will be 4.

1.5 DJANGO MODELS

(To create tables, their fields represents column and various constraints in the database)
Syntax:(In the created app)
Djnago models is the sql Database one uses with Django.Django models simplify the tasks and organize tables into models.Each models maps to a single database table to store data into the database.
-We can use the admin panel of Django to create,update,delete or retrieve fields of a model and other operations.
BASICS OF MODELS INCLUDE:
1. Each model is a python class that subclasses djnago.db.models.Model
2. Each attribute od model represents a database fields.
3. Django gives us an automatically generated database-access API.

SYNTAX:
```
from django.db import models
class ModelName(models.Model):
      field_name = models.Field(**options)
```

Example 1 :
```
from django.db import models
from django.db.models import Model
class Post(models.Model):
   title = models.CharField(max_length=255)
   body = models.TextField()

   def __str__(self) -> str:
      return self.title
```

Example 2 :
```
from django.db import models
```

```
class GeeksModel(models.Model):
    title = models.CharField(max_length=200)
    description = models.TextField()
    last_modified = models.DateTimeField(auto_now_add=True)
    img = models.ImageField(upload_to=& quot
                            images/&quot
                        )(an image field that will be uploaded to a directory called images)

    # renames the instances of the model
    # with their title name
    def __str__(self):
        return self.title
```

Whenever we CREATE a model,DELETE a model or UPDATE anything in any models.py,we need to run
2 commands:
1. python manage.py makemigrations
We can run the makemigrations individually for specific app as well like: if a and b is the name of the app then,

```
    python manage.py makemigrations a
    python manage.py makemigrations b
```

2. python manage.py migrate
Then it will create table in database.
RENDER A MODEL IN DJANGO ADMIN INTERFACE
- To render a model in Django admin,we need to modify app/admin.py:

```
from django.contrib import admin

# Register your models here.
from .models import GeeksModel
admin.site.register(GeeksModel)
```

We can then manage the model from admin panael along with CRUD operations on the model.

1.5.1 DJANGO CRUD - INSERTING,UPDATING AND DELETING DATA:

Django lets us interact with database models i.e add,delete,modify and query objects using a database-abstraction API called ORM(OBJECT RELATIONAL MAPPER).
It is a bridge that gap betn the database and the applications code.
We can access the ORM by:

1.Django Shell
2.Django Admin

FOR EXMAPLE: In models.py

```
class Album(models.Model):
    title = models.CharField(max_length = 30)
    artist = models.CharField(max_length = 30)
    genre = models.CharField(max_length = 30)

    def __str__(self): (i.e changing object display name )
        return self.title

class Song(models.Model):
    name = models.CharField(max_length = 100)
    album = models.ForeignKey(Album, on_delete = models.CASCADE)

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

<------------------DJANGO SHELL----------------->

Then,Django Shell i.e Django ORM can be accessed by :

```
- python manage.py shell
```

Importing our models using :

```
>>> form books.models import Song,Album
```

Django ORM Queries:

1. INSERING/ADDING DATA WITH DJANGO ORM

```
  >>> a=Album(title="Divide",artist="Ed Shreen",genre="pop")
  >>> a.save()
  >>> s = Song(name = "Castle on the Hill", album = a)
  >>> s.save()
   or,
  >>>> a = GeeksModel(
         title = "GeeksForGeeks",
         description = "A description here",
         img = "geeks/abc.png"
         )
  >>> a.save()
```

2. RETRIVING DATA WITH DJANGO ORM

```
  >>> Album.objects.all()
  or
  >>> Album.objects.filter(artist="Ed Shreen")(returns the matching parameters)
  >>> Album.objects.exclude(genre = "Rock")(returns others rather than the matching
parameters)
  >>> Album.objects.get(pk = 3)(returns the single object which matches the given lookup
parameter)
```

3. MODIFYING EXISTING OBJECTS

```
>>> a = GeeksModel.objects.get(id = 3)
>>> a.title = "Pop"
>>> a.save()
```

4. DETEING DATA WITH DJANGO ORM

   Deleting a single object:

```
>>> a = Album.objects.get(pk = 2)
>>> a.delete()
>>> Album.objects.all()
```

   Deleting multiple objects using filter() or exclude()

```
>>> Album.objects.filter(genre = "Pop").delete()
>>> Album.objects.all()
```

VALIDATION ON FIELDS IN A MODEL

- Every field comes in with built-in validations from Django validators.

Like IntegerField need only integer value of certain range only.

Field Options :

Field Options are the arguments given to each field for applying some constraint or imparting a particular characteristic to a particular Field.

1. Null
2. Blank
3. db_column
4. Default
5. help_text
6. primary_key
7. editable
8. error_messages
9. help_text
10. verbose_name
11. validators
12. unique

CUSTOM FIELD VALIDATIONS IN DJANGO MODELS

-Adding custom validation to a particular field.

SYNTAX:

```
field_name = models.Field(validators = [function 1, function 2])
```

Example:FOR MAIL CUSTOM VALIDATION

1. In app/geeks.py

```
from django.db import models
from django.db.models import Model
# Create your models here.
from django.core.exceptions import ValidationError

# creating a validator function
def validate_geeks_mail(value):
    if "@gmail.com" in value:
        return value
    else:
        raise ValidationError("This field accepts mail id of google only")

class GeeksModel(Model):
    geeks_mail = models.CharField(
                    max_length = 200,
                    validators = [validate_geeks_mail]
                    )
```

BASIC MODEL DATA TYPES AND FIELDS LIST:

1. AutoField
2. BigAutoField
3. BigIntegerField
4. BinaryField
5. BooleanField
6. CharField
7. DateField
8. DateTimeField
9. DecimalField
10. DurationField
11. EmailField
12. FileField
13. FloatField
14. ImageField
15. IntegerField
16. GenericIPAddressField
17. NullBooleanField
18. PositiveIntegerField
19. PositiveSmallIntegerField
20. SlugField
21. SmallIntegerField
22. TextField
23. TimeField
24. URLField
25. UUIDField

RELATIONSHIP FIELDS

1. ForeignKey(Many-to-one):

Many-to-one fields:

This is used when one record of a model A is related to multiple records of another model B. For example – a model Song has many-to-one relationship with a model Album, i.e. an album can have many songs, but one song cannot be part of multiple albums. Many-to-one relations are defined using ForeignKey field of django.db.models.

Below is an example to demonstrate the same.

```
from django.db import models

class Album(models.Model):
        title = models.CharField(max_length = 100)
        artist = models.CharField(max_length = 100)

class Song(models.Model):
        title = models.CharField(max_length = 100)
        album = models.ForeignKey(Album, on_delete = models.CASCADE)
```

It is a good practice to name the many-to-one field with the same name as the related model, lowercase.

2. ManyToManyField:

This is used when one record of a model A is related to multiple records of another model B and vice versa. For example – a model Book has many-to-many relationship with a model Author, i.e. an book can be written by multiple authors and an author can write multiple books. Many-to-many relations are defined using ManyToManyField field of django.db.models.

```
from django.db import models

class Author(models.Model):
        name = models.CharField(max_length = 100)
        desc = models.TextField(max_length = 300)

class Book(models.Model):
        title = models.CharField(max_length = 100)
        desc = models.TextField(max_length = 300)
        authors = models.ManyToManyField(Author)
```

It is a good practice to name the many-to-many field with the plural version of the related model, lowercase. It doesn't matter which of the two models contain the many-to-many field, but it shouldn't be put in both the models.

3. OneToOneField:

This is used when one record of a model A is related to exactly one record of another model B. This field can be useful as a primary key of an object if that object extends another object in some way. For example – a model Car has one-to-one relationship with a model Vehicle, i.e. a car is a vehicle. One-to-one relations are defined using OneToOneField field of django.db.models.

```
from django.db import models

class Vehicle(models.Model):
    reg_no = models.IntegerField()
    owner = models.CharField(max_length = 100)

class Car(models.Model):
    vehicle = models.OneToOneField(Vehicle,
        on_delete = models.CASCADE, primary_key = True)
    car_model = models.CharField(max_length = 100)
```

It is a good practice to name the one-to-one field with the same name as that of the related model, lowercase.

Data integrity options:

Since we are creating models which depend on other models, we need to define the behavior of a record in one model when the corresponding record in the other is deleted. This is achieved by adding an optional on_delete parameter in the relational field, which can take the following values:

1. on_delete = models.CASCADE – This is the default value. It automatically deletes all the related records when a record is deleted.(e.g. when an Album record is deleted all the Song records related to it will be deleted)

2. on_delete = models.PROTECT – It blocks the deletion of a record having relation with other records.(e.g. any attempt to delete an Album record will be blocked)

3. on_delete = models.SET_NULL – It assigns NULL to the relational field when a record is deleted, provided null = True is set.

4. on_delete = models.SET_DEFAULT – It assigns default values to the relational field when a record is deleted, a default value has to be provided.

5. on_delete = models.SET() – It can either take a default value as parameter, or a callable, the return value of which will be assigned to the field.

6. on_delete = models.DO_NOTHING – Takes no action. Its a bad practice to use this value.

DJANGO FIELD CHOICES

Field Choices are a sequence consisting itself of iterables of exactly two items (e.g. [(A, B), (A, B) …]) to use as choices for some field.

For example, consider a field semester which can have options as { 1, 2, 3, 4, 5, 6 } only.

EXAMPLE:

1. In app/models.py

```
from django.db import models
# specifying choices
SEMESTER_CHOICES = (
        ("1", "1"),
        ("2", "2"),
        ("3", "3"),
        ("4", "4"),
        ("5", "5"),
        ("6", "6"),
        ("7", "7"),
        ("8", "8"),
)

# declaring a Student Model
```

```
class Student(models.Model):
        semester = models.CharField(
                max_length = 20,
                choices = SEMESTER_CHOICES,
                default = '1'
                )
```

One can also collect your available choices into named groups;

```
MEDIA_CHOICES = [
   ('Audio', (
        ('vinyl', 'Vinyl'),
        ('cd', 'CD'),
      )
   ),
   ('Video', (
        ('vhs', 'VHS Tape'),
        ('dvd', 'DVD'),
      )
   ),
   ('unknown', 'Unknown'),
]
```

ADDING THE SLUG FIELD INSIDE DJANGO MODELS

The slug field within Django models is a pivotal step for improving the structure and readability of URLs in web applications. This addition allows developers to automatically generate URL-friendly slugs based on titles, enhancing user experience and search engine optimization (SEO). By implementing this feature, you can create cleaner, more meaningful, and SEO-friendly URLs for your content, which is essential for attracting and retaining website visitors.

LIKE: www.geeksforgeeks.org/posts/the-django-book-by-geeksforgeeks

1. In app/models.py

```
STATUS_CHOICES = (
('draft', 'Draft'),
('published', 'Published'),
)

class Post(models.Model):
title = models.CharField(max_length = 250)
slug = models.SlugField(max_length = 250, null = True, blank = True)
text = models.TextField()
published_at = models.DateTimeField(auto_now_add = True)
updated = models.DateTimeField(auto_now = True)

status = models.CharField(max_length = 10, choices = STATUS_CHOICES,
                                        default ='draft')


class Meta:
   ordering = ('-published_at', )

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

2. Now we need to convert the title into a slug automatically.For this we will use signals:

Add new file util.py in the same directory where settings.py is saved.

```
import string, random
from django.db.models.signals import pre_save
from django.dispatch import receiver
from django.utils.text import slugify

def random_string_generator(size = 10, chars = string.ascii_lowercase + string.digits):
        return ''.join(random.choice(chars) for _ in range(size))

def unique_slug_generator(instance, new_slug = None):
        if new_slug is not None:
                slug = new_slug
        else:
                slug = slugify(instance.title)
        Klass = instance.__class__
        max_length = Klass._meta.get_field('slug').max_length
        slug = slug[:max_length]
        qs_exists = Klass.objects.filter(slug = slug).exists()

        if qs_exists:
                new_slug = "{slug}-{randstr}".format(
                        slug = slug[:max_length-5], randstr = random_string_generator(size = 4))

                return unique_slug_generator(instance, new_slug = new_slug)
        return slug
```

SIGNALS: They are the utilities that allow associating events with actions.

In app/models.py

```
@receiver(pre_save, sender=Post)
def pre_save_receiver(sender, instance, *args, **kwargs):
if not instance.slug:
        instance.slug = unique_slug_generator(instance)
```

3. Modify URL with slug:
By creating URL patterns that include the slug as a parameter.

```
from django.urls import path
from . import views

urlpatterns = [
        path('posts/<slug:slug>/', views.post_detail, name='post_detail'),
        # Other URL patterns
]
```

4. Modify views:
detail view function ,and In urls.py edit detail path with path('posts/', detail). In views.py edit the detail function with :

```
def detail(request, slug):
    # Filter posts based on the slug (case-insensitive)
    q = Post.objects.filter(slug__iexact=slug)

    if q.exists():
        # If a post with the given slug exists, retrieve the first matching post
        q = q.first()
    else:
        # If no post is found, return an "Post Not Found" response
        return HttpResponse('<h1>Post Not Found</h1>')

    # Create a context dictionary containing the retrieved post
    context = {'post': q}

    # Render the 'details.html' template with the context
    return render(request, 'posts/details.html', context)
```

5. Last Step:

The last step is to add the link in HTML file <a href=”/posts/{{ a.slug }}” class=”btn btn-primary”>View</a>. Now we are ready to go to 127.0.0.1:8000/posts/title-you-have-added and it will show you the page details.html.

INTERMEDIATE FIELDS IN DJANGO

In Django, a many-to-many relationship exists between two models A and B, when one instance of A is related to multiple instances of B, and vice versa. For example – In a shop management system, an Item and a Customer share a many-to-many relationship, as one customer can buy multiple items, and multiple customers can buy the same item.

However, there may be some fields that are neither specific to the customer, nor to the item bought, but rather to the purchase of the item by the customer. e.g. quantity purchased date of buying, etc. For storing such intermediary data, we need intermediate models. We need to specify the intermediate model via through parameter in ManyToManyField.

```
from django.db import models

class Item(models.Model):
    name = models.CharField(max_length = 128)
    price = models.DecimalField(max_digits = 5, decimal_places = 2)
    def __str__(self):
        return self.name

class Customer(models.Model):
    name = models.CharField(max_length = 128)
    age = models.IntegerField()
    items_purchased = models.ManyToManyField(Item, through = 'Purchase')

    def __str__(self):
        return self.name

class Purchase(models.Model):
    item = models.ForeignKey(Item, on_delete = models.CASCADE)
    customer = models.ForeignKey(Customer, on_delete = models.CASCADE)
    date_purchased = models.DateField()
    quantity_purchased = models.IntegerField()
```

we can create instances of our Purchase model;

```
i = Item.objects.create(name = "Water Bottle", price = 100)
c = Customer.objects.create(name = "Abhishek", age = 21)
p = Purchase(item = i, customer = c,
                date_purchased = date(2019, 7, 7),
                quantity_purchased = 3)

p.save()


c.items_purchased.all()


i.customer_set.all()
```

UPLOADING IMAGES IN DJANGO

In most websites, we often deal with media data such as images, files, etc. In Django, we can deal with the images with the help of the model field which is ImageField.

1. Settings.py

```
  MEDIA_ROOT =  os.path.join(BASE_DIR, 'media')
  MEDIA_URL = '/media/'
```

2. Urls.py

```
from django.conf import settings
from django.conf.urls.static import static
if settings.DEBUG:
        urlpatterns += static(settings.MEDIA_URL,
                              document_root=settings.MEDIA_ROOT)
```

EXAMPLE:

A sample models.py should be like this, in that we have created a Hotel model which consists of hotel name and its image. In this project we are taking the hotel name and its image from the user for hotel booking website.

In models.py

```
# models.py
 class Hotel(models.Model):
    name = models.CharField(max_length=50)
    hotel_Main_Img = models.ImageField(upload_to='images/')
```

In app/forms.py

```
# forms.py
from django import forms
from .models import Hotel



class HotelForm(forms.ModelForm):


    class Meta:
        model = Hotel
        fields = ['name', 'hotel_Main_Img']
```

In templates under app;

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hotel_image</title>
</head>
<body>
    <form method = "post" enctype="multipart/form-data">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Upload</button>
    </form>
</body>
</html>
```

In app/views.py;

```
from django.http import HttpResponse
from django.shortcuts import render, redirect
from .forms import HotelForm

# Create your views here.


def hotel_image_view(request):

    if request.method == 'POST':
        form = HotelForm(request.POST, request.FILES)

        if form.is_valid():
            form.save()
            return redirect('success')
    else:
        form = HotelForm()
    return render(request, 'hotel_image_form.html', {'form': form})


def success(request):
    return HttpResponse('successfully uploaded')
```

In urls.py;

```
from django.contrib import admin
from django.urls import path
from django.conf import settings
from django.conf.urls.static import static
from .views import hotel_image_view

urlpatterns = [
    path('image_upload', hotel_image_view, name='image_upload'),
    path('success', success, name='success'),
]

if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
                          document_root=settings.MEDIA_ROOT)
```

Then makemigrations and migrate;

Now for accessing those images ;
In views.py;

```
def display_hotel_images(request):

    if request.method == 'GET':

        # getting all the objects of hotel.
        Hotels = Hotel.objects.all()
        return render((request, 'display_hotel_images.html',
                    {'hotel_images': Hotels}))
```

A sample html file template for displaying images;

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Hotel Images</title>

   <meta name="viewport" content="width=device-width, initial-scale=1">
   <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
   <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js">
   </script>
   <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js">
   </script>
</head>
<body>

   {% for hotel in hotel_images %}
         <div class="col-md-4">
               {{ hotel.name }}
               <img src="{{ hotel.hotel_Main_Img.url }}" class="img-responsive" style="width:
100%; float: left; margin-right: 10px;" />
         </div>
   {% endfor %}

</body>
</html>
```

Then,Insert the url path in the urls.py file;

```
# urls.py
path('hotel_images', display_hotel_images, name = 'hotel_images'),
```

RENDER MODEL IN DJANGO ADMIN INTERFACE

Rendering model in admin refers to adding the model to the admin interface so that data can be manipulated easily using admin interface. Django's ORM provides a predefined admin interface that can be used to manipulate data by performing operations such as INSERT, SEARCH, SELECT, CREATE, etc. as in a normal database. To start entering data in your model and using admin interface, one needs to specify or render model in admin.py.

In app/models.py;

```
from django.db import models
from django.db.models import Model
# Create your models here.

class GeeksModel(models.Model):
   title = models.CharField(max_length = 200)
   content = models.TextField(max_length = 200, null = True, blank = True)
   views = models.IntegerField()
   url = models.URLField(max_length = 200)
   image = models.ImageField()
```

Creation of superuser:

```
Python manage.py createsuperuser
```

Then,
In app/admin.py;

```
from django.contrib import admin

# Register your models here.
from .models import GeeksModel

admin.site.register(GeeksModel)
```

1.6 DJANGO FORMS

Forms are used for taking input from the user in some manner and using that information for logical operations on databases.

Django maps the fields defined in Django forms into HTML input fields.Django handles three distinct parts of the work involved in forms:

1. Preparing and restructuring data to make it ready for rendering.
2. Creating HTML forms for the data.
3. Receiving and processing submitted forms and data from the client.

SYNTAX:

```
field_name=forms.FieldType(**options)
```

EXAMPLE:

```
from django import forms

# creating a form
class GeeksForm(forms.Form):
   title = forms.CharField()
   description = forms.CharField()
```

1.6.1 CREATING FORMS IN DJANGO

-Creating forms jn Django is completely similar to creating a model, one needs to specify what fields would exist in the form and of what type.

EXAMPLE:

```
# import the standard Django Forms
# from built-in library
from django import forms

# creating a form
class InputForm(forms.Form):

   first_name = forms.CharField(max_length = 200)
   last_name = forms.CharField(max_length = 200)
   roll_number = forms.IntegerField(
                  help_text = "Enter 6 digit roll number"
                  )
   password = forms.CharField(widget = forms.PasswordInput())
```

1.6.2 RENDER DJANGO FORMS

Now to render this form into a view by creating instance of the form class created above,
In app/views.py

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

# Create your views here.
def home_view(request):
        context ={}
        form=InputForm(request.POST or None)
        context['form']= form
        return render(request, "home.html", context)
```

In templates;
home.html

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

And,its running now.
Django provides some predefined ways to show forms in a convenient manner.In templates,the following will modify the inputs as:

{{ form.as_table }} will render them as table cells wrapped in <tr> tags
{{ form.as_p }} will render them wrapped in <p> tags (BEST)
{{ form.as_ul }} will render them wrapped in <li> tags

One can modify these settings also and show fields as he wants using {{ form.field_name }} but this may alter the normal process of validation if some field is empty and hence needs extraordinary care.

We can also render Django forms fields manually;

-We can render these fields manually to improve some visual stuff. Each field is available as an attribute of the form using {{ form.name_of_field }}, and in a Django template, will be rendered appropriately.

In html file;

```
<html>

<head>
	<link
	rel="stylesheet"
	href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css">
	<style>
		.i-am-centered {
			margin: auto;
			max-width: 300px;
			padding-top: 20%;
		}
	</style>
</head>

<body>
	<div class="i-am-centered">
		<form method="POST">
			{% csrf_token %}
			<div class="form-group">
				<label>First Name </label>
				{{ form.first_name }}
			</div>
			<div class="form-group">
				<label>Last Name </label>
				{{ form.last_name }}
```

```
                    </div>
                    <div class="form-group">
                              <label>Roll Number</label>
                              {{ form.roll_number }}
                    </div>
                    <div class="form-group">
                              <label>Password</label>
                              {{ form.password }}
                    </div>
                    <button type="submit" class="btn btn-primary">Submit</button>
          </form>
     </div>
</body>

</html>
```

These are the basic modifications using bootstrap.One can customize it to an advanced level using various CSS tricks and methods.

{{ field }} attributes

1. {{ field.label }}
The label of the field, e.g. Email address.

2. {{ field.label_tag }}
The field's label wrapped in the appropriate HTML tag. This includes the form's label_suffix. For example, the default label_suffix is a colon:

```
<label for="id_email">Email address:</label>
```

3. {{ field.id_for_label }}
The ID that will be used for this field (id_email in the example above). If you are constructing the label manually, you may want to use this in place of label_tag. It's also useful, for example, if you have some inline JavaScript and want to avoid hardcoding the field's ID.
{{ field.value }}

The value of the field. e.g someone@example.com.

4. {{ 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.

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

6. {{ field.errors }}
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.

7. {{ 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 %}
```

8. {{ field.field }}
The Field instance from the form class that this BoundField wraps. You can use it to access Field attributes, e.g. {{ char_field.field.max_length }}

1.6.3 CREATING DJANGO FORMS FROM MODELS

Django ModelForm is a class that is used to directly convert a model into a Django form. If you're building a database-driven app, chances are you'll have forms that map closely to Django models.

In app/models.py;

```
# import the standard Django Model
# from built-in library
from django.db import models

# declare a new model with a name "GeeksModel"
```

```
class GeeksModel(models.Model):
        # fields of the model
    title = models.CharField(max_length = 200)
    description = models.TextField()
    last_modified = models.DateTimeField(auto_now_add = True)
    img = models.ImageField(upload_to = "images/")

        # renames the instances of the model
        # with their title name
    def __str__(self):
        return self.title
```

This form takes 2 arguments fields or exclude.

1.FOR FIELDS:

To create a form directly from this model,
In app/forms.py;

```
# import form class from django
from django import forms

# import GeeksModel from models.py
from .models import GeeksModel

# create a ModelForm
class GeeksForm(forms.ModelForm):
    # specify the name of model to use
    class Meta:
        model = GeeksModel
        fields = "__all__"
```

2.FOR EXCLUDE:

```
class PartialAuthorForm(ModelForm):
    class Meta:
```

```
    model = Author
    exclude = ['title']
```

Then, to complete our MVT structure,
In app/views.py;

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

def home_view(request):
	context ={}

	# create object of form
	form = GeeksForm(request.POST or None, request.FILES or None)

	# check if form data is valid
	if form.is_valid():
		# save the form data to model
		form.save()

	context['form']= form
	return render(request, "home.html", context)
```

1.6.4 RENDER HTML FORMS (GET & POST) IN DJANGO

At first,we need to be familier with GET AND POST METHODS OF HTTPS;
HTTP(HYpertext Transfer Protocol) request methods GET and POST requests in python.

HTTP is a set of protocols designed to enable communications between clients and servers.It works as a request-response protocol betn client and server.A web browser may be the client, and an application on a computer that hosts a website may be the server. So, to request a response from the server, there are mainly two methods:

GET: To request data from the server.
POST: To submit data to be processed to the server.

https://media.geeksforgeeks.org/wp-content/uploads/getpostRequest.png

Now,to make HTTP requests in Python,we can use several HTTP libraries like:
1. httplib
2. urllib
3. requests (most elegant and simplest)

```
pip install requests
```

MAKING A GET REQUEST:
The above example finds the latitude, longitude, and formatted address of a given location by sending a GET request to the Google Maps API. An API (Application Programming Interface) enables you to access the internal features of a program in a limited fashion. And in most cases, the data provided is in JSON(JavaScript Object Notation) format (which is implemented as dictionary objects in Python!).

```
# importing the requests library
import requests

# api-endpoint
URL = "http://maps.googleapis.com/maps/api/geocode/json"

# location given here
location = "delhi technological university"

# defining a params dict for the parameters to be sent to the API
PARAMS = {'address':location}

# sending get request and saving the response as response object
r = requests.get(url = URL, params = PARAMS)

# extracting data in json format
data = r.json()


# extracting latitude, longitude and formatted address
# of the first matching location
latitude = data['results'][0]['geometry']['location']['lat']
longitude = data['results'][0]['geometry']['location']['lng']
formatted_address = data['results'][0]['formatted_address']

# printing the output
print("Latitude:%s\nLongitude:%s\nFormatted Address:%s"
   %(latitude, longitude,formatted_address))
```

MAKING A POST REQUEST:

-This example explains how to paste your source_code to pastebin.com by sending a POST request to the PASTEBIN API. First of all, you will need to generate an API key by signing up here:https://pastebin.com/signup and then accessing your API key here:https://pastebin.com/doc_api#1

```
# importing the requests library
import requests

# defining the api-endpoint
API_ENDPOINT = "http://pastebin.com/api/api_post.php"

# your API key here
API_KEY = "XXXXXXXXXXXXXXXXX"

# your source code here
source_code = '''
print("Hello, world!")
a = 1
b = 2
print(a + b)
'''

# data to be sent to api
data = {'api_dev_key': API_KEY,
        'api_option': 'paste',
        'api_paste_code': source_code,
        'api_paste_format': 'python'}
```

```
# sending post request and saving response as response object
r = requests.post(url=API_ENDPOINT, data=data)

# extracting response text
pastebin_url = r.text
print("The pastebin URL is:%s" % pastebin_url)
```

Here are some important points to ponder upon:

When the method is GET, all form data is encoded into the URL and appended to the action URL as query string parameters. With POST, form data appears within the message body of the HTTP request.

In the GET method, the parameter data is limited to what we can stuff into the request line (URL). Safest to use less than 2K of parameters, some servers handle up to 64K.No such problem in the POST method since we send data in the message body of the HTTP request, not the URL.

Only ASCII characters are allowed for data to be sent in the GET method. There is no such restriction in the POST method.

GET is less secure compared to POST because the data sent is part of the URL. So, the GET method should not be used when sending passwords or other sensitive information.

GET METHOD:

In app/templates/home.html;

```
<form action = "" method = "get">
        <label for="your_name">Your name: </label>
        <input id="your_name" type="text" name="your_name">
        <input type="submit" value="OK">
</form>
```

In mainapp/urls.py;

```
from django.urls import path

# importing views from views..py
from .views import geeks_view

urlpatterns = [
    path('', home_view ),
]
```

Now, let’s move to our home_view and start checking how are we going to get the data. Entire data from an HTML form in Django is transferred as a JSON object called a request. Let’s create a view first and then we will try all methods to fetch data from the form.

```
from django.shortcuts import render

# Create your views here.
def home_view(request):

    # logic of view will be implemented here
    return render(request, "home.html")
```

By default every form ever written in HTML makes a GET request to the back end of an application, a GET request normally works using queries in the URL.

The above URL is appended with a name attribute of the input tag and the name entered in the form. This is how the GET request works whatever be the number of inputs they would be appended to the URL to send the data to the back end of an application. Let’s check how to finally get this data in our view so that logic could be applied based on input.
In views.py

```
from django.shortcuts import render

# Create your views here.
def home_view(request):
    print(request.GET)
    return render(request, "home.html")
```

request.GET returns a query dictionary that one can access like any other python dictionary and finally use its data for applying some logic.

Similarly, if the method of transmission is POST, you can use request.POST as query dictionary for rendering the data from the form into views.

In home.html;

```
<form action = "" method = "POST">
        {% csrf_token %}
        <label for="your_name">Your name: </label>
        <input id="your_name" type="text" name="your_name">
        <input type="submit" value="OK">
</form>
```

Note that whenever we create a form request, Django requires you to add {% csrf_token %} in form for security purposes

Now, in views.py let's check what request.POST has got.

```
from django.shortcuts import render

# Create your views here.
def home_view(request):
        print(request.POST)
        return render(request, "home.html")
```

This way one can use this data for querying into the database or for processing using some logical operation and pass using the context dictionary to the template.

1.6.5 DJANGO FORM FIELD CUSTOM WIDGETS:

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. 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.

Custom Django form field widgets:

One can override the default widget of each field for various purposes.https://docs.djangoproject.com/en/3.0/ref/forms/widgets/

To override the default widget we need to explicitly define the widget we want to assign to a field.

Example:

```
from django import forms
```

```
class GeeksForm(forms.Form):
        title = forms.CharField(widget = forms.Textarea)
        description = forms.CharField(widget = forms.CheckboxInput)
        views = forms.IntegerField(widget = forms.TextInput)
        available = forms.BooleanField(widget = forms.Textarea)
```

Using Widgets to customize Datefield

-widgets have a great use in Form Fields especially using Select type of widgets where one wants to limit the type and number of inputs form a user.

In forms.py;

-from django import forms

```
class GeeksForm(forms.Form):
        title = forms.CharField()
        description = forms.CharField()
        views = forms.IntegerField()
        date = forms.DateField()
```

By default DateField as widget TextInput;Lets change the widget for better and convenient input from the user of a date.Add SelectDateWidget to DateField in forms.py;

LIKE;

```
from django import forms

class GeeksForm(forms.Form):
        title = forms.CharField()
        description = forms.CharField()
        views = forms.IntegerField()
        date = forms.DateField(widget = forms.SelectDateWidget)
```

Now input of date can be seen as very easy and helpful in the front end of the application. This way we can use multiple widgets for modifying the input fields.

1.6.6 PYTHON FORM VALIDATION USING DJANGO

In app/models.py;

```
from django.db import models

# model named Post
class Post(models.Model):
        Male = 'M'
        FeMale = 'F'
        GENDER_CHOICES = (
        (Male, 'Male'),
        (FeMale, 'Female'),
        )

        # define a username field with bound max length it can have
        username = models.CharField( max_length = 20, blank = False,null = False)

        # This is used to write a post
        text = models.TextField(blank = False, null = False)

        # Values for gender are restricted by giving choices
        gender = models.CharField(max_length = 6, choices = GENDER_CHOICES,
                                                default = Male)

        time = models.DateTimeField(auto_now_add = True)
```

After creating the data models, the changes need to be reflected in the database to do this run the following command:

```
-python manage.py makemigrations
-python manage.py migrate
```

Then,
In app/forms.py;

```
from django.forms import ModelForm
from django import forms
from formValidationApp.models import *
```

```
# define the class of a form
class PostForm(ModelForm):
        class Meta:
                # write the name of models for which the form is made
                model = Post

                # Custom fields
                fields =["username", "gender", "text"]

        # this function will be used for the validation
        def clean(self):

                # data from the form is fetched using super function
                super(PostForm, self).clean()

                # extract the username and text field from the data
                username = self.cleaned_data.get('username')
                text = self.cleaned_data.get('text')

                # conditions to be met for the username length
                if len(username) < 5:
                        self._errors['username'] = self.error_class([
                                'Minimum 5 characters required'])
                if len(text) <10:
                        self._errors['text'] = self.error_class([
                                'Post Should Contain a minimum of 10 characters'])

                # return any errors if found
                return self.cleaned_data
```

In mainapp/urls.py;

```
from django.contrib import admin
from django.urls import path, include
from django.conf.urls import url
from django.shortcuts import HttpResponse
from . import views


urlpatterns = [
        path('', views.home, name ='index'),
]
```

In, mainapp/views.py;

```
from .models import Post
from .forms import PostForm
from .import views
from django.shortcuts import HttpResponse, render, redirect


def home(request):

        # check if the request is post
        if request.method =='POST':

                # Pass the form data to the form class
                details = PostForm(request.POST)

                # In the 'form' class the clean function
                # is defined, if all the data is correct
                # as per the clean function, it returns true
                if details.is_valid():

                        # Temporarily make an object to be add some
                        # logic into the data if there is such a need
                        # before writing to the database
                        post = details.save(commit = False)

                        # Finally write the changes into database
                        post.save()

                        # redirect it to some another page indicating data
                # was inserted successfully
                return HttpResponse("data submitted successfully")

            else:

                # Redirect back to the same page if the data
                # was invalid
                return render(request, "home.html", {'form':details})
        else:

            # If the request is a GET request then,
            # create an empty form object and
            # render it into the page
            form = PostForm(None)
            return render(request, 'home.html', {'form':form})
```

home.html;

```
{% load bootstrap3 %}
{% bootstrap_messages %}
<!DOCTYPE html>
<html lang="en">

<head >

        <title>Basic Form</title>

        <meta charset="utf-8" />

        <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">

<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js">
</script>
```

```
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js">
</script>
</head>

<body style="padding-top: 60px;background-color: #f5f7f8 !important;">
        <div class="container">
        <div class="row">
                <div class="col-md-4 col-md-offset-4">
                <h2>Form</h2>
                        <form action="" method="post"><input type='hidden'/>
                        {%csrf_token %}

                                {% bootstrap_form form %}
<!-This is the form variable which we are passing from the function
of home in views.py file. That's the beauty of Django we
don't need to write much codes in this it'll automatically pass
all the form details in here
->
                        <div class="form-group">
                                <button type="submit" class="btn btn-default ">
                                Submit
                                </button>

                                </div>

                                </form>
                </div>
        </div>
</div>

</body>

</html>
```

1.6.7 DJANGO FORMSETS

-Formsets in a Django is an advanced way of handling multiple forms on a single webpage. In other words, Formsets are a group of forms in Django.A formset is a layer of abstraction to work with multiple forms on the same page.

Creation of FormSet:

```
from django.forms import formset_factory
GeeksFormSet = formset_factory(GeeksForm)
```

Suppose having a project named mainapp and app named app;

In app/forms.py;

```
from django import forms

# create a form
class GeeksForm(forms.Form):
   title = forms.CharField()
   description = forms.CharField()
```

In app/views.py;
Create formset_view;

```
from django.shortcuts import render

# relative import of forms
from .forms import GeeksForm

# importing formset_factory
from django.forms import formset_factory

def formset_view(request):
        context ={}

        # creating a formset
        GeeksFormSet = formset_factory(GeeksForm)
        formset = GeeksFormSet()

        # Add the formset to context dictionary
        context['formset']= formset
        return render(request, "home.html", context)
```

To render the formset through HTML, create home.html;
In templates/home.html;

```
<form method="POST" enctype="multipart/form-data">
   {% csrf_token %}
   {{ formset.as_p }}
   <input type="submit" value="Submit">
</form>
```

1.6.8 HOW TO CREATE MULTIPLE FORMS USING DJANGO FROMSETS:

-Django formsets are used to handle multiple instances of a form.One can create multiple forms easily using extra attribute of Django FormSets.

In app/views.py;

```
from django.shortcuts import render

# relative import of forms
from .forms import GeeksForm

# importing formset_factory
from django.forms import formset_factory

def formset_view(request):
        context ={}

        # creating a formset and 5 instances of GeeksForm
        GeeksFormSet = formset_factory(GeeksForm, extra = 5)
```

```
    formset = GeeksFormSet()

    # Add the formset to context dictionary
    context['formset']= formset
    return render(request, "home.html", context)
```

The keyword argument extra makes multiple copies of same form. i.e 5 forms .

HANDLING MULTIPLE FORMS USING DJANGO FORMSETS:

-Creating a form is much easier than handling the data entered into those fields at the back end.When trying to handle formset, Django formsets required one extra argument {{ formset.management_data }}.

LINK:https://docs.djangoproject.com/en/3.0/topics/forms/formsets/#understanding-the-managementform

In templates/home.html;

```
<form method="POST" enctype="multipart/form-data">

    <!-- Management data of formset -->
    {{ formset.management_data }}

    <!-- Security token -->
    {% csrf_token %}

    <!-- Using the formset -->
    {{ formset.as_p }}

    <input type="submit" value="Submit">
</form>
```

Now edit,formset_view to print the data;

In app/views.py;

```
from django.shortcuts import render

# relative import of forms
```

```
from .forms import GeeksForm

# importing formset_factory
from django.forms import formset_factory

def formset_view(request):
        context ={}

        # creating a formset and 5 instances of GeeksForm
        GeeksFormSet = formset_factory(GeeksForm, extra = 3)
        formset = GeeksFormSet(request.POST or None)

        # print formset data if it is valid
        if formset.is_valid():
                for form in formset:
                        print(form.cleaned_data)

        # Add the formset to context dictionary
        context['formset']= formset
        return render(request, "home.html", context)
```

Link:https://docs.djangoproject.com/en/3.0/topics/forms/formsets/

1.6.9 DJANGO MODELFORMSETS

ModelFormsets in a Django is an advanced way of handling multiple forms created using a model and use them to create model instances. In other words, ModelFormsets are a group of forms in Django.

SYNTAX:

```
from django.forms import formset_factory
GeeksFormSet = modelformset_factory(GeeksModel)
```

CREATING AND USING MODELFORMSETS:

Suppose we have a project named mainapp and app named app;

In app/models.py;

```
# import the standard Django Model
# from built-in library
from django.db import models

# declare a new model with a name "GeeksModel"
class GeeksModel(models.Model):

        # fields of the model
        title = models.CharField(max_length = 200)
        description = models.TextField()

        # renames the instances of the model
        # with their title name
        def __str__(self):
                return self.title
```

In app/views.py;

```
from django.shortcuts import render

# relative import of forms
from .forms import GeeksForm

# importing formset_factory
from django.forms import formset_factory

def formset_view(request):
        context ={}

        # creating a formset
        GeeksFormSet = modelformset_factory(GeeksForm)
        formset = GeeksFormSet()

        # Add the formset to context dictionary
        context['formset']= formset
        return render(request, "home.html", context)
```

In templates/home.html;

```
<form method="POST" enctype="multipart/form-data">
   {% csrf_token %}
   {{ formset.as_p }}
   <input type="submit" value="Submit">
</form>
```

1.6.10 CREATING MULTIPLE FROMS USING DJANGO MODELFORMSETS:

-Django formsets are used to handle multiple instances of a form. One can create multiple forms easily using extra attribute of Django Formsets.

In app/views.py;

```
from django.shortcuts import render

# relative import of forms
from .models import GeeksModel

# importing formset_factory
from django.forms import modelformset_factory

def modelformset_view(request):
        context ={}

        # creating a formset and 5 instances of GeeksForm
        GeeksFormSet = modelformset_factory(GeeksModel, fields =['title', 'description'], extra
= 3)
        formset = GeeksFormSet()
        # Add the formset to context dictionary
        context['formset']= formset
        return render(request, "home.html", context)
```

HANDLING MULTIPLE FORMS USING DJANGO FORMSETS:
-Using {{formset.management_data}}

In templates/home.html;

```
<form method="POST" enctype="multipart/form-data">

        <!-- Management data of formset -->
        {{ formset.management_data }}

        <!-- Security token -->
        {% csrf_token %}

        <!-- Using the formset -->
        {{ formset.as_p }}

        <input type="submit" value="Submit">
</form>
```

Now to check how and what type of data is being rendered edit formset_view to print the data.

In app/views.py;

```
from django.shortcuts import render

# relative import of forms
from .forms import GeeksForm

# importing formset_factory
from django.forms import formset_factory

def formset_view(request):
        context ={}
```

```
    # creating a formset and 5 instances of GeeksForm
    GeeksFormSet = formset_factory(GeeksForm, extra = 3)
    formset = GeeksFormSet(request.POST or None)

    # print formset data if it is valid
    if formset.is_valid():
        for form in formset:
            print(form.cleaned_data)

    # Add the formset to context dictionary
    context['formset']= formset
    return render(request, "home.html", context)
```

DJANGO FORMS DATA TYPES AND FIELDS LIST

Name	Class	HTML Input
BooleanField	class BooleanField(**kwargs)	CheckboxInput
CharField	class CharField(**kwargs)	TextInput
ChoiceField	class ChoiceField(**kwargs)	Select
TypedChoiceField	class TypedChoiceField(**kwargs)	Select
DateField	class DateField(**kwargs)	DateInput
DateTimeField	class DateTimeField(**kwargs)	DateTimeInput
DecimalField	class DecimalField(**kwargs)	NumberInput when Field.localize is False, else TextInput
DurationField	class DurationField(**kwargs)	TextInput
EmailField	class EmailField(**kwargs	EmailInput
FileField	class FileField(**kwargs)	ClearableFileInput
FilePathField	class FilePathField(**kwargs)	Select
FloatField	class FloatField(**kwargs)	NumberInput when Field.localize is False, else TextInput
ImageField	class ImageField(**kwargs)	ClearableFileInput
IntegerField	class IntegerField(**kwargs)	NumberInput when Field.localize is False, else TextInput
GenericIPAddressField	class GenericIPAddressField(**kwargs)	TextInput
MultipleChoiceField	class MultipleChoiceField(**kwargs)	SelectMultiple
TypedMultipleChoiceField	class TypedMultipleChoiceField(**kwargs)	SelectMultiple

NullBooleanField	class NullBooleanField(**kwargs)	NullBooleanSelect
RegexField	class RegexField(**kwargs)	TextInput
SlugField	class SlugField(**kwargs)	TextInput
TimeField	class TimeField(**kwargs)	TimeInput
URLField	class URLField(**kwargs)	URLInput
UUIDField	class UUIDField(**kwargs)	TextInput

CORE FIELD ARGUMENTS:

Field Options	Description
required	By default, each Field class assumes the value is required, so to make it not required you need to set required=False
label	The label argument lets you specify the "human-friendly" label for this 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.
widget	The widget argument lets you specify a Widget class to use when rendering this Field. See Widgets for more information.
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 by one of the convenience Form methods.
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.
validators	The validators argument lets you provide a list of validation functions for this field.
localize	The localize argument enables the localization of form data input, as well as the rendered output.
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.

INITIAL-FORM-DATA DJANGO FORMS

Method 1: Adding initial form data in views.py;

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

def home_view(request):
        context ={}

        # dictionary for initial data with
        # field names as keys
        initial_dict = {
                "title" : "My New Title",
                "description" : " A New Description",
                "available":True,
                "email":"abc@gmail.com"
        }

        # add the dictionary during initialization
        form = GeeksForm(request.POST or None, initial = initial_dict)

        context['form']= form
        return render(request, "home.html", context)
```

Method 2:Adding initial form data using fields in forms.py;
Using initial attribute;

```
from django import forms
class GeeksForm(forms.Form):
        # adding initial data using initial attribute
        title = forms.CharField(initial = "Method 2 ")
        description = forms.CharField(initial = "Method 2 description")
        available = forms.BooleanField(initial = True)
        email = forms.EmailField(initial = "abc@gmail.com")
```

1.7 MISC

1.7.1 HANDLING AJAX REQUEST IN DJANGO

```
CLIENT(1 GOES 1 COME)
 ||
AJAX(1 GOES 1 COME)
 ||
DJANGO VIEW(1 GOES 1 COME)
```

EXAMPLE: POST-LIKING APP;

1. Initialize the django project:
 djangoa-admin startproject base .

2. creating an app called post
 python manage.py startapp post

3. Adding the app in settings .py

4. Creating models:
In post/models.py;

```
class Post(models.Model):
    post_heading = models.CharField(max_length=200)
    post_text = models.TextField()
    def __str__(self):
          return str(self.post_heading)

class Like(models.Model):   (Creating table for like)
    post = models.ForeignKey(Post, on_delete = models.CASCADE)
    user = models.ForeignKey(User, on_delete=models.CASCADE)  # Who liked the post
    created_at = models.DateTimeField(auto_now_add=True)  # Time when like was added

    class Meta:
        unique_together = ('post', 'user')
```

5. python manage.py makemigrations
   python manage.py migrate

6. Creating views;
In post/views.py;

```
from .models import Post, Like
from django.http import HttpResponse

def index(request):
    posts = Post.objects.all()  # Getting all the posts from database
    return render(request, 'post/index.html', { 'posts': posts })

def likePost(request):
    if request.method == 'GET':
        post_id = request.GET.get('post_id')
        post = Post.objects.get(pk=post_id)

        user = request.user  # Get the logged-in user

        # Check if the like already exists
        like, created = Like.objects.get_or_create(post=post, user=user)

        if created:
            return JsonResponse({"message": "Post liked!"})
        else:
            return JsonResponse({"message": "You already liked this post!"})

    return JsonResponse({"error": "Invalid request"}, status=400)
```

7. Creating urls;
In base/urls.py;

```
from django.urls import include, path
from django.contrib import admin
```

```
urlpatterns = [
     path('admin/', admin.site.urls),
     path('', include('post.urls')),   # To make post app available at /
   ]
```

In post/urls.py;

```
from django.urls import path
from . import views
urlpatterns = [
        path('', views.index, name='index'),  # index view at /
        path('likepost/', views.likePost, name='likepost'),   # likepost view at /likepost
   ]
```

8. Making templates and carring out ajax requests:

post/templates/post/index.html;

```
<!DOCTYPE html>
<html>
<head>
    <title>Like Post App</title>
</head>
<body>
    <p id="message"></p>
    {% for post in posts %}
    <h3>{{ forloop.counter }}) {{ post.post_heading }}</h3>
    <p>{{ post.post_text }} </p>
    <a class="likebutton" id="like{{post.id}}" href="#" data-catid="{{ post.id }}">Like</a>
    {% endfor %}
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.0/jquery.min.js"></script>
<script type="text/javascript">
$(document).ready(function() {
    function getCSRFToken() {
        return document.querySelector('[name=csrfmiddlewaretoken]').value;
    }

    $('.likebutton').click(function(event) {
        event.preventDefault();
        var catid = $(this).data("catid");

        $.ajax({
            type: "POST",
            url: "/likepost/",
            data: {
                post_id: catid,
                csrfmiddlewaretoken: getCSRFToken()
            },
            success: function(response) {
                $('#like' + catid).remove();
                $('#message').text(response.message);
            },
            error: function() {
                $('#message').text("An error occurred.");
            }
        });
    });
});
</script>

</body>
</html>
```

9. Register model in admin panel;
In post/admin.py;

```
from django.contrib import admin
from .models import Post, Like

class PostAdmin(admin.ModelAdmin):
    list_display = ('post_heading', 'post_text')  # Display columns in admin

class LikeAdmin(admin.ModelAdmin):
    list_display = ('post', 'user', 'created_at')
```

```
admin.site.register(Post)
admin.site.register(Like)
```

10. Add some post and run server.

1.7.2 USER GROUPS WITH CUSTOM PERMISSIONS IN DJANGO

Our main objective is to design and write code for the back-end in a very efficient way(following the DRY Principle).

There are multiple methods of implementing this in Django but the most suitable and efficient method is Grouping the Users and defining the permissions of these groups. User of that particular group will automatically inherit the permission of that particular group.

Create a app called user;

In user/models.py;

```
# importing necessary django classes
from django.contrib.auth.models import AbstractUser
from django.utils import timezone
from django.db import models

# User class
class User(AbstractUser):

    # Define the extra fields
    # related to User here
    first_name = models.CharField(_('First Name of User'),
                                            blank = True, max_length = 20)

    last_name = models.CharField(_('Last Name of User'),
                                            blank = True, max_length = 20)

# More User fields according to need

    # define the custom permissions
    # related to User.
```

```
        class Meta:

                permissions = (
                        ("can_go_in_non_ac_bus", "To provide non-AC Bus facility"),
                        ("can_go_in_ac_bus", "To provide AC-Bus facility"),
                        ("can_stay_ac-room", "To provide staying at AC room"),
                        ("can_stay_ac-room", "To provide staying at Non-AC room"),
                        ("can_go_dehradoon", "Trip to Dehradoon"),
                        ("can_go_mussoorie", "Trip to Mussoorie"),
                        ("can_go_haridwaar", "Trip to Haridwaar"),
                        ("can_go_rishikesh", "Trip to Rishikesh"),

# Add other custom permissions according to need.
```

After migrating the models written above, we have two option for making the group.

1. Django Admin Panel : In Admin Panel you will see Group in bold letter, Click on that and make 3-different group named level0, level1, level3 . Also, define the custom permissions according to the need.
2. By Programmatically creating a group with permissions: Open python shell using python manage.py shell.

```
# importing group class from django
from django.contrib.auth.models import Group, Permission
from django.contrib.contenttypes.models import ContentType

# import User model
from users.models import User

new_group, created = Group.objects.get_or_create(name ='new_group')

# Code to add permission to group
ct = ContentType.objects.get_for_model(User)

# If I want to add 'Can go Haridwar' permission to level0 ?
permission = Permission.objects.create(codename ='can_go_haridwar',
                                        name ='Can go to Haridwar',
content_type = ct)
new_group.permissions.add(permission)
```

We will set different set of permissions in the same way to all the three groups. Until then, we have made groups and linked it with custom permissions.

Now, check that a particular user is accessing the appropriate functionality like, put a limit that level0 does not access the functionalities of level1 users or level2 user and so on. To do this, check the permission on every view function made.

FOR FUNCTION BASED VIEW;we will use custom decorator;

```
@group_required('level0')
def my_view(request):
        ...
```

group_required() also takes an optional login_url parameter like:

```
@group_required('toto', login_url='/loginpage/')
```

As in the login_required() decorator, login_url defaults to settings.LOGIN_URL.

If the raise_exception parameter is given, the decorator will raise PermissionDenied, prompting the 403 (HTTP Forbidden) view instead of redirecting to the login page.

EXAMPLE:

```
from django.utils import six
from django.core.exceptions import PermissionDenied
from django.contrib.auth.decorators import user_passes_test

def group_required(group, login_url=None, raise_exception=False):
    """
    Decorator for views that checks whether a user has a group permission,
    redirecting to the log-in page if necessary.
    If the raise_exception parameter is given the PermissionDenied exception
    is raised.
    """
    def check_perms(user):
        if isinstance(group, six.string_types):
            groups = (group, )
        else:
            groups = group
        # First check if the user has the permission (even anon users)

        if user.groups.filter(name__in=groups).exists():
            return True
        # In case the 403 handler should be called raise the exception
        if raise_exception:
            raise PermissionDenied
        # As the last resort, show the login form
        return False
    return user_passes_test(check_perms, login_url=login_url)
```

FRO CLASS BASED VIEW:we can not simply just add a decorator function, but we have to make a permission-mixing class.

```
class GroupRequiredMixin(object):
            ...............
            ....Class Definition.....



class DemoView(GroupRequiredMixin, View):
group_required = [u'admin', u'manager']

# View code...
```

EXAMPLE:
In app/mixins.py;

```
from django.core.exceptions import PermissionDenied



class GroupRequiredMixin(object):
    """
        group_required - list of strings, required param
    """

    group_required = None

    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_authenticated():
            raise PermissionDenied
        else:
            user_groups = []
            for group in request.user.groups.values_list('name', flat=True):
                user_groups.append(group)
            if len(set(user_groups).intersection(self.group_required)) <= 0:
                raise PermissionDenied
        return super(GroupRequiredMixin, self).dispatch(request, *args, **kwargs)
```

In views.py;

```
from .mixins import GroupRequiredMixin
from django.views.generic import View

class DemoView(GroupRequiredMixin, View):
  group_required = [u'admin', u'manager']

  # View code...
```

NOTE:I had to change is_authenticated() to is_authenticated using python3 and django2

LINK:https://docs.djangoproject.com/en/1.11/topics/class-based-views/mixins/

LINK:https://bradmontgomery.blogspot.com/2009/04/restricting-access-by-group-in-django.html

LINK:https://simpleisbetterthancomplex.com/2015/12/07/working-with-django-view-decorators.html

LINK:https://micropyramid.com/blog/custom-decorators-to-check-user-roles-and-permissions-in-django

1.7.3 DJANGO ADMIN INTERFACE

To access admin inteface,we must create superuser by:
python manage.py createsuperuser

Then,runserver and log in,

RESET DJANGO ADMIN PASSWORD:

```
  python manage.py changepassword <username>
  python manage.py changepassword xeno
```

After creating models,we can register it in admin panel by:
At first import model at the top of admin.py file;

```
from .models import <model.name>
from .models import Faclity_details
```

Then,
admin.site.register<Model.name>
Like:
admin.site.register(Faclity_details)

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

1.7.4 DEALING WITH WARNINGS

TIPS:

1. We can change the name of the outer folder because it is just a folder containing your project but please don't change the name of the inner folder.
2. unapplied migrations: red warnings then simply we must do is python manage.py makemigrations and migrate

1.7.5 SESSIONS FRAMEWORK USING DJANGO

The sessions framework can be used to provide persistent behavior for anonymous users on the website. Sessions are the mechanism used by Django for you to store and retrieve data on a per-site-visitor basis. Django uses a cookie containing a unique session ID.

Django provides full support for anonymous sessions. The session framework lets you store and retrieve arbitrary data on a per-site-visitor basis. It keeps data on the server side and abstracts the sending and receiving of cookies.

HOW TO USE SESSION IN DJANGO:

1. Enable session in django:

   2 things to be considered i.e.

   MIDDLEWARE_CLASSES has 'django.contrib.sessions.middleware.SessionMiddleware' activate

   INSTALLED_APPS has 'django.contrib.sessions' added

Then,Session database table has to be created by:

COMMAND:

```
python manage.py syncdb
```

Once Sessions are created,then testing of the cookies has to be done.

CONFIGURE SESSION STORAGE:

-By default,Django uses a database-backend session storage.We can configure it in our settings.py.

To use database-backend session storage:

```
SESSION_ENGINE = 'django.contrib.sessions.backends.db'
```

or,

To use cache-based sessions (faster),we can configure it by:

```
SESSION_ENGINE = 'django.contrib.sessions.backends.cache'
```

Additionally,configure the cache settings in our settings.py file,

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
    }
}
```

USING SESSIONS
In app/views.py;

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

def home(request) :
        a = Album.objects.all()
        return render(request, "dhun/home.html ", {"Album":a})

def index(request) :
        num_authors = Author.objects.count()
        request.session.set_test_cookie()
        num_visits = request.session.get( 'num_visits', 0)
        request.session ['num_visits'] = num_visits + 1
        context ={
                'num_books':num_books,
                'num_instances':num_instances,
                'num_instances available':num_instances_available,
                'num_authors':num_authors,
                'num_visits':num_visits,
        }

def about(request):
        LANGUAGE_CODE ='en-us '
        TIME_ZONE ='UTC'
        if request.session.test_cookie_worked():
                print ("Cookie Tested !")
                request.session.delete_test_cookie()
```

Run the server and visit the about page;
The “Cookie Tested!” will be printed out to the console.

TO KNOW HOW MANY TIMES THE SITE HAS BEEN VISITED,We must do the following 2 things in views.py;
1. Add and update the code in the index view function
2. Update the about view function

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

def home(request):
        a = AIbum. objects.all()
        return render(request, "dhun/home.html", {"album":a})

def index(request):
        visits = int(reques.COOKIES.get('visits', '0'))
        response = HttpResponse(template.render(context))

        if request.COOKIES.has_key('last_visit'):
                last_visit = request. COOKIES [ ' last_visit']
                last_visit_time = datetime.strptime(last_visit[:-7], "%Y-%m-%d %H:%M:%S") "
                curr_time = datetime.now()
                if (curr_time—last_visit_time).days > O:
                        response.set_cookie( 'visits ', visits + 1)
                        response. set_cookie( ' last_visit', datetime.now())
                else :
                        response.set_cookie( ' last_visit', datetime.now())
                return response

def about(request) :
        context = RequestContext(request)
        if request.COOKIES.has_key(' visits '):
                v = request.COOKIES [' visits ']
        else :
            v = 0
        return render_to_response('music/about.html', { 'visits':v}, context)
```

SESSION EXPIRY:

-By default,Django sessions expire when the user's browser is closed.We can configure session expiry by setting the SESSION_COOKIE_AGE setting in settings.py file.

```
SESSION_COOKIE_AGE = 1800  # 30 minutes (in seconds)
```

1.7.6 EXTENDING AND CUSTOMIZING DJANGO-ALLAUTH

EXTENDING THE SIGNUP FORM OR ADDING THE CUSTOM FIELDS IN DJANGO-ALLAUTH

-Adding addictional fields or custom fields to the signup form.We can extend the SignupForm class from allauth.account.forms by creating a custom class and passing the SignupForm to the custom class and defining the custom fields and save it.We must return the user object as it will be passed to other modules for validation. We also need to include a variable in settings.py;

In forms.py;

```
from allauth.account.forms import SignupForm
from django import forms

class CustomSignupForm(SignupForm):
   first_name = forms.CharField(max_length=30, label='First Name')
   last_name = forms.CharField(max_length=30, label='Last Name')

   def save(self, request):
        user = super(CustomSignupForm, self).save(request)
       user.first_name = self.cleaned_data['first_name']
       user.last_name = self.cleaned_data['last_name']
       user.save()
       return user
```

Here,CustomSignupForm is extended the class which inherits all the features of SignupForm class and adds the necessary features. Here custom fields by the name first_nameand last_name are created and saved using the signup module in the same class.

In settings.py;

```
ACCOUNT_FORMS = {
'signup': 'YourProject.forms.CustomSignupForm',
}
```

Any other custom forms created can be extended in ACCOUNT_FORMS.

Similarly, LoginForm UserForm AddEmailForm and others can be extended. However, remember that when you extend these forms and link them in settings.py. Don't forget to pass the original form (For example SignupForm) as a parameter to your class and sometimes we might have to handle custom validations and return users or some other value.

LINK:https://github.com/pennersr/django-allauth/blob/main/allauth/account/forms.py#L362

USER INTERVENTION AND CUSTOM VALIDATIONS IN USER REGISTRATION FLOW:
-DefaultAccountAdapter is very useful and indeed can be used to solve most of the customization problems that a user might encounter while using django–allauth.

FOR EXAMPLE 1: RESTRICTED LIST OF EMAILS

After figuring out a way to store and fetch the restricted list, you can use the adapters and raise validation errors in the registration form when a restricted email tries to register. Extend a DefaultAccountAdapter and override the clean_email method.

Create an adapter.py in your project directory and extend the default adapter class.

```
from allauth.account.adapter import DefaultAccountAdapter
from django.forms import ValidationError

class RestrictEmailAdapter(DefaultAccountAdapter):
    def clean_email(self, email):
        RestrictedList = ['Your restricted list goes here.']
        if email in RestrictedList
            raise ValidationError('You are restricted from registering.\
                                  Please contact admin.')
        return email
```

Finally, point the account adapter in settings.py to your extended class.

```
ACCOUNT_ADAPTER='YourProject.adapter.RestrictEmailAdapter'
```

FOR EXAMPLE 2: ADD A MAXIMUM LENGTH TO A USERNAME

Extend the DefaultAccountAdapterclass and overriding the clean_username method. You need to also reference the clean_username once again after our custom validation to complete other inbuilt validations.

The last sentence in the above paragraph is the key to work with DefaultAccountAdapter. You should never forget to reference the original module name for the module to complete other validations.

```
from allauth.account.adapter import DefaultAccountAdapter
from django.forms import ValidationError

class UsernameMaxAdapter(DefaultAccountAdapter):
    def clean_username(self, username):
        if len(username) &gt; 'Your Max Size':
            raise ValidationError('Please enter a username value\
                            less than the current one')

        # For other default validations.
        return DefaultAccountAdapter.clean_username(self, username)
```

Finally, point to the subclass in your settings.py

```
ACCOUNT_ADAPTER = 'YourProject.adapter.UsernameMaxAdapter'
```

LINK OF ADAPTERS.PY TO EXTEND OTHER MODULES:
https://github.com/pennersr/django-allauth/blob/main/allauth/account/adapter.py

The modules such as populate_username, clean_password (which can be customized to restrict commonly used passwords) can have the custom process and flow without rewriting them.

DefaultAccountAdapter can be a potent tool if used in the right circumstances to intervene in allauth's default process. allauth comes with a vast variety of inbuilt settings, and they are here.

ENTIRE CODE
LINK:https://stackoverflow.com/questions/50924482/django-allauth-restrict-registration-to-list-of-emails/50934047#50934047

1.7.7 DJANGO SIGN UP AND LOGIN WITH CONFIRMATION EMAIL

AUTHENTICATION SYSTEM:
LINK:https://www.geeksforgeeks.org/django-sign-up-and-login-with-confirmation-email-python/

Django Sign Up and Login with Confirmation Email

1. At first install crispy_forms by:

pip install --upgrade django-crispy-forms

pip install crispy-bootstrap4

2. Starting a project by:

django-admin startproject project .

3. Creating a app:

python manage.py startapp user

4. Then,In user/templates/user then index.html,login.html,Email.html,register.html files
5. Now add the "user" app and "crispy_form" in your todo_site in settings.py, and add CRISPY_TEMPLATE_PACK = 'bootstrap4' at last of settings.py ;

Like:

'user',

'crispy_forms',

'crispy_bootstrap4',

CRISPY_TEMPLATE_PACK = 'bootstrap4'

Configure email settings in settings.py;
i.e.

```
EMAIL_BACKEND='django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST='smtp.gmail.com'
EMAIL_PORT=587
EMAIL_USE_TLS=True
EMAIL_HOST_USER="Your email"
EMAIL_HOST_PASSWORD="Your password"
```

6. Edit project/urls.py ;

```
from django.contrib import admin
from django.urls import path, include
from user import views as user_view
from django.contrib.auth import views as auth

urlpatterns = [
        path('admin/', admin.site.urls),

        ##### user related path##########################
        path('', include('user.urls')),
        path('login/', user_view.Login, name ='login'),
        path('logout/', user_view.logout_view, name ='logout'),
        path('register/', user_view.register, name ='register'),

]
```

7. Edit user/urls.py;

```
from django.urls import path, include
from django.conf import settings
from . import views
from django.conf.urls.static import static

urlpatterns = [
          path('', views.index, name ='index'),
]
```

8. Edit views.py in user

```
from django.shortcuts import render, redirect
from django.contrib import messages
from django.contrib.auth import authenticate, login,logout
from django.contrib.auth.decorators import login_required
from django.contrib.auth.forms import AuthenticationForm
from .forms import UserRegisterForm
from django.core.mail import send_mail
from django.core.mail import EmailMultiAlternatives
from django.template.loader import get_template
from django.template import Context



#################### index#######################################
def index(request):
     return render(request, 'user/index.html', {'title':'index'})

########### register here #####################################
def register(request):
     if request.method == 'POST':
          form = UserRegisterForm(request.POST)
          if form.is_valid():
               form.save()
               username = form.cleaned_data.get('username')
               email = form.cleaned_data.get('email')
               ######################### mail system
####################################
               htmly = get_template('user/Email.html')
               d = { 'username': username }
               subject, from_email, to = 'welcome', 'your_email@gmail.com', email
               html_content = htmly.render(d)
               msg = EmailMultiAlternatives(subject, html_content, from_email, [to])
```

```
                              msg.attach_alternative(html_content, "text/html")
                              msg.send()

####################################################################
                              messages.success(request, f'Your account has been created ! You are now
able to log in')
                              return redirect('login')
          else:
                    form = UserRegisterForm()
          return render(request, 'user/register.html', {'form': form, 'title':'register here'})

################ login forms####################################################
def Login(request):
          if request.method == 'POST':

                    # AuthenticationForm_can_also_be_used__

                    username = request.POST['username']
                    password = request.POST['password']
                    user = authenticate(request, username = username, password = password)
                    if user is not None:
                              form = login(request, user)
                              messages.success(request, f' welcome {username} !!')
                              return redirect('index')
                    else:
                              messages.info(request, f'account done not exit plz sign in')
          form = AuthenticationForm()
          return render(request, 'user/login.html', {'form':form, 'title':'log in'})

def logout_view(request):
     logout(request)
     return redirect(‘index’)
```

9. Create user/forms.py;

```
from django import forms
from django.contrib.auth.models import User
from django.contrib.auth.forms import UserCreationForm

class UserRegisterForm(UserCreationForm):
        email = forms.EmailField()
        phone_no = forms.CharField(max_length = 20)
        first_name = forms.CharField(max_length = 20)
        last_name = forms.CharField(max_length = 20)
        class Meta:
                model = User
                fields = ['username', 'email', 'phone_no', 'password1', 'password2']
```

10. Navigate to templates/user/ and edit files;

index.html file

This file includes metadata, loads external CSS and JavaScript files (Bootstrap and Font Awesome), and uses Django template tags to handle dynamic content. The template features a navigation bar, displays alert messages, and adjusts the page content based on user authentication, showing a personalized welcome message or a login prompt. This code is designed for building user-friendly web interfaces within a Django project.

```
{% load static %}
{% load crispy_forms_tags %}
<!DOCTYPE html>
<html lang="en">

<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<meta name="title" content="project">
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<meta name="language" content="English">
<meta name="author" content="vinayak sharma">

<title>{{title}}</title>
```

```
<!-- bootstrap file -->
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" />
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>
<!-- bootstrap file-->

<!-- jQuery -->
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi
6jizo" crossorigin="anonymous"></script>

<link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css">



<!-- main css -->
<link rel="stylesheet" type="text/css" href="{% static "index.css" %}" />



<!-- message here -->

{% if messages %}
{% for message in messages %}

<script>
        alert("{{ message }}");
</script>

{% endfor %}
{% endif %}

<!--_______________________________________________-->




</head>
```

```
<body class="container-fluid">



<header class="row">

        <!-- navbar-->
        <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container-fluid">
                <div class="navbar-header">
                <button class="navbar-toggle" data-toggle="collapse"
data-target="#mainNavBar">
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                </button>
                <a class="navbar-brand" class="styleheader" href="{% url "index"
%}">project</a>
                </div>
                <div class="collapse navbar-collapse" id="mainNavBar">
                <ul class="nav navbar-nav navbar-right">
                        <li><a href="{% url "index" %}">Home</a></li>

                        {% if user.is_authenticated %}
                        <li><a href="{% url "logout" %}"><span class="glyphicon
glyphicon-log-out"></span>  Logout</a></li>
                        {% else %}
                        <li><a href="{% url "register" %}"><span class="glyphicon
glyphicon-user"></span>  Sign up</a></li>
                        <li><a href="{% url "login" %}"><span class="glyphicon
glyphicon-log-in"></span>  Log in</a></li>
                        {% endif %}

                </ul>
                </div>
        </div>
        </nav>
</header>
<br/>
<br>
<br>
<div class="row">
        {% block start %}
        {% if user.is_authenticated %}
        <center><h1>welcome back {{user.username}}!</h1></center>
        {% else %}
        <center><h1>log in, plz . . .</h1></center>
        {% endif %}
        {% endblock %}
</div>
</body>

</html>
```

Email.html:

The provided HTML code is an email template for a registration confirmation message. It uses the Roboto font, has a centered thank-you message with user-specific content (username), and a horizontal line for separation. This template is designed to deliver a visually pleasing and informative confirmation email to users.

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
        <head>
                <meta charset="utf-8">
                <title></title>
                <style>
                        @import
url('https://fonts.googleapis.com/css?family=Roboto:400,100,300,500,700,900');
                </style>
        </head>
        <body style="background: #f5f8fa;font-family: 'Roboto', sans-serif;">
```

```
                <div style="width: 90%;max-width:600px;margin: 20px auto;background:
#ffffff;">
                        <section style="margin: 0 15px;color:#425b76;">
                                <h2 style="margin: 40px 0 27px 0;text-align: center;">Thank you
to registration</h2>
                                <hr style="border:0;border-top: 1px solid
rgba(66,91,118,0.3);max-width: 50%">
                                <p style="font-size:15.5px;font-weight: bold;margin:40px 20px
15px 20px;">Hi {{username}}, we have received your details and will process soon.</p>
                        </section>
                </div>
        </body>
</html>
```

Login.html

Inside this block, it creates a centered login form with specific styling, including a black border, padding, and a rounded border. The form includes a CSRF token for security and uses the crispy filter to render form fields with enhanced formatting, along with a login button and a link to the registration page.

```
{% extends "user/index.html" %}
{% load crispy_forms_tags %}
{% block start %}

<div class="content-section col-md-8 col-md-offset-2">
<center>
<form method="POST" style="border: 1px solid black; margin: 4%; padding:10%;
border-radius:1%;">
        {% csrf_token %}
        <fieldset class="form-group">
        {{ form|crispy}}
        </fieldset>
<center>
        <button style="background: black; font-size: 2rem; padding:1%;" class="btn
btn-outline-info" type="submit"><span class="glyphicon glyphicon-log-in"></span>
login</button>
```

```
</center>
<br/>
<sub style="text-align: left;"><a href="{% url 'register' %}" style="text-decoration: none; color:
blue; padding:2%; cursor:pointer; margin-right:2%;">don't have account,sign up</a></sub>
</form>
</center>
</div>
{% endblock start %}
```

Register.html:

This file creates a centered sign-up form with specific styling, including a black border, padding, and rounded corners. The form includes a CSRF token for security and uses the crispy filter for enhanced form field rendering, along with a sign-up button and a link to the login page for users with existing accounts.

```
{% extends "user/index.html" %}
{% load crispy_forms_tags %}
{% block start %}

<div class="content-section col-md-8 col-md-offset-2">
<form method="POST" style="border: 1px solid black; margin: 4%; padding:10%;
border-radius:1%;">
        {% csrf_token %}
        <fieldset class="form-group">
        {{ form|crispy}}
        </fieldset>
        <center>
        <button style="background: black; padding:2%; font-size: 2rem; color:white;" class="btn
btn-outline-info" type="submit"><span class="glyphicon glyphicon-check"></span>  sign
up</button>
        </center>
        <br />
        <sub><a href="{% url "login" %}" style="text-decoration: none; color: blue;
padding:3%; cursor:pointer;">Already have an account ?</a></sub>
</form>
</div>
{% endblock start %}
```

11. Make migrations and migrate them:

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

12. Run the server:

```
python manage.py runserver
```