lendi Institute of Engineering & Technology
An Autonomous Institution
Accredited by NAAC with '' A'' Grade, Accredited by NBA (ECE, CSE.EEE & MECH)
Approved by A.I.C.T.E. & Permanently Affiliated to J. N. T. U. Gurajada, VIZIANAGARAM
Via 5th APSP Battalion, Jonnada (V), Denkada (M), NH-3, Vizianagaram Dist - 535005, A.P. Website : www.lendi.org
Ph : 08922-241111, 241666, Cell No : 9490344747, 9490304747,e-mail : lendi_2008@yahoo.com

DJANGO FRAMEWORK

LABORATARY MANUAL

DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION TECHNOLGOY

VISION

To excel in computing arena and to produce globally competent computer science and Information Technology graduates with Ethical and Human values to serve the society

MISSION

- To impart strong theoretical and practical background in computer science and information technology discipline with an emphasis on software development.
- To provide an open environment to the students and faculty that promotes professional growth
- To inculcate the skills necessary to continue their education and research for contribution to society.

COURSE OUTCOMES (CO's)

CO1: Understand the environment of DJango Web Server Framework.

CO2: Create URL Mappings and Views using Templates.

CO3: Create DJango models for processing data from templates

CO4: Understand DJango Forms and Signals

CO5: Implement Restfull APIs using DJango Rest Framework

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

PEO1: Graduates of Computer Science and Information Technology will acquire strong knowledge to analyze, design, and develop computing products and solutions for real-life problems utilizing the latest tools, techniques, and technologies.

PEO2: Graduates of Computer Science and Information Technology shall have interdisciplinary approach, professional attitude and ethics, communication, teamwork skills and leadership capabilities to solve social issues through their Employment, Higher Studies and Research.

PEO3: Graduates will engage in life-long learning and professional development to adapt to dynamically computing environment.

PROGRAM OUTCOMES (POs)

PO1: Engineering Knowledge
PO2: Problem Analysis
PO3: Design & Development
PO4: Investigations
PO5: Modern Tools
PO6: Engineer & Society
PO7: Environment & Sustainability
PO8: Ethics
PO9: Individual & Team Work
PO10: Communication Skills
PO11: Project Mgt & Finance
PO12: Life Long Learning

PROGRAM SPECIFIC OUTCOMES (PSOs)

PSO1: Ability to solve contemporary issues utilizing skills.
PSO2: To acquire knowledge of the latest tools and technologies to provide technical solutions
PSO3: To qualify in national and international competitive examinations for successful higher studies and employment.

DJANGO FRAME WORK LAB SYLLABUS

Module 1: DJANGO FRAMEWORK- Introduction to DJango, Features of DJango, Application areas of DJango, Flask vs DJango, DJango Components, Install and Configure DJango Components.

List of Programs:

1. Create DJango environment setup and installation in Windows/Linux. (L5)
2. Create DJango Projct and app structure with django-admin commands. (L5)
3. Deployment of Project in server. (L2)

Module 2: DJANGO TEMPLATES: URLs, Views, Static Files, Images, Forms, Application development using Templates, Template Objects, tags, Filters, Loops and Inheritance.

List of Programs:

1. Create template in DJango Project to process user interface. (L5)
2. Create multiple routes from using Django URLS. (L5)
3. Implement template inheritance with views and images. (L3)

Module 3: DJANGO MODELS: Introduction to DJango Models, Admin Panel, Database Relationships, One-One, One- Many, Many-Many, Model Queries, Rendering Data to Templates, Dynamic URLs and Routing, CRUD operations.

List of Programs:

1. Create database configuration in DJango admin with Sqlite3. (L5)
2. Implement CRUD operations using Django models. (L3)
3. Implement database relationships using django models. (L3)
4. Implement data rendering with templates and dynamic routing. (L3)

Module 4: DYNAMIC FORMS & SIGNALS: Inline Form sets, Search Forms, User Registration and Login Authentication, User Roles & Permissions, User Profiles, Image Upload, DJango Signals, Creating customer profiles with DJango.

List of Programs:

1. Create user registration and login authentication using django forms. (L5)
2. Implement the roles and permissions for user profile. (L3)

3. Implement Django signal for user profiles. (L3)

Module 5 : DJANGO REST FRAMEWORK: Introduction to DJango Rest Framework**,** Features of Rest APIs, Installation of DJango Rest Framework, api_view, Response, JSONResponse, Models and Serializers, PATH and urlpatterns, HTTP methods GET, POST, PUT and DELETE methods

List of Programs:

1. Install and configure DJango Rest framework package. (L2)
2. Create django rest end points using api_view and JSONResponse. (L5)
3. Create Django rest end points using Serializers and Models. (L5)
4. Implement HTTP rest end points with all CRUD operations. (L3)

COURSE OUTCOMES Vs PO's & PSO's

SNO	DESCRIPTION	PO(1..12) MAPPING	PSO(1..3) MAPPING
SC3201.1	Understand the environment of DJango Web Server Framework.	PO2,PO3	PSO1,PSO2
SC3201.2	Create URL Mappings and Views using Templates.	PO3,PO11	PSO1
SC3201.3	Create DJango models for processing data from templates	PO3,PO11	PSO2
SC3201.4	Understand DJango Forms and Signals	PO1,PO6	PSO1
SC3201.5	Implement Restfull APIs using DJango Rest Framework	PO3,PO6	PSO2
SC3201.*	Able to analyze the real world problem and implement using Restfull APIs Developing The applications that relates to Services which are Enterprise.	PO1,PO2,PO3,PO5 PO6,PO11	PSO1,PSO2 PSO3
COURSE OVERALL PO/PSO MAPPING:			

SYLLABUS INDEX

13	Implement Django signal for user profiles.	49-51	CO1,CO2	PO1,PO2, PO3,PSO1
PROGRAMS BEYOND SYLLABUS				
14	Real-Time Collaborative Application with Django and WebSockets	61	CO1,CO2,CO3,CO4,CO5,CO6	PO1,PO2, PO3,PO11 PSO1,PSO2
15	Django with Redis for Caching and Session Management	69	CO1,CO2,CO3,CO4,CO5,CO6	PO1,PO2, PO3,PSO1 PSO2
OPEN ENDED EXPERIMENTS				
16	Develop a Multi-Tenant SaaS Application	70		
17	Create a Social Media Platform with Django	74		

Instructions to students

Pre-lab activities:

- Prepare observation note book which contains the following :
  - Procedure/algorithm/program to solve the problems discussed in the theory class
  - Solutions to the exercises given in the previous lab session
- Refer the topics covered in theory class

In-lab activities:

- Note down errors observed while executing program and remedy for that.
- Note down corrections made to the code during the lab session
- Answer to vivo-voce
- Get the observation corrected
- Note down inferences on the topic covered by the programs executed

Post-lab activities:

- Solve the given exercises
- Devise possible enhancements that can be made to the solved problem to simplify the logic
- Executed programs should be recorded in the lab record and corrected within one week after completion of the experiment.
- After completion of every module, a test will be conducted, and assessment results will have weight in the final internal marks.

General Instructions:

- Student should sign in the log register before accessing the system.
- Student is only responsible for any damage caused to the equipment in the laboratory during his session.
- Usage of pen drives is not allowed in the lab.
- If a problem is observed in any hardware equipment, please report to the lab staff immediately; do no attempt to fix the problem yourself.
- Systems must be shut down properly before leaving the lab.
- Please be considerate of those around you, especially in terms of noise level. While labs are a natural place for conversations regarding programming, kindly keep the volume turned down

MODULE-1

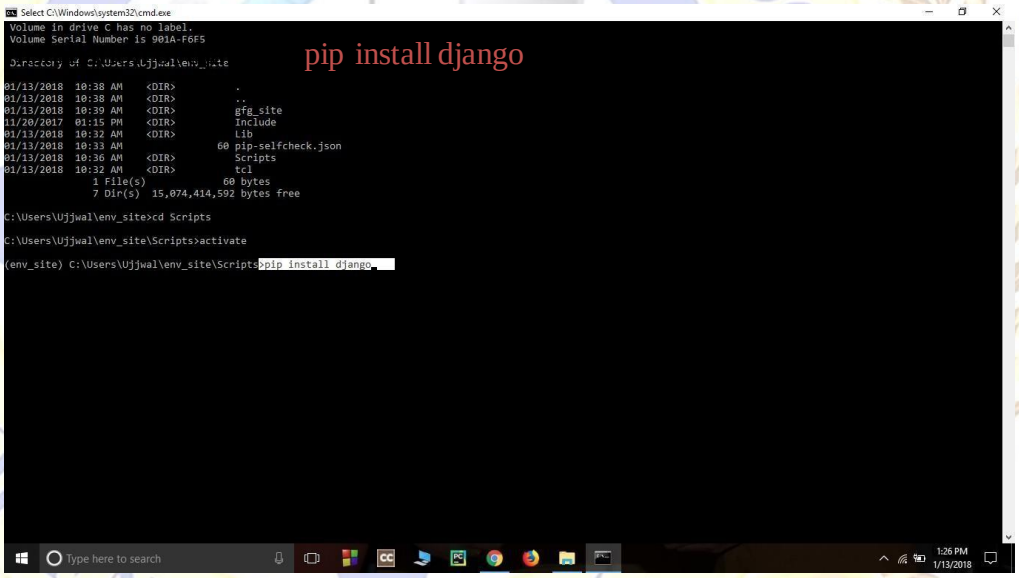
1A) AIM: Create Django environment setup and installation in windows/Linux

DESCRIPTION:

- First go to search bar and search CMD
- Move E drive by using command e: enter
- Then we get E:\>
- Create a folder by using cmd : mkdir <folder name> (press enter)
- Move to folder using cmd : cd <folder name> (press enter)

OUTPUT: E:\<<FOLDER NAME>>

- pip install django



2) AIM : Create DJANGO project and app structure with django-admin commands

DESCRIPTION :

To know commands of django use cmd : >django-admin Output

C:\Users\lendi>django-admin

Type 'django-admin help <subcommand>' for help on a specific subcommand.

Available subcommands: [django]

- check
- compilemessages

- createcachetable

dbshell

- diffsettings
- dumpdata
- flush
- inspectdb
- loaddata
- makemessages
- makemigrations
- migrate
- optimizemigration
- runserver
- sendtestemail
- shell
- showmigrations
- sqlflush
- sqlmigrate
- sqlsequencereset
- squashmigrations
- startapp
- startproject
- test
- testserver

These are the commands available in django

Steps to create project :

E:/<folder-name>>django-admin startproject <project name> (press enter)

Steps for creating app inside project :

E:\<FOLDER-NAME>> cd <project name> (press enter)

E:\<FOLDER-NAME>/<project name> django-admin startapp <app name> (press enter)

OUTPUT :

C:\Users\lendi>e:

E:\>mkdir django5b1

E:\>cd django5b1

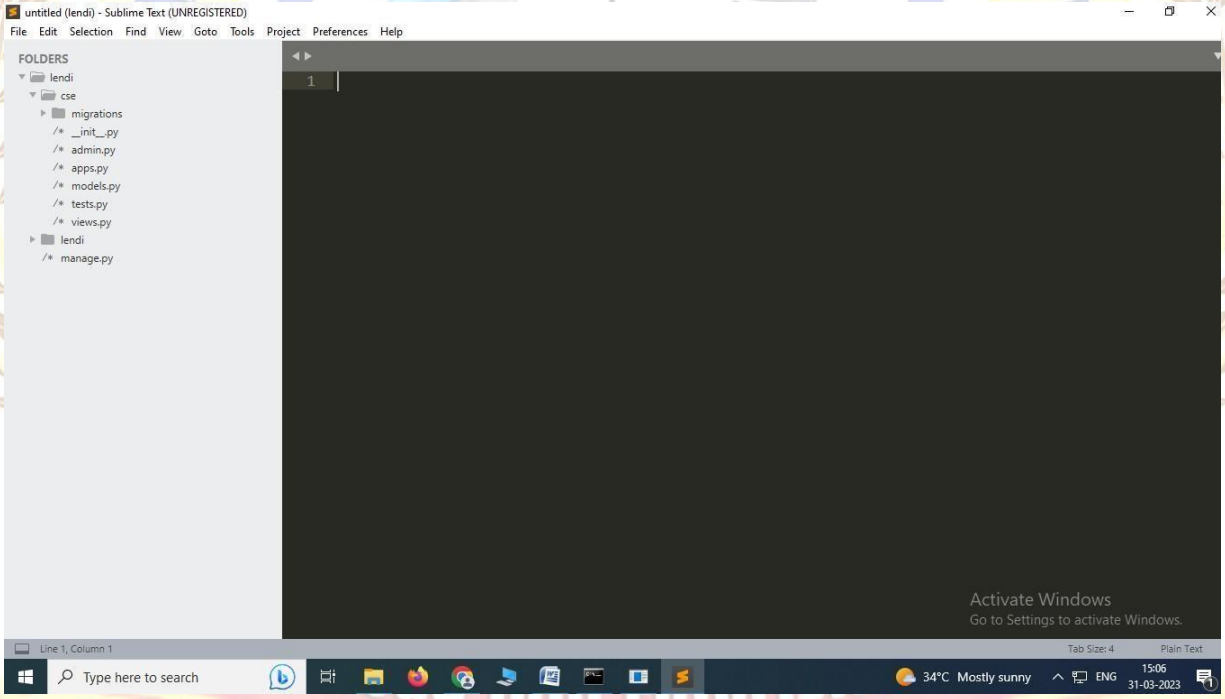
E:\django5b1>django-admin startproject lendi

E:\django5b1>cd lendi

E:\django5b1\lendi>django-admin startapp cse

GO TO SUBLIME TEXT (OR) VISUAL STUDIO CODE:

Drag the file lendi and drop in sublime text :

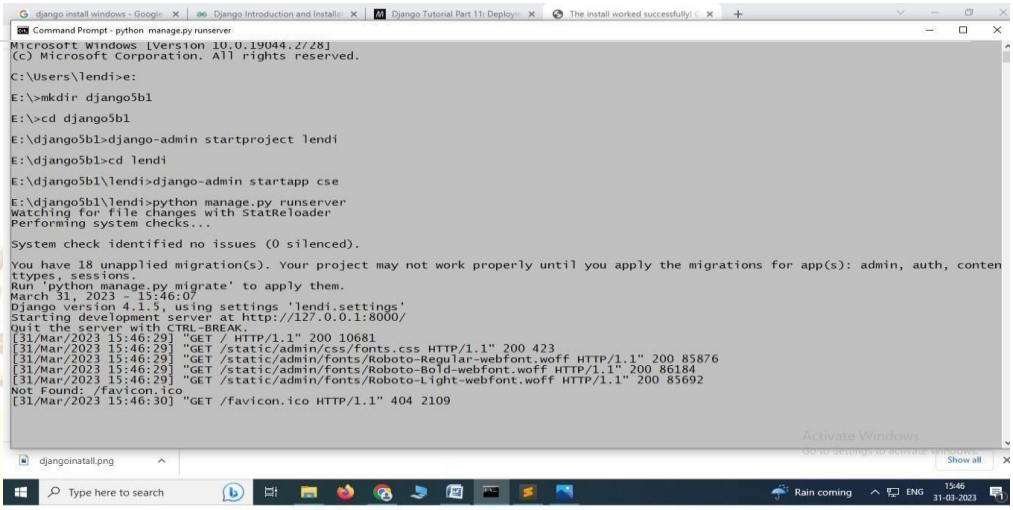


3. AIM :Deployment of project in server

DESCRIPTION:

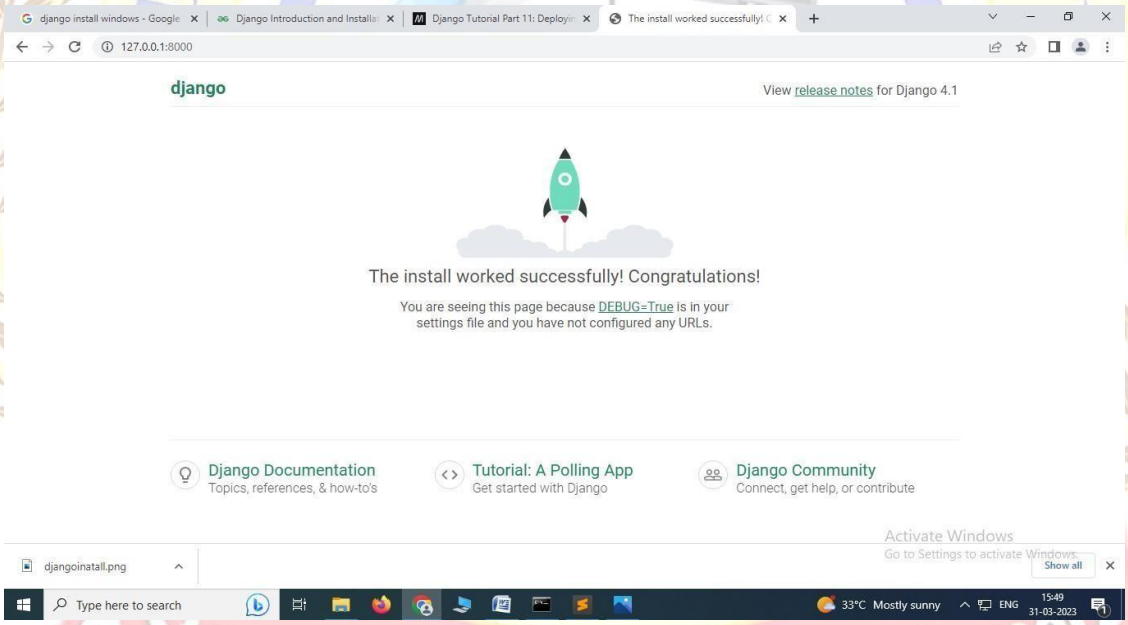
Steps for deployment

- Using cmd : E:\django5b1\lendi>python manage.py runserver



Using : http://127.0.0.1:8000/ run server

in chromeOUTPUT :



VIVA QUESTIONS :

1. What is Django and why is it used in web development?

2. Can you explain the concept of "MTV" in Django?

3. What are Django's main features?.

4. How does Django handle URLs and routing in a web application?

5. What are Django models, and how do they work?

6. How do you create a Django project and a Django app?

7. What is the role of Django templates?.

8. What is the purpose of the `settings.py` file in Django?

9. What is Django's Admin interface, and how can you customize it?

10. How does Django handle forms and form validation?

MODULE-2

1. AIM : Create template in Django Project to process user interface

DESCRIPTION: Step 1: connect to drive

>D:

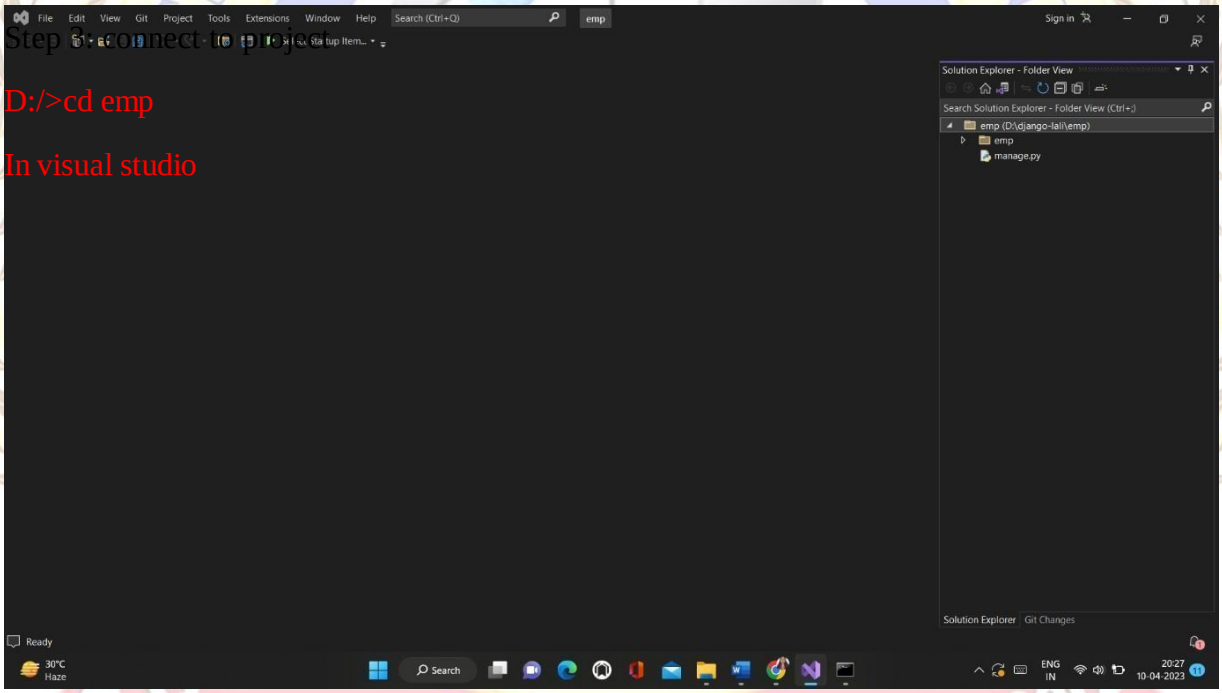
Step 2: create project

D:/>django-admin startproject emp

Step 3: connect to project

D:/>cd emp

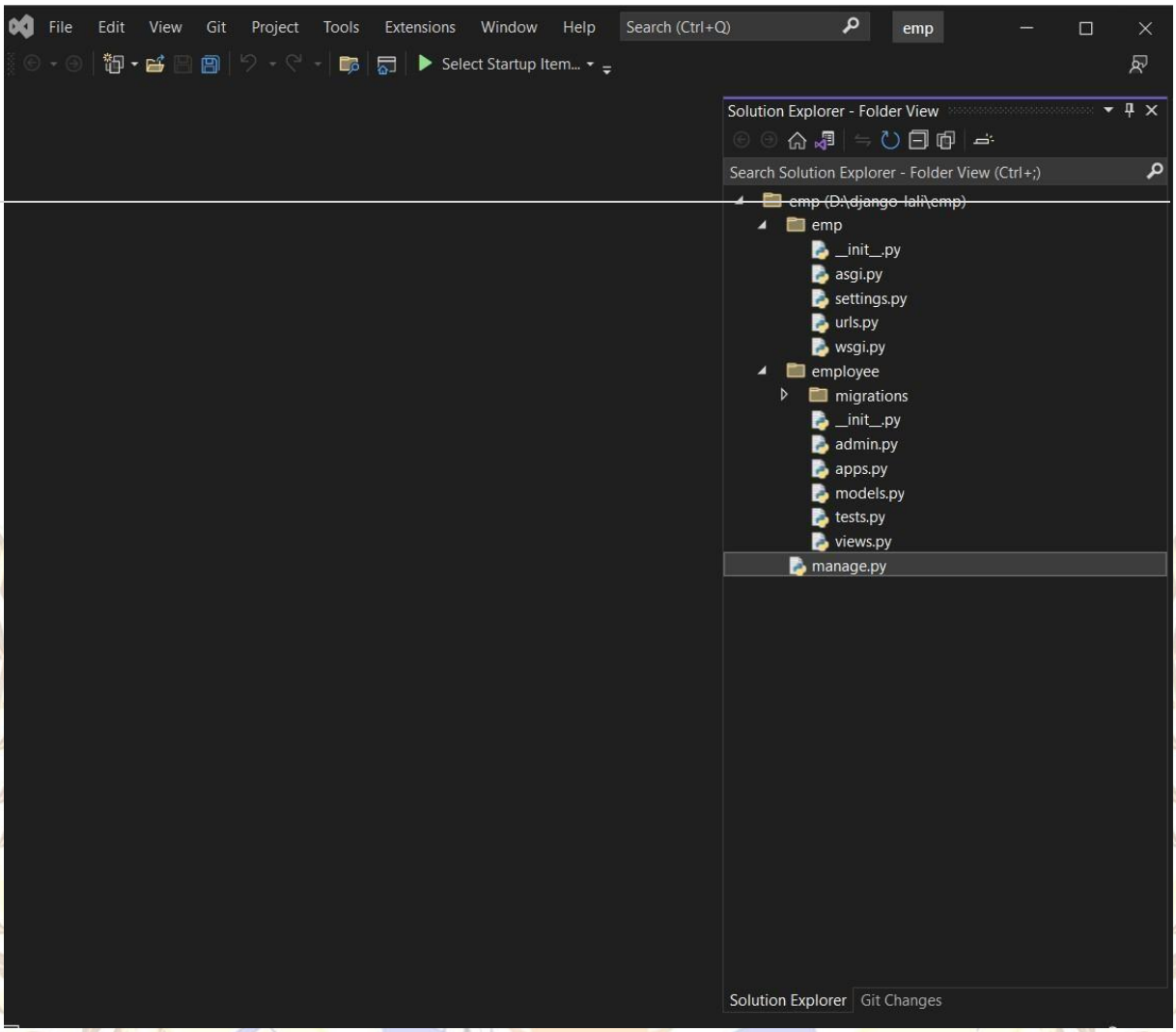
In visual studio



Step 4: creating app inside project

D:/>django-admin startproject employee

Step 5: open folder in sublime text or visual studio

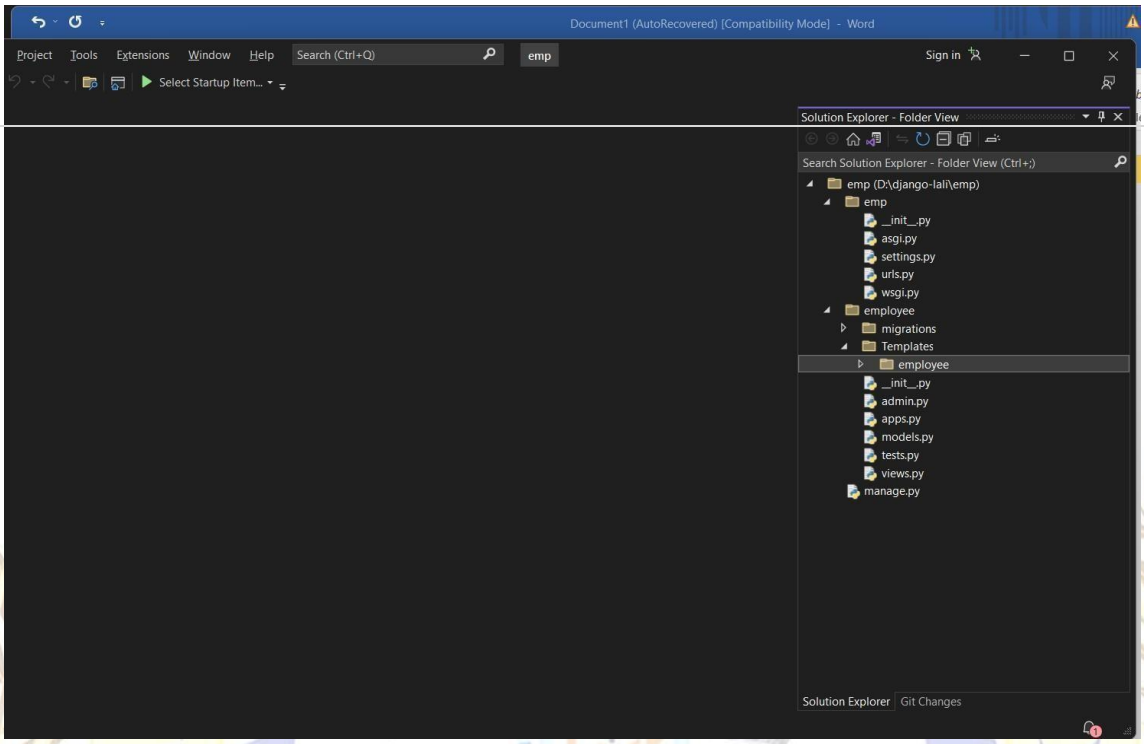


Step 6: create template inside employee

Click on employee new ->new folder-> name it as templates

Step 7: Create employee as folder inside template

Templates-> new-> new folder-> employee



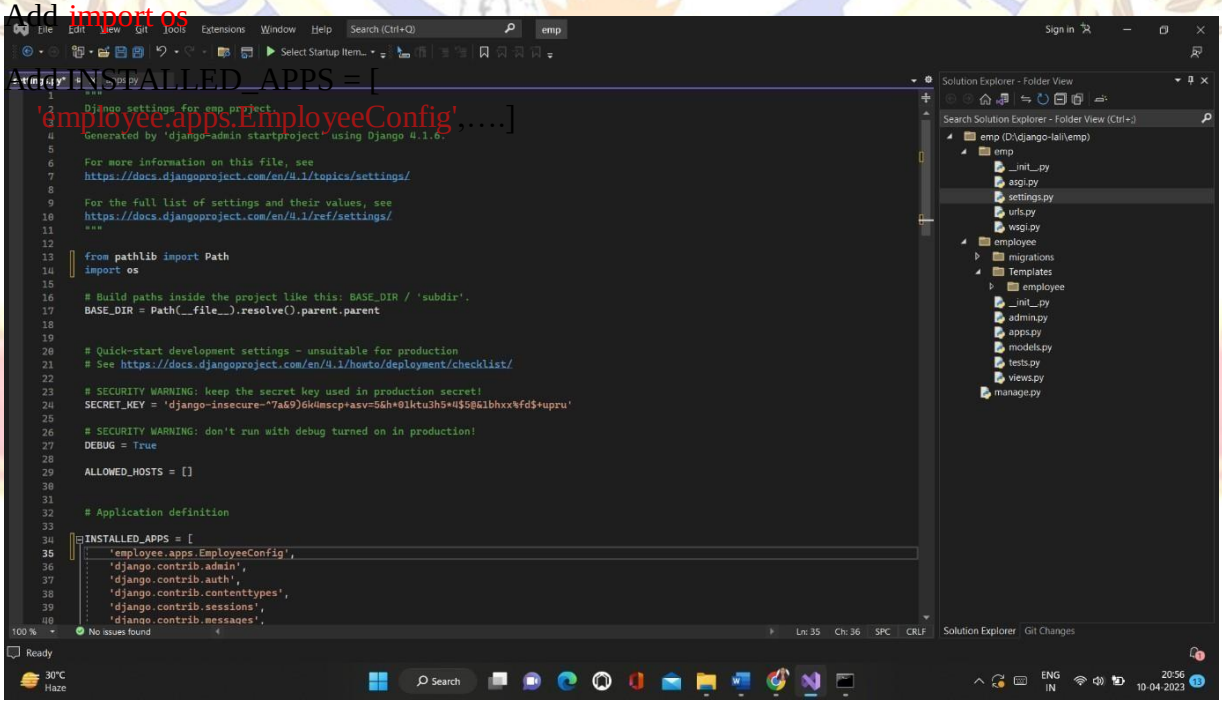
Step 8: move to employee- apps.py-copy EmployeeConfig

Step 9:in emp go to settings.py-

Add import os

Add INSTALLED_APPS = [

'employee.apps.EmployeeConfig',....]



Step 10:in employee->templates->employee-> create Home.html(new file)

In Home.html write a basic html code

```
<html>
<head>
<title></title>
</head>
<body>
  <h1> Welcome </h1>
</body>
</html>
```

Step 11: employee->views.py

Add code

```
def emphome(request):
    return render(request,'employee/Home.html')
```

Step 12: in employee create urls.py(new file)

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

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

Save changes

Step 13: in emps->urls.py Add code

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

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

Step 14: save changes

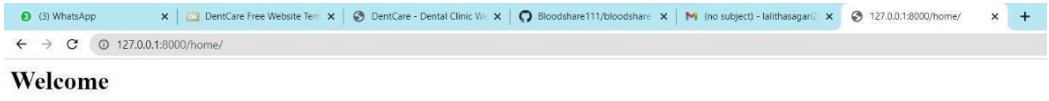
Step 15: go cmd for running server

D:\django\emp>python manage.py runserver

Copy http://127.0.0.1:8000/

Goto chrome and run server http://127.0.0.1:8000/home/

OUTPUT :



2. AIM :Create multiple routes from using Django URL's.

DESCRIPTION: **urls.py** file in your app's

Step 1: To create a new URL route in Django, you'll need to edit the directory. This file contains a list of URL patterns that the Django router uses to match incoming requests to view functions.(for same project as in module-2.1)

Step 2: in views.py add code

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

def home(request):
    return HttpResponse("Hello, world!")

def about(request):
    return HttpResponse("<h1> WElcome to home page")

# Create your views here.
```

Step 3: in employee->urls.py

Add routes

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

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

Step 4: emp->urls.pyfrom django.contrib

```
 import admin from django.urls

import path,include from.import views
urlpatterns = [
    path('', views.home),
```

```
    path('about/', views.about),
]
```

In this code, we've imported the path function from the django.urls module, as well as two view functions from our app's views.py file. We've then defined a list of URL patterns using the urlpatterns variable.

The first URL pattern, path('', views.index, name='index'), maps the root URL of our app (i.e., http://localhost:8000/) to the index view function. The name parameter is an optional identifier that we can use to refer to this URL pattern later in our code.

The second URL pattern, path('about/', views.about, name='about'), maps the URL http://localhost:8000/about/ to the about view function. This demonstrates how you can create a sub-URL within your app, by appending a unique path segment to the root URL.

Let's take a closer look at how view functions work in Django.

View functions are Python functions that take an incoming request as their first parameter, and return an HTTP response. They are responsible for rendering HTML templates, interacting with models and databases, and processing user input.

Here's an example views.py file that defines the two view functions used in the urls.py file above:

pythonCopy code

from django.shortcuts import render from django.http import HttpResponse def index(request): return HttpResponse("Hello, world!") def about(request): return render(request, 'about.html')

In this code, we've imported two functions from the django.shortcuts and django.http modules: render and HttpResponse. We've then defined two view functions that correspond to the URL patterns in our urls.py file.

The index function simply returns an HTTP response containing the text "Hello, world!" This demonstrates the most basic type of view function, which doesn't use any HTML templates or interact with databases.

The about function, on the other hand, uses the render function to generate an HTTP response containing the contents of an HTML template. The render function takes two parameters: the incoming request, and the name of the template file to be rendered. In this case, we're rendering a template called about.html, which should be stored in our app's templates directory.

Step 5:

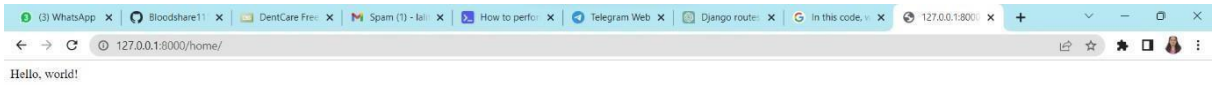
Save changes and runserver

>python manage.py run server

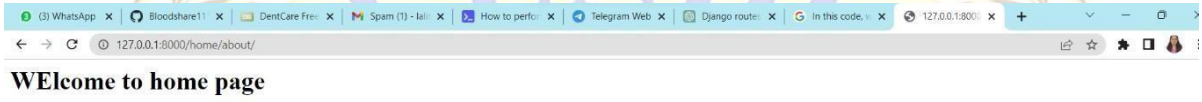
http://127.0.0.1:8000/

http://127.0.0.1:8000/home/

OUTPUT



http://127.0.0.1:8000/home/about/



3. AIM :Implement template inheritance with views and images.

DESCRIPTION:

To load images, we have to create static app .

Step 1: In emp we have to create new folder named static.

Step 2: in static folder we have to add images folder.

Step 3: In images folder we have to add a .png file.

Step 4: in settings.py we have to load static and images.

```
STATIC_URL = 'static/'
MEDIA_URL='/images/'
STATICFILES_DIRS=[
   os.path.join(BASE_DIR,'static')
]
```

Step 5: In home.html add code for image

```
<!DOCTYPE html>
{% load static %}
<html>
<head>
<title></title>
</head>
<body>
<img src="{% static '/images/django.png' %}"width="100" height="100">
</body>
</html>
```

Step 6: save changes

Step 7: runserver

VIVA QUESTIONS

1. What are Django templates?

2. What is Django Template Language (DTL)?

3. How do you pass data from views to templates in Django?

4. What are template tags in Django?

5. What are template filters in Django?.

6. What is the purpose of the `{% block %}` tag in templates?

7. What is template inheritance in Django?

8. How do you include other templates in a Django template?

9. What are static files in Django templates, and how do you use them?

10. What is the `{% extends %}` tag in Django templates?

MODULE-3

1) AIM : Create database configuration in Django admin with

DESCRIPTION:

sqlite3. Step 1: Create a new project and make all steps that are mentioned in previous modulesStep 2: Here I have taken app named charity

In that I want add donors data base – in models.py

```
from django.db import
models from django.utils
import timezoneclass
Donate(models.Model):
    donationType = models.CharField(max_length =200)
    donationAmount = models.FloatField(null=True)
    donationDate = models.DateTimeField(auto_now_add=timezone.now())
```

Step 3: in admin.py

```
from django.contrib
import adminfrom
.models import Donate
admin.site.register(Dona
te)
```

Step 4: in cmd we perform migrations

> python manage.py makemigrations

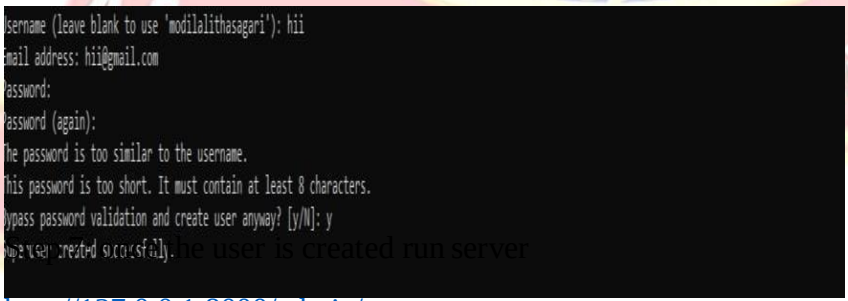
This will create a new directory called **migrations** inside your app directory with the initial migration files.

Step 5: > python manage.py migrate

Step 6: we have to create users for that we use command

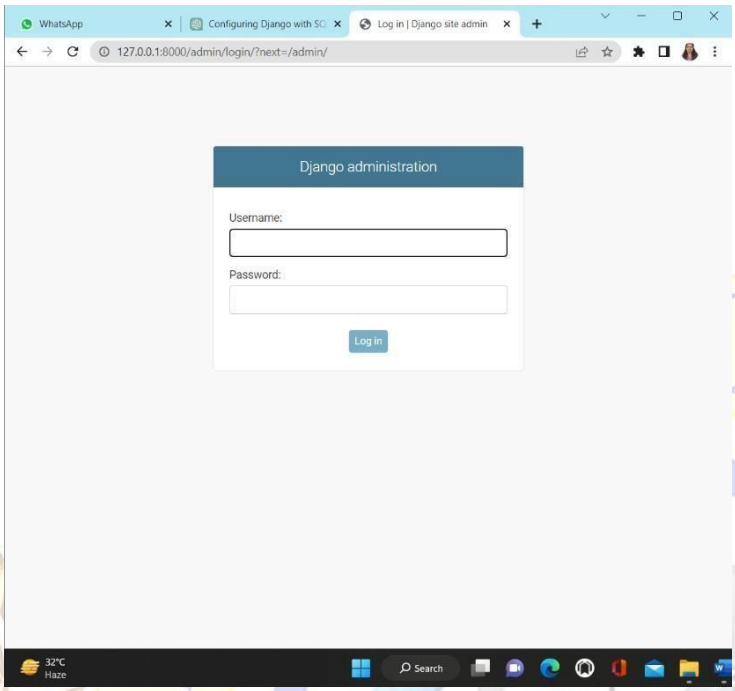
> python manage.py createsuperuser

Then it will ask details like



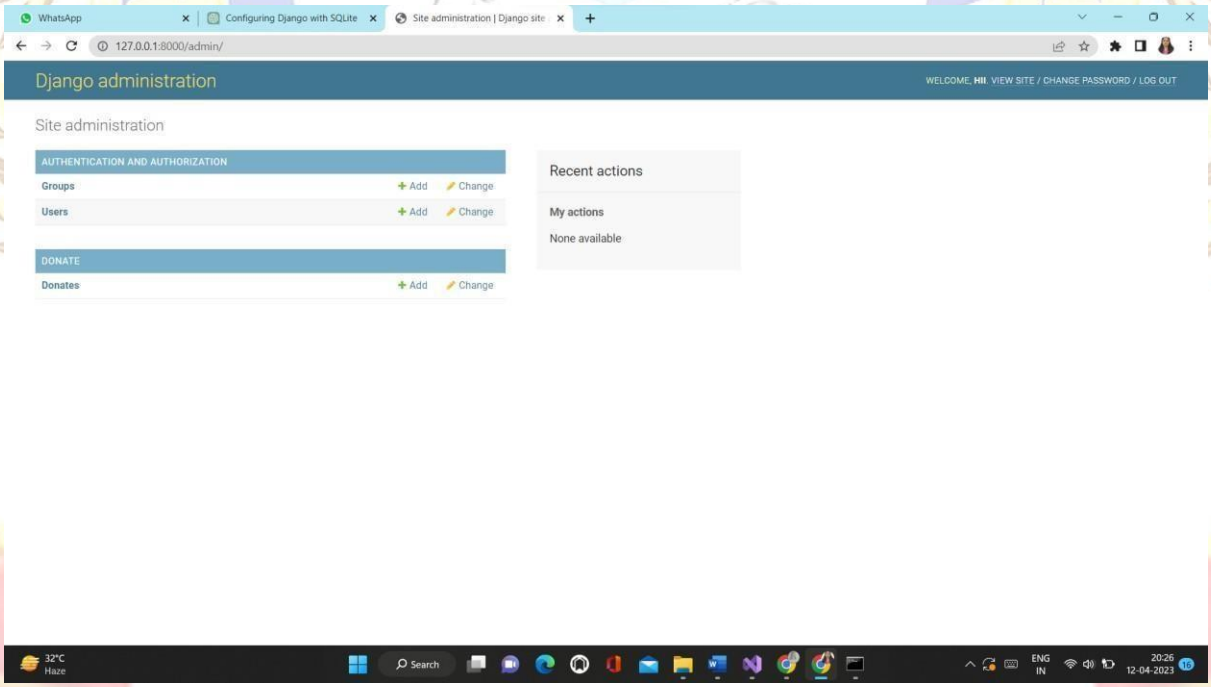
he user is created run server

http://127.0.0.1:8000/admin/



Login using user details.

Step 8: after logging in we are directed to site administartion



2) AIM :Implement CRUD operations using django models

DESCRIPTION :

Here's an example of how you can implement CRUD operations using Django models:

Let's say we have a model called Book that has the following fields:

```
from django.db import models


class Book(models.Model):
    title = models.CharField(max_length=255)
    author = models.CharField(max_length=255)
    publication_date = models.DateField()
price = models.DecimalField(max_digits=5, decimal_places=2)
```

To perform CRUD operations on this model, we can define views that correspond to each operation.

Create

To create a new Book instance, we can define a view that handles a POST request to a URL like /books/new/. The view should take the form data submitted with the POST request and use it to create a new Book instance:

```
from django.shortcuts import render, redirect
from .models import Book
from .forms import BookForm
def create_book(request):
    if request.method == 'POST':
        form = BookForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('book_list')
    else:
        form = BookForm()
    return render(request, 'book_form.html', {'form': form})
```

Here, we're using a Django ModelForm to generate an HTML form based on the Book model. If the form is submitted with valid data, we save the form data as a new Book instance and redirect the user to a page that lists all the books.

Read

To retrieve a list of all Book instances, we can define a view that handles a GET request to a URL like /books/. The view should query the database for all Book instances and pass them to a template for rendering:

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


def book_list(request):
    books = Book.objects.all()
    return render(request, 'book_list.html', {'books': books})
```

Here, we're using the Django ORM's all() method to retrieve all Book instances from the database. We're then passing the list of books to a template called book_list.html, which will render each book in a table row.

To retrieve a single Book instance, we can define a view that handles a GET request to a URL like /books/<int:pk>/. The pk parameter is the primary key of the Book instance we want to retrieve. We can query the database for the Book instance with this primary key and pass it to a template for rendering:

```
from django.shortcuts import render, get_object_or_404
from .models import Book


def book_detail(request, pk):
    book = get_object_or_404(Book, pk=pk)
    return render(request, 'book_detail.html', {'book': book})
```

Here, we're using the Django ORM's get_object_or_404() function to retrieve the Book instance with the given primary key. If the instance doesn't exist, the function will raise a Http404 exception. We're then passing the Book instance to a template called book_detail.html, which will render the book's details.

Update

To update an existing Book instance, we can define a view that handles a POST request to a URL like /books/<int:pk>/edit/. The view should retrieve the Book instance with the given primary key, update its fields with the form data submitted with the POST request, and save the updated instance to the database:

```
from django.shortcuts import render, redirect, get_object_or_404
from .models import Book
from .forms import BookForm
def update_book(request, pk):
    book = get_object_or_404(Book, pk=pk)
    if request.method == 'POST':
        form = BookForm(request.POST, instance=book)
        if form.is_valid():
            form.save()
            return redirect('book_list')
    else:
        form = BookForm(instance=book)
    return render(request, 'book_form.html', {'form': form})
```

Here, we're using a Django `ModelForm` to generate an HTML form based on the `Book` instance we want to update. If the form is submitted with valid data, we save the updated form data to the existing `Book` instance and redirect the user to a page that lists all the books.

Delete

To delete an existing `Book` instance, we can define a view that handles a POST request to a URL like `/books/<int:pk>/delete/`. The view should retrieve the `Book` instance with the given primary key and delete it from the database:

```
from django.shortcuts import render, redirect, get_object_or_404
from .models import Book


def delete_book(request, pk):
    book = get_object_or_404(Book, pk=pk)
    if request.method == 'POST':
        book.delete()
        return redirect('book_list')
    return render(request, 'book_confirm_delete.html', {'book': book})
```

Here, we're using the `delete()` method on the `Book` instance to delete it from the database. If the request method is POST (i.e. the user has confirmed the deletion), we delete the instance and redirect the user to a page that lists all the books. If the request method is GET, we render a confirmation page that asks the user to confirm the deletion.

3) Implement database relations using Django models

you can implement database relations using Django models with different types of relationships:

One-to-Many Relationship

A one-to-many relationship is a relationship in which a single instance of one model is related to multiple instances of another model. In Django, this is represented by a ForeignKey field on the many-side model.
Let's say we have a Book model and an Author model, and each book is written by a single author:

```
from django.db import models


class Author(models.Model):
    name = models.CharField(max_length=255)
class Book(models.Model):
    title = models.CharField(max_length=255)
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
```

Here, the Book model has a ForeignKey field called author that refers to an instance of the Author model. The on_delete argument specifies what should happen to the Book instances if the related Author instance is deleted. In this case, we're using models.CASCADE, which means that if an Author instance is deleted, all related Book instances will also be deleted.

Many-to-Many Relationship
A many-to-many relationship is a relationship in which each instance of one model can be related to multiple instances of another model, and vice versa. In Django, this is represented by a ManyToManyField on both models.

```
from django.db import models
class Genre(models.Model):
    name = models.CharField(max_length=255)
class Book(models.Model):
    title = models.CharField(max_length=255)
    genres = models.ManyToManyField(Genre)
```

Let's say we have a Book model and a Genre model, and each book can belong to multiple genres

Here, the Book model has a ManyToManyField called genres that refers to instances of the Genre model. The genres field creates a many-to-many relationship between Book and Genre instances.

One-to-One Relationship
A one-to-one relationship is a relationship in which each instance of one model is related to exactly one instance of another model, and vice versa. In Django, this is represented by a OneToOneField on one of the models.
Let's say we have a Book model and a Publisher model, and each book is published by a single publisher:

```
from django.db import models


class Publisher(models.Model):
    name = models.CharField(max_length=255)
class Book(models.Model):
    title = models.CharField(max_length=255)
    publisher = models.OneToOneField(Publisher, on_delete=models.CASCADE)
```

Here, the Book model has a OneToOneField called publisher that refers to an instance of the Publisher model. The publisher field creates a one-to-one relationship between Book and Publisher instances.

4) AIM :Implement data rendering with templates and data routing.

DESCRIPTION : Here's an example of how to render data using Django templates and routing:

Routing
To route URLs in Django, you can define URL patterns in a urls.py file. Here's an example urls.py file for a simple blog app:

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

urlpatterns = [
    path('', views.post_list, name='post_list'),
    path('post/<int:pk>/', views.post_detail, name='post_detail'),
]
```

Here, we're defining two URL patterns: one for the list of blog posts (''), and one for the detail view of a single post ('post/<int:pk>/'). The int:pk part of the second pattern means that Django will match an integer and assign it to the pk argument of the view function.

Rendering Data with Templates
To render data in a Django template, you can use template tags and filters. Here's an example post_list.html template that renders a list of blog posts:

```
{% extends 'base.html' %}
{% block content %}
    <h1>Blog Posts</h1>
    {% for post in posts %}
        <h2><a href="{% url 'post_detail' pk=post.pk %}">{{ post.title }}</a></h2>
        <p>{{ post.content }}</p>
    {% empty %}
        <p>No posts yet.</p>
    {% endfor %}
{% endblock %}
```

Here, we're using the extends template tag to extend a base template, and the block template tag to define a content block that will be filled in by the child template. We're using the for template tag to loop over a list of Post objects, and the url template tag to generate a URL for the detail view of each post. Finally, we're using the empty template tag to display a message if there are no posts.

Here's an example post_detail.html template that renders the detail view for a single post:

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


{% block content %}
    <h1>{{ post.title }}</h1>
    <p>{{ post.content }}</p>
{% endblock %}
```

Here, we're again using the extends template tag to extend a base template, and the block template tag to define a content block. We're using the {{ }} syntax to display the title and content of the Post object passed to the template context.

Views.py
Finally, we need to define the views that will render these templates and provide data to them. Here's an example views.py file:

```
from django.shortcuts import render, get_object_or_404
from .models import Post


def post_list(request):
    posts = Post.objects.all()
    return render(request, 'post_list.html', {'posts': posts})
def post_detail(request, pk):
    post = get_object_or_404(Post, pk=pk)
    return render(request, 'post_detail.html', {'post': post})
```

VIVA QUESTIONS

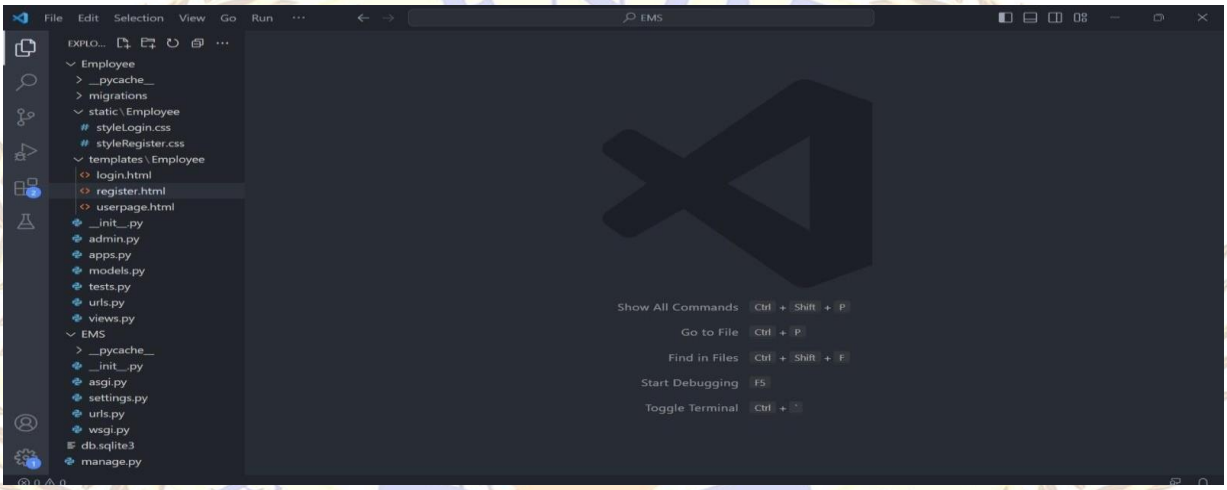
1. **What is the role of Django models in web development?**
2. **Explain the concept of Django ORM and how it relates to models.**
3. **What are the different types of fields available in Django models, and how do they work?**
4. **What is a ForeignKey in Django models, and how is it used to define relationships?**
5. **What is the significance of the `save()` method in Django models?**

MODLE-4

1. AIM :Create user registration and login authentication using Django forms

DESCRIPTION

- The project Employee management system(EMS) has basically two important pages
  - Registration page
  - Login page
- The project directory structure looks like



The important files here are

1. Urls.py →EMS

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

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

2. Urls.py →Employee app

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

```
from . import views

urlpatterns = [
    path('register/',views.Register),
    path('register/user/',views.User),
    path('login/',views.Login),
    path('login/validateuser/',views.validate),
]
```

3. Views.py → Employee app

```
from django.shortcuts import render,redirect
from Employee.models import Employee
from django.contrib import messages

# Create your views here.
def Register(request):
    return render(request,'Employee/register.html')

def Login(request):
    return render(request,'Employee/login.html')

def User(request):
    if request.method == 'POST':
        eid=request.POST['eid']
        email=request.POST['mail']
        ename=request.POST['ename']
        id=Employee.objects.all().count()+1
        flag=Employee.objects.filter(eid=eid).count()
        if(flag!=1):
            Employee(id,eid,email,ename).save()
            return render(request,'Employee/userpage.html',{'name':ename})
        else:
            messages.error(request, 'User already exists')
            return redirect('/register')

def validate(request):
    if request.method == 'POST':
        eid=request.POST['eid']
        email=request.POST['mail']
        flag=Employee.objects.filter(eid=eid,email=email).count()
        if(flag==1):
            row=Employee.objects.filter(eid=eid,email=email)
            for i in row:
                name=i.ename
            return render(request,'Employee/userpage.html',{'name':name})
```

```
        else:
            messages.error(request, 'Invalid id or mail')
            return redirect('/login')
```

4. register.html → templates→Employee

```
{% load static %}
<!DOCTYPE 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>register</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
alpha1/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-
GLhlTQ8iRABdZLl6O3oVMWSktQOp6b7In1Zl3/Jr59b6EGGoI1aFkw7cmDA6j6gD"
crossorigin="anonymous">
    <link rel="stylesheet" href="{% static 'Employee/styleRegister.css' %}">
</head>
<body>
    <div class="container-fluid" id="signup">
        <form class="p-5" action="user/" method="post">
          {% csrf_token %}
          <div class="mb-3">
            <label for="eid" class="form-label">Employee Id</label>
            <input type="text" placeholder="your id" class="form-control" id="eid" name="eid"
required>
          </div>
          <div class="mb-3">
            <label for="mail" class="form-label">Email address</label>
            <input type="email" placeholder="your email" class="form-control" id="mail"
name="mail" required>
          </div>
          <div class="mb-3">
            <label for="ename" class="form-label">Name</label>
            <input type="text" placeholder="name" class="form-control" id="ename"
name="ename" required>
          </div>
          <button type="submit" class="btn btn-secondary">Register</button>
          <div class="mb-3">
```

```
          <label class="p-3">Are you already a user <a href="/login" class="mx-
2">login</a></label>
         </div>
         <div class="mb-3">
           <p style="color:firebrick;text-align: center; font-weight: bold; font-
family:cursive;">{%  for message in messages %}
             {{ message }}
             {% endfor %}</p>
       </div>
       </form>
     </div>

     <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
alpha1/dist/js/bootstrap.bundle.min.js" integrity="sha384-
w76AqPfDkMBDXo30jS1Sgez6pr3x5MlQ1ZAGC+nuZB+EYdgRZgiwxhTBTkF7CXvN"
crossorigin="anonymous"></script>
</body>
</html>
```

5. styleRegister.css → static →Employee

```
#signup{
   margin-top: 50px;
}
#signup  form{
   width: 35%;
   margin-left: auto;
   margin-right: auto;
   background-color: aliceblue;
   box-shadow: 2px 3px 3px 1px black;
   border-radius: 10px;
}
#signup form label{
   font-size: medium;
   font-family: 'Times New Roman', Times, serif;
}
```

6. login.html → templates→Employee

```
{% load static %}
<!DOCTYPE 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>login</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
alpha1/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-
GLhlTQ8iRABdZLl6O3oVMWSktQOp6b7In1Zl3/Jr59b6EGGoI1aFkw7cmDA6j6gD"
crossorigin="anonymous">
    <link rel="stylesheet" href="{% static 'Employee/styleLogin.css' %}">
</head>
<body>
    <div class="container-fluid py-5" id="login">
        <p>Login here</p>
        <form action="validateuser/" method="post" class="p-5">
          {% csrf_token %}
            <div class="mb-3">
              <label for="eid" class="form-label">Employee id</label>
              <input type="text" placeholder="Id" class="form-control" id="eid" name="eid"
required>
            </div>
            <div class="mb-3">
              <label for="mail" class="form-label">Email</label>
              <input type="email" placeholder="mail" class="form-control" id="mail"
name="mail" required>
            </div>
            <button type="submit" class="btn btn-secondary">Login</button>
            <div class="mb-3">
              <label class="p-3">Are you a new user <a href="/register" class="mx-
2">signup</a></label>
            </div>
            <div class="mb-3">
              <p style="color:firebrick;text-align: center; font-weight: bold; font-
family:cursive;">{% for message in messages %}
                {{ message }}
                {% endfor %}</p>
          </div>
        </form>
    </div>

    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
alpha1/dist/js/bootstrap.bundle.min.js" integrity="sha384-
w76AqPfDkMBDXo30jS1Sgez6pr3x5MlQ1ZAGC+nuZB+EYdgRZgiwxhTBTkF7CXvN"
crossorigin="anonymous"></script>
</body>
</html>
```

7. styleLogin.css→ static→ Employee

```
#login{
    margin-top: 50px;
}
#login form{
    width: 35%;
    margin-left: auto;
    margin-right: auto;
    background-color: aliceblue;
    box-shadow: 2px 3px 3px 1px black;
    border-radius: 10px;
}
#login form label{
    font-size: medium;
    font-family: 'Times New Roman', Times, serif;
}

#login p{
    font-size: 17px;
    font-family:Arial, Helvetica, sans-serif;
    text-align: center;
}
```

8. Models.py

```
from django.db import models

# Create your models here.
class Employee(models.Model):
    eid=models.CharField(max_length=20)
    email=models.CharField(max_length=50)
    ename=models.CharField(max_length=20)
```

9. admin.py

```
from django.contrib import admin
from .models import Employee
# Register your models here.
admin.site.register(Employee)
```

10. userpage.html

```
```

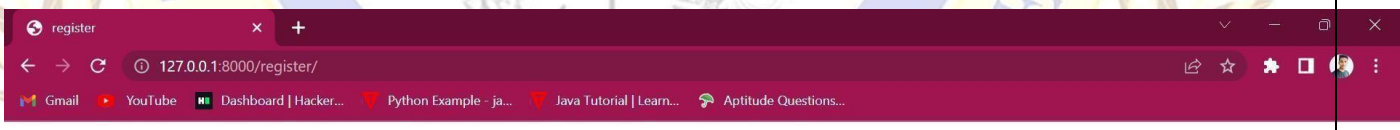
```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Document</title>
</head>
<body>
   <div>
      <h1>hello {{ name }}</h1>
   </div>
</body>
</html>
```

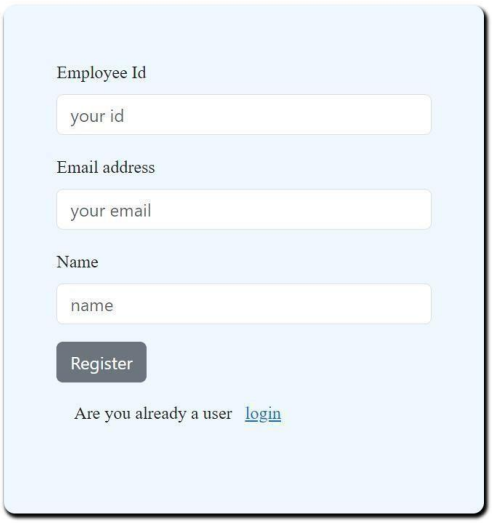
D:\django\Ems>python manage.py runserver

Copy http://127.0.0.1:8000/

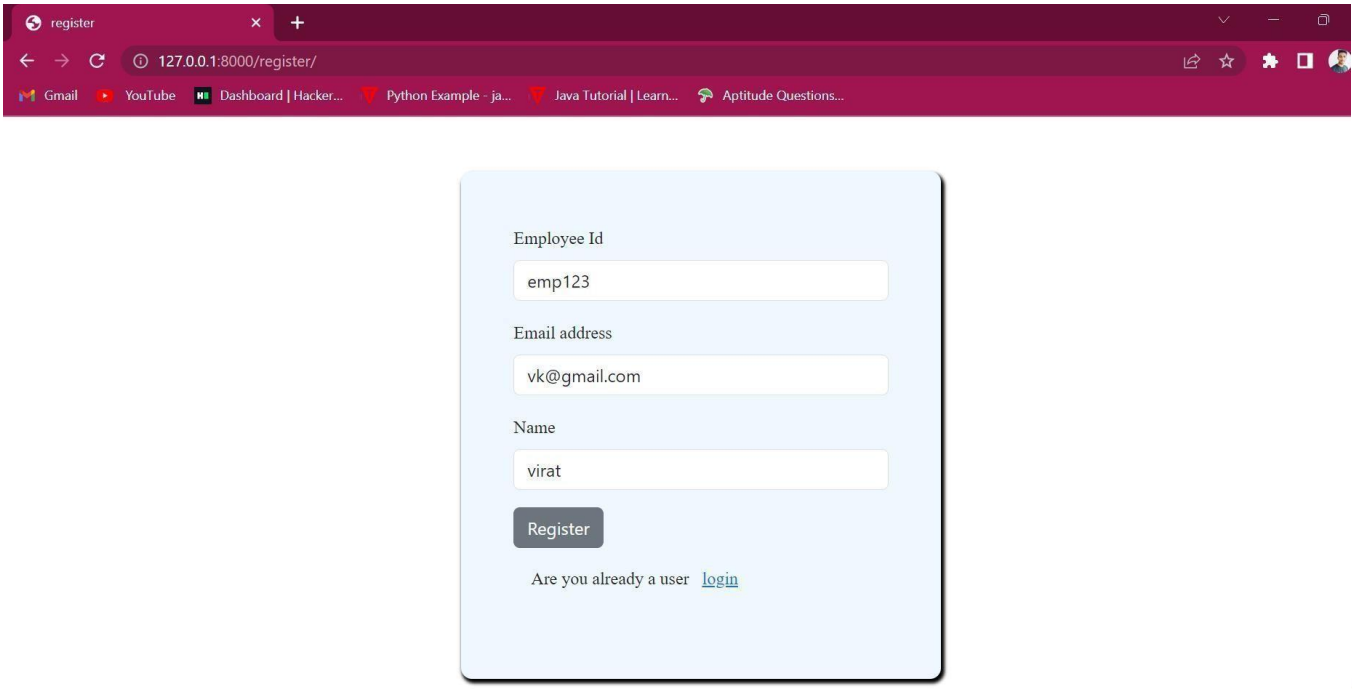
Goto your browser and paste this url in the address bar http://127.0.0.1:8000/register/ y

OUTPUT

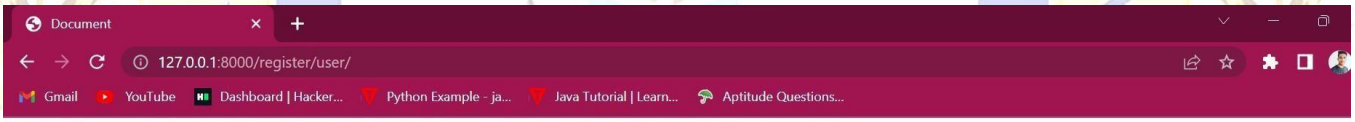




When u try to register with your details



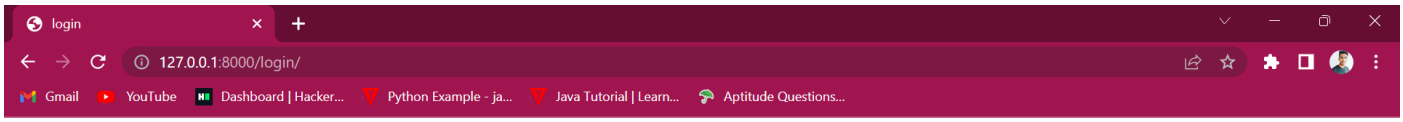
You get the output as follows

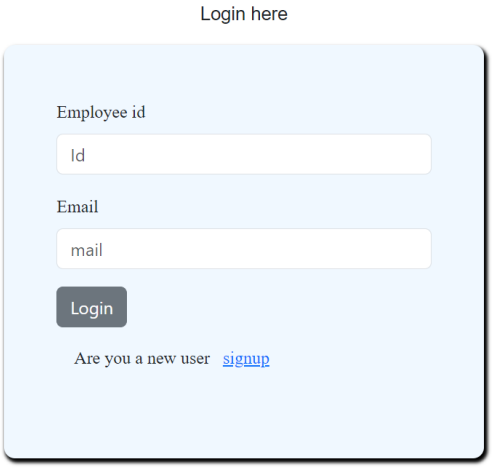


hello virat

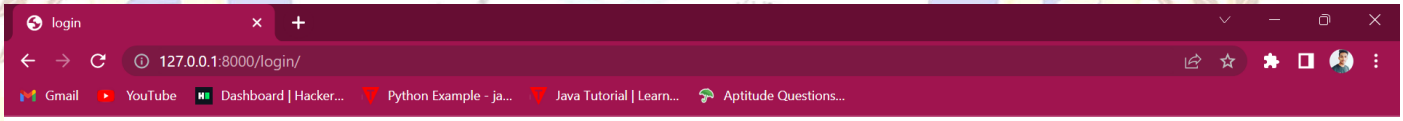
Similarly if you paste this url in the address bar http://127.0.0.1:8000/login/

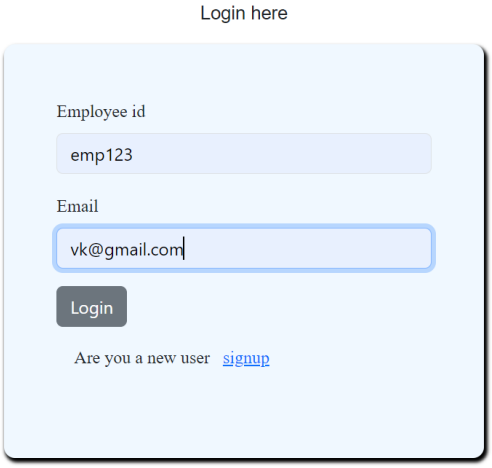
you get output as follows



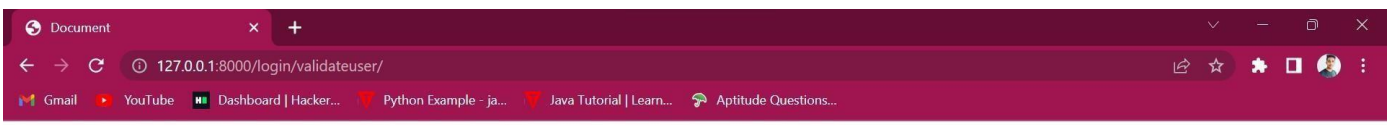


If you login with your credentials





You get this page



VIVA QUESTIONS

1. What are dynamic forms in Django?

2. How do you create a dynamic form in Django?

3. What is a Django signal?

4. How do you connect a signal to a model in Django?

5. What are the common use cases for Django signals?

MODULE-5

DJANGO REST FRAMEWORK

To work with the Django rest framework we should have the two basic requirements.

Python (3.6, 3.7, 3.8, 3.9, 3.10)

Django (2.2, 3.0, 3.1, 3.2, 4.0, 4.1)

1. AIM :Install and configure Django Rest Framework Package.

DESCRIPTION :

Step1: navigate to your project folder from the command prompt

Step2: Install Django rest framework using pip.

```
D:\django\EMS> pip install djangorestframework
```

Step3: Add 'rest_framework' to your INSTALLED_APPS in your settings.py file.

```
INSTALLED_APPS = [
    ...
    'rest_framework',
]
```

Now the Django rest framework is ready and available and you can use it in your project to create the Web APIs.

2. Create Django rest end points using api_view and JSON Response.

- To create the rest end points you have to return a JSON response from the requested url

Urls.py

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

urlpatterns = [
  path('',views.my_view),
```

```
]
```

```
from rest_framework.response import Response
from rest_framework.decorators import api_view

@api_view(['POST','GET','DELETE'])
def my_view(request):
    return Response({'status':'success','message':'request is success','details':'you can give any request("get","put","delete")'},status=200)
```

- In views.py we have used a decorator api_view and a Response class from rest_framework.

@api_vie is a decorator provided by the Django REST Framework (DRF) that is used to specify that a function-based view or method-based view should be treated as a web API view.

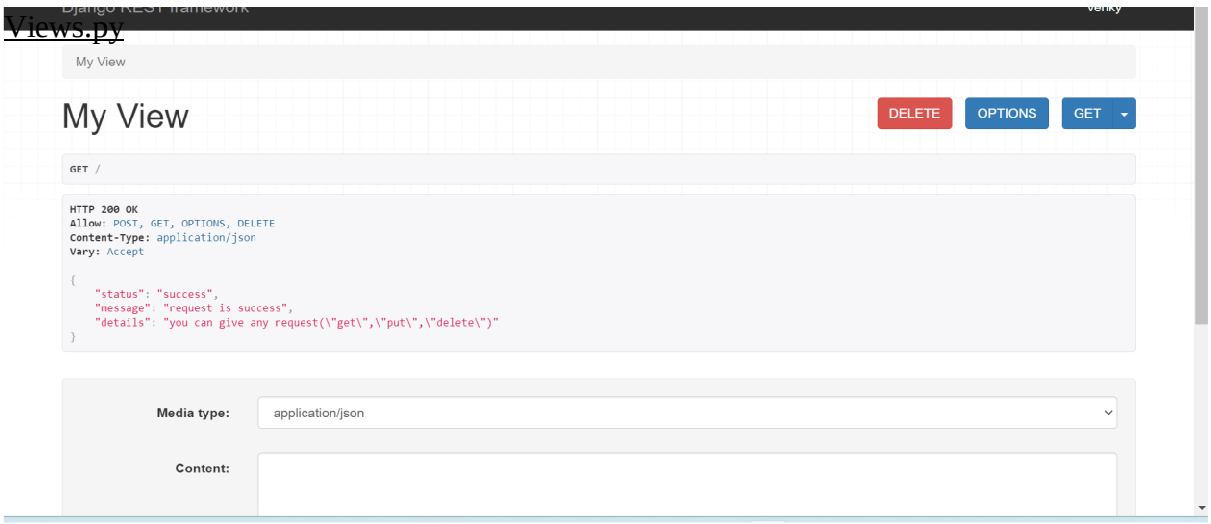
@api_vie takes an HTTP method or a list of HTTP methods as an argument. For example, to specify that a view should handle only POST requests, you would use the @api_view(['POST']) decorator.

The response class is used to return the JSON response to the client.

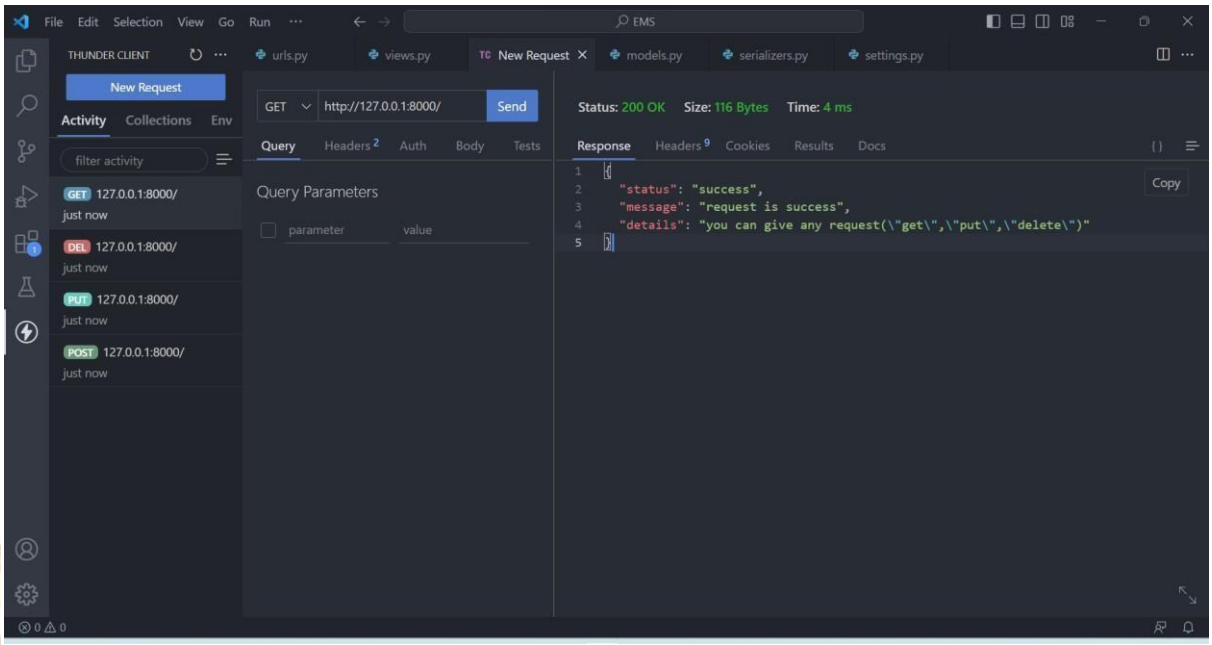
OUTPUT

- Now if you run the server and paste the url in your browser you get the output as

Views.py



- You can also give the request to that particular url from the editor that you are working.
- Here in this case I am using VScode as my editor.
- Download the Thunder client extension and you can give a request fron the editor itself.
- The below picture is the different requests given from the thunder client to our url.



3. Create Django rest end points using Serializers and Models.

- In Django, a serializer is a component of the Django REST framework (DRF) that is used to convert complex data types, such as Django model instances, into Python native data types that can be easily rendered into JSON, XML.
- Inorder to create a serializer you have to create a serializers.py file inside your app employee

Urls.py

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

urlpatterns = [
    path('employees',views.Employees),
]
```

Serializers.py

```
from rest_framework import serializers
from .models import *
class EmployeeSerializer(serializers.ModelSerializer):
    class Meta:
        model = Employee
        fields='__all__'
```

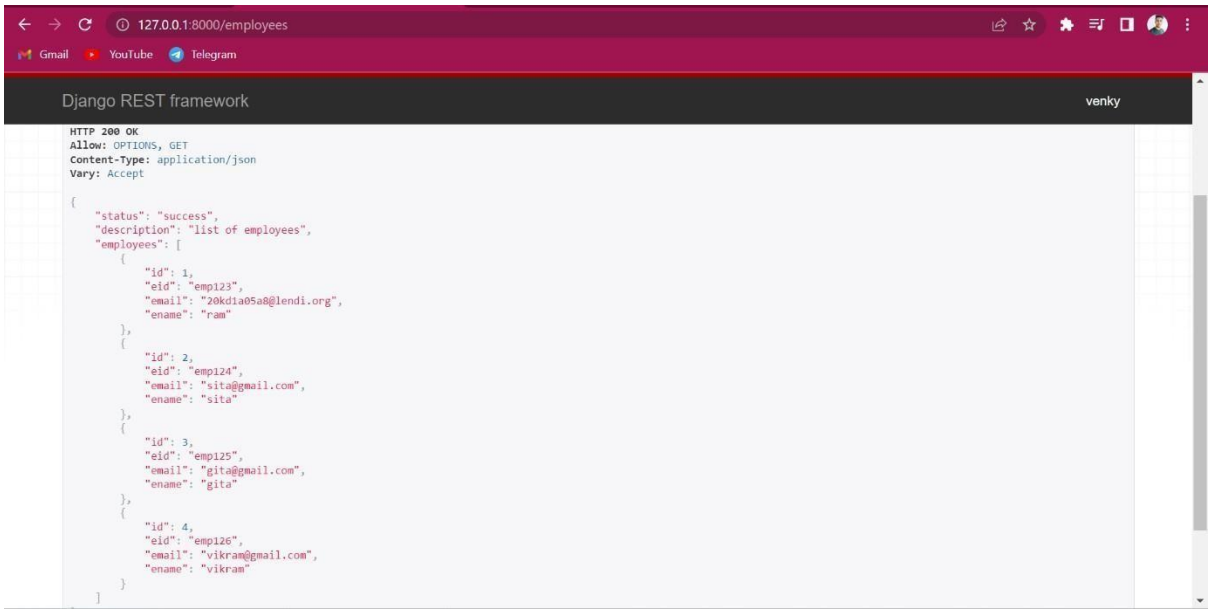
views.py

```
from rest_framework.response import Response
from rest_framework.decorators import api_view
from .models import *
from .serializers import *

@api_view(['GET'])
def Employees(request):
    emps=Employee.objects.all()
    serializer=EmployeeSerializer(emps,many=True)
    return Response({'status':'success','description':'list of
employees','employees':serializer.data},status=200)
```

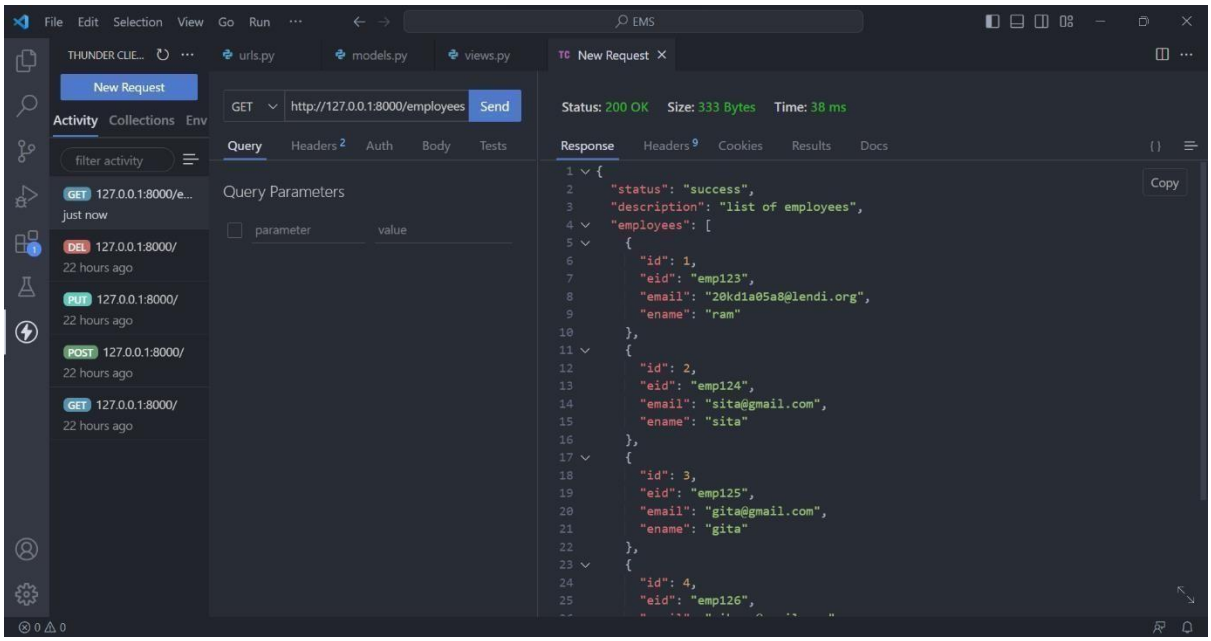
models.py

```
from django.db import models

# Create your models here.
class Employee(models.Model):
    eid=models.CharField(max_length=20)
    email=models.CharField(max_length=50)
    ename=models.CharField(max_length=20)
```

Run the server and paste the url in the browser and you can see the output as



And you can see the output from the VScode editor as follows.



4. Implement HTTP rest end points with all CRUD operations.

CRUD operations refers to the operations done on the database, where

- C- create -to create a new record.
- R- retrive-retriving the records from database.
- U- update-to update the records in the database.
- D- delete the records in the database.

To implement the HTTP rest end points with all CRUD operations using DRF

we should use the api_view decorators inorder to specify the HTTP method.

The following are the endpoints which are responsible for different CRUD operations

Urls.py

```
urlpatterns = [
    path('all',views.All),
    path('new',views.New),
    path('update/<str:eid>/',views.Update),
    path('delete/<str:eid>/',views.Delete),
]
```

The following are the views that handles the different CRUD operations

Views.py

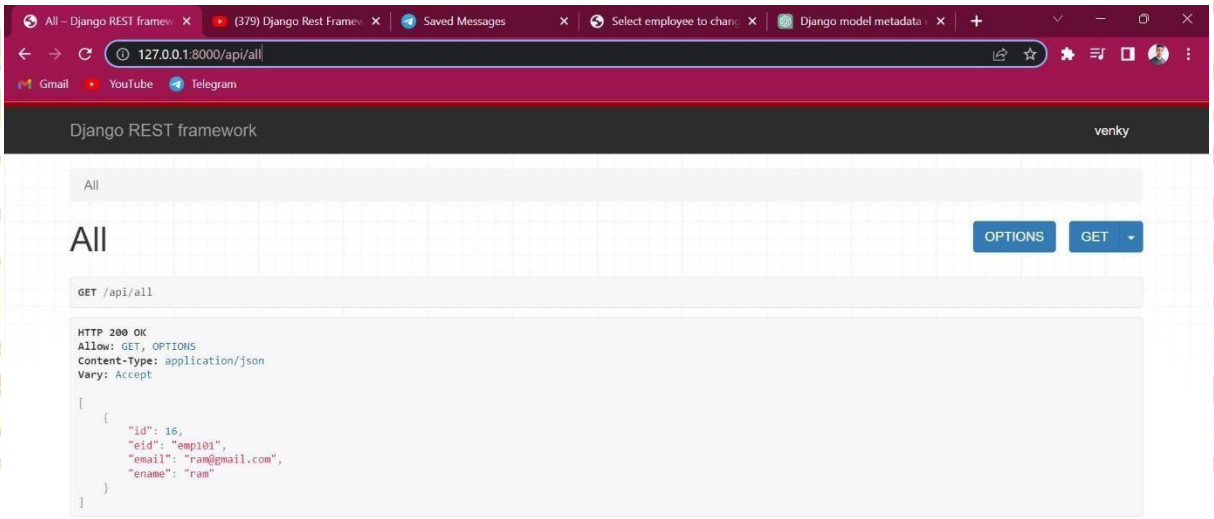
```
from rest_framework.response import Response
from rest_framework.decorators import api_view
from .models import *
from .serializers import *


#list of employees
@api_view(['GET'])
def All(request):
    emps=Employee.objects.all()
    serializer=EmployeeSerializer(emps,many=True)
    return Response(serializer.data,status=200)


#add a new employee
@api_view(['POST'])
def New(request):
    serializer=EmployeeSerializer(data=request.data)
    if serializer.is_valid():
        serializer.save()
    return Response(serializer.data,status=200)

#update a employee by using emp id
@api_view(['PUT'])
def Update(request,eid):
    emp=Employee.objects.get(eid=eid)
    serializer=EmployeeSerializer(instance=emp,data=request.data)
    if serializer.is_valid():
        serializer.save()
    return Response(serializer.data,status=200)
```

```python
#delete a employee by using emp id
@api_view(['DELETE'])
def Delete(request,eid):
    emp=Employee.objects.get(eid=eid)
    emp.delete()
    return Response("employee is deleted",status=200)
```

so the each endpoint is handling the different CRUD operations as follows

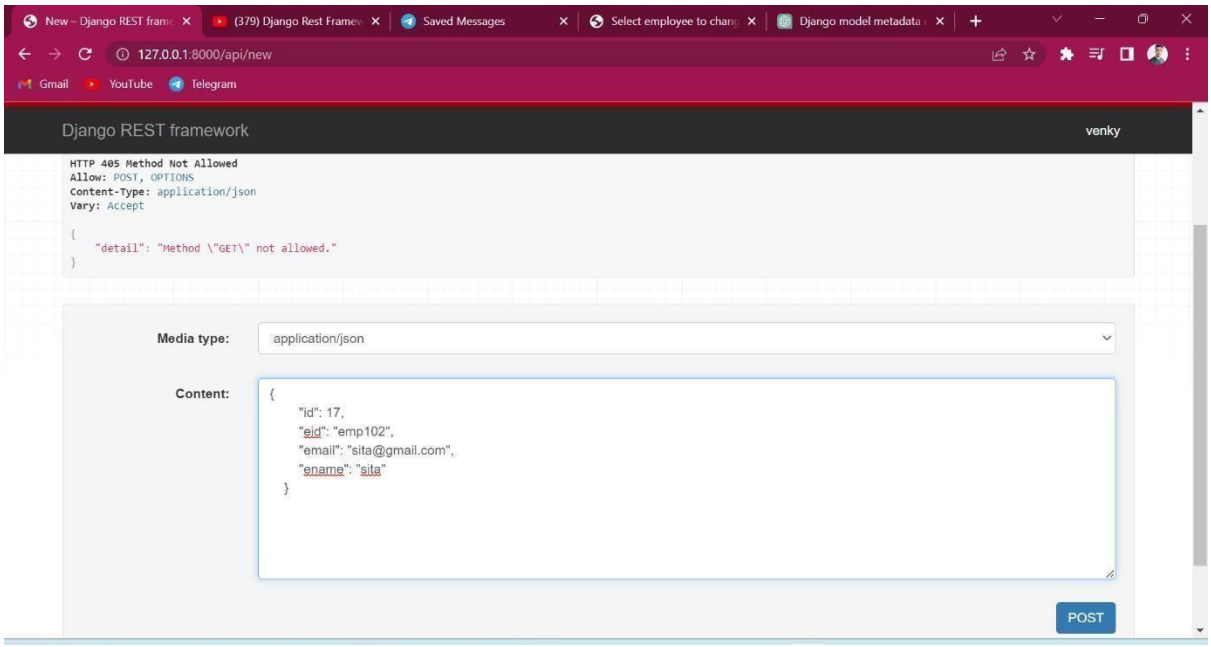
Retriving all the employees (GET):

run the server and paste the url in the browser **http://127.0.0.1:8000/api/all**



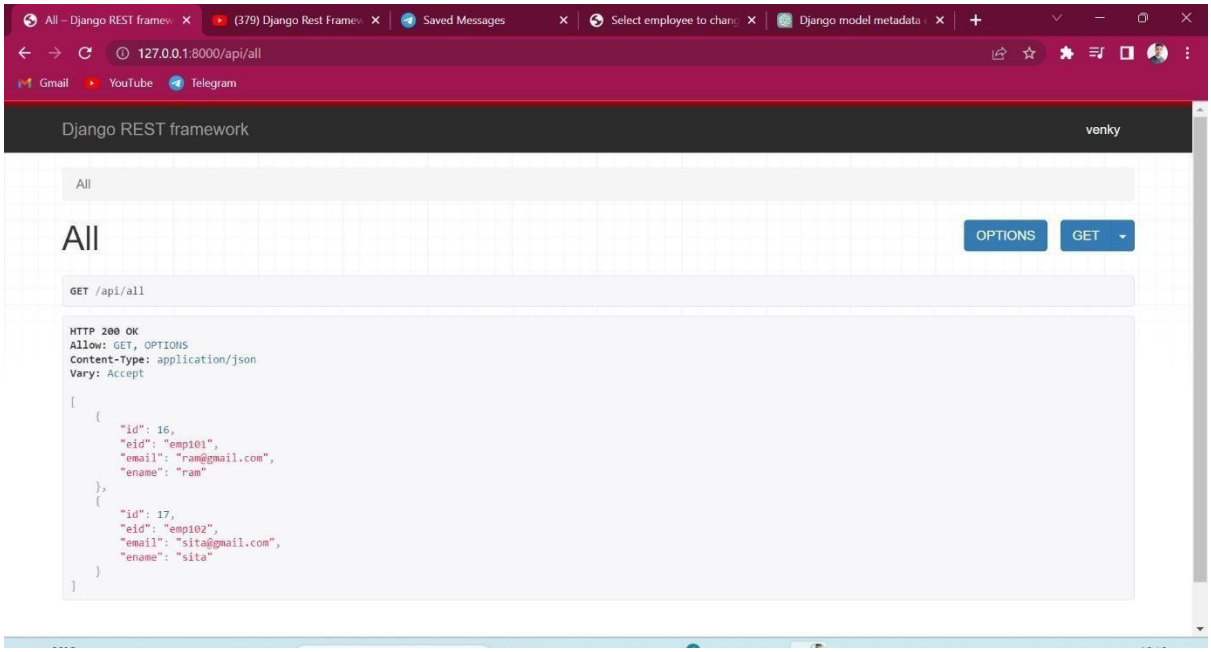
Adding a new employee (POST):

run the server and paste the url in the browser **http://127.0.0.1:8000/api/new**



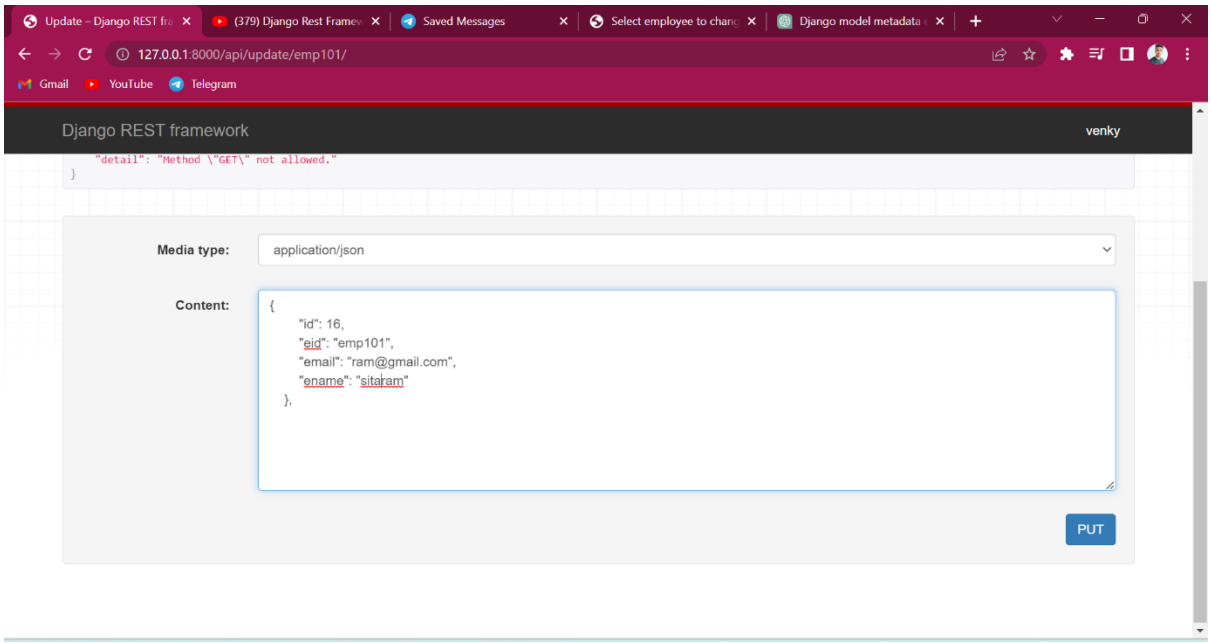
After clicking on post the record is added to the employee table

If you again see the total no. of employees present , you can observe the difference.



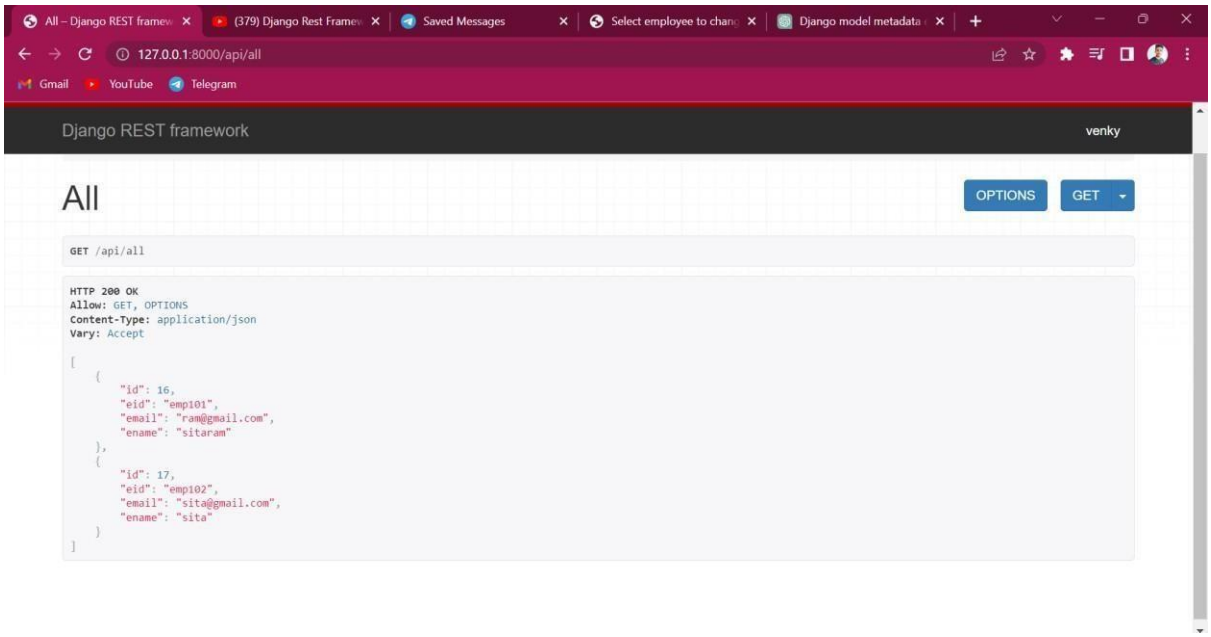
Updating an employee using empid (PUT):

run the server and paste url **http://127.0.0.1:8000/api/update/emp101/**



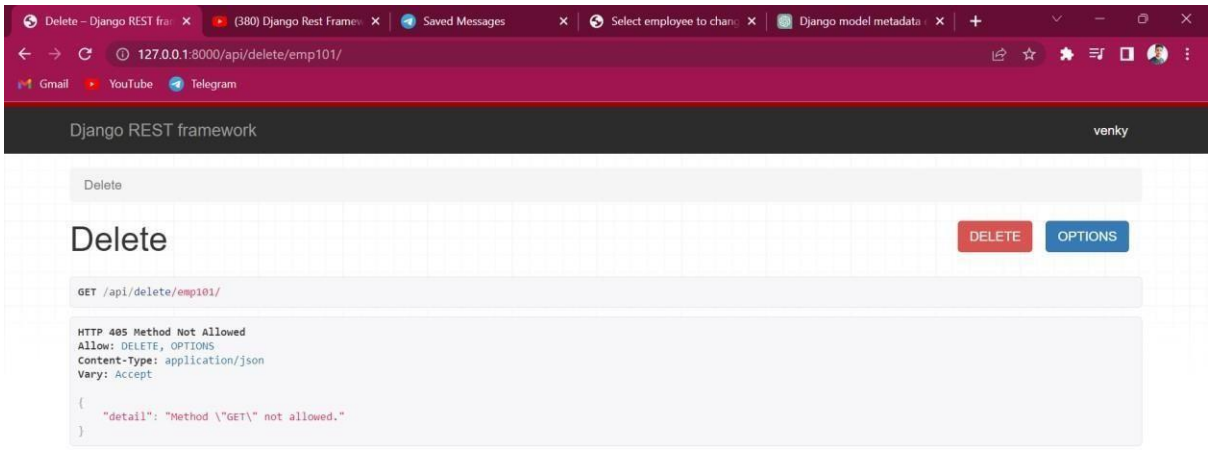
After clicking on put, the record with empid emp101 is updated i.e the name of the employee from ram to sitaram

If you again see the total no. of employees present , you can observe the difference.



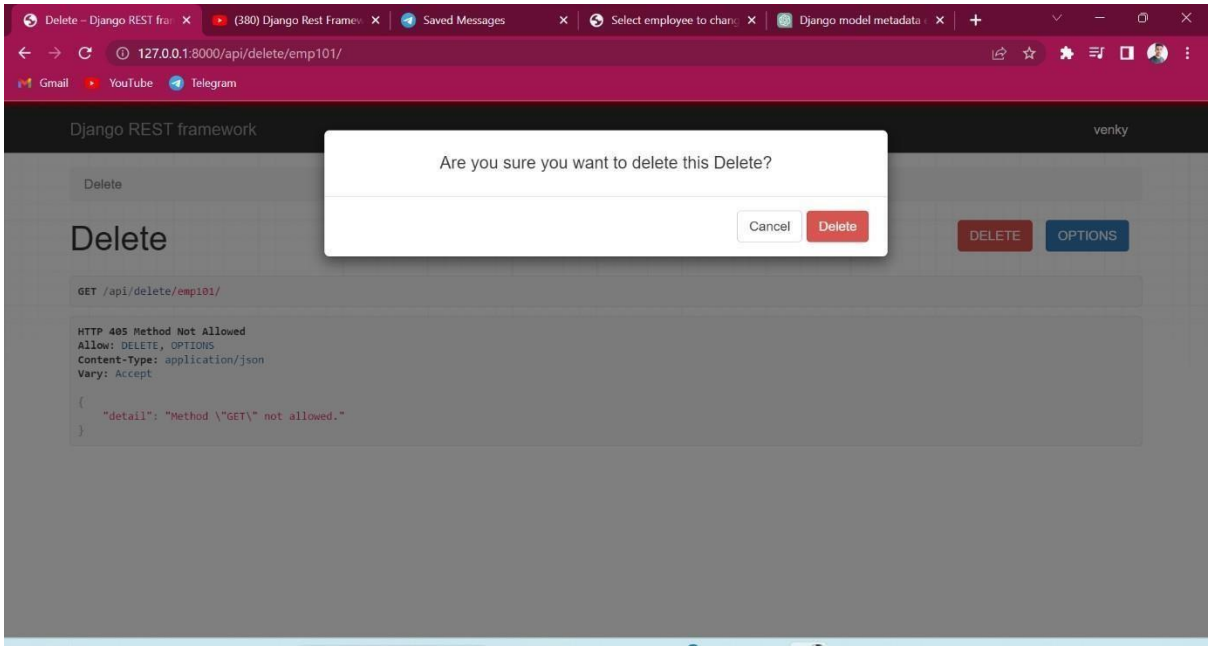
Deleting an employee using empid (DELETE):

run the server and paste url **http://127.0.0.1:8000/api/delete/emp101/**

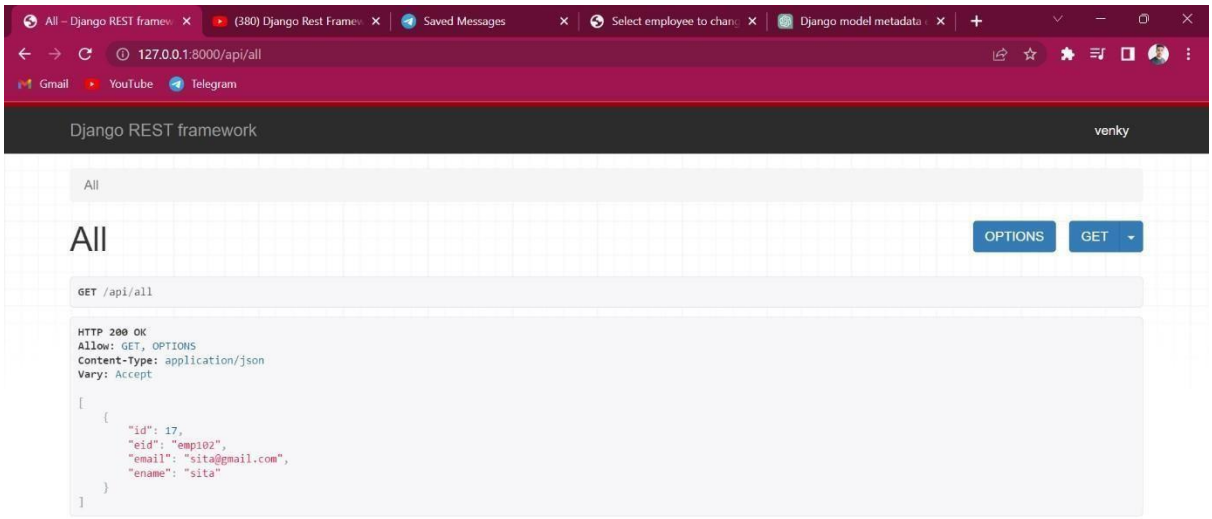


After clicking on delete, the record with empid emp101 is going to delete and there is prompt is there for us

If you click on delete again the record with the empid value as emp101 is deleted. If you again see the total no. of employees present , you can observe the difference.



Click **DELETE**



VIVA QUESTIONS

1. **What is Django Rest Framework (DRF)?**
2. **What is a serializer in Django Rest Framework?**
3. **What is the difference between `APIView` and `ViewSet` in DRF?**
4. **How does Django Rest Framework handle authentication?**
5. **What is a `router` in Django Rest Framework?**

14. AIM :Real-Time Collaborative Application with Django and WebSockets

DESCRIPTION :

Steps to Build a Real-Time Collaborative Application:

1. Setting up Django Project

First, create a Django project if you haven't already.

```
bash
Copy code
django-admin startproject realtime_app
cd realtime_app
python manage.py startapp chat
```

2. Install Django Channels

Django Channels is an extension for Django that adds support for WebSockets and other asynchronous protocols. To install Channels, run:

```
bash
Copy code
pip install channels
```

After installing Channels, you need to configure it in your `settings.py`.

3. Configure Channels in `settings.py`

In your `settings.py`, add `'channels'` to your `INSTALLED_APPS`:

```
python
Copy code
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'chat',  # your app
    'channels',  # channels app
]
```

Set `ASGI_APPLICATION` to point to your routing configuration:

```
python
Copy code
ASGI_APPLICATION = 'realtime_app.asgi.application'
```

Then, configure the database to use for Channels (e.g., Redis) for handling WebSocket connections. Redis is typically used as a backend to handle the channel layer, which makes it easy to handle multiple connections.

Install Redis and Channels Redis:

```
bash
Copy code
pip install channels_redis
```

In `settings.py`, add the following:

```
python
Copy code
CHANNEL_LAYERS = {
    'default': {
        'BACKEND': 'channels_redis.core.RedisChannelLayer',
        'CONFIG': {
            "hosts": [('127.0.0.1', 6379)],
        },
    },
}
```

Make sure you have Redis installed and running on your local machine or use a hosted Redis solution.

4. Create an ASGI Configuration

Create an `asgi.py` file in your project root (next to `settings.py`) if it doesn't already exist.

```
python
Copy code
# realtime_app/asgi.py
import os
from django.core.asgi import get_asgi_application
from channels.routing import ProtocolTypeRouter, URLRouter
from channels.auth import AuthMiddlewareStack
from chat.consumers import ChatConsumer

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

application = ProtocolTypeRouter({
    "http": get_asgi_application(),
    "websocket": AuthMiddlewareStack(
        URLRouter([
            # Add URL routing for WebSocket connections
            # Example for chat
            path('ws/chat/<str:room_name>/', ChatConsumer.as_asgi()),
        ])
    ),
})
```

5. Create a WebSocket Consumer

In Django Channels, a "consumer" is similar to a Django view but for handling WebSocket connections. Create a `consumers.py` file inside your app (e.g., `chat/consumers.py`):

```
python
Copy code
# chat/consumers.py
import json
from channels.generic.websocket import AsyncWebsocketConsumer

class ChatConsumer(AsyncWebsocketConsumer):
    async def connect(self):
        self.room_name = self.scope['url_route']['kwargs']['room_name']
```

```
        self.room_group_name = f'chat_{self.room_name}'

        # Join room group
        await self.channel_layer.group_add(
            self.room_group_name,
            self.channel_name
        )

        await self.accept()


async def disconnect(self, close_code):
        # Leave room group
        await self.channel_layer.group_discard(
            self.room_group_name,
            self.channel_name
        )

    # Receive message from WebSocket
    async def receive(self, text_data):
        text_data_json = json.loads(text_data)
        message = text_data_json['message']

        # Send message to room group
        await self.channel_layer.group_send(
            self.room_group_name,
            {
                'type': 'chat_message',
                'message': message
            }
        )

    # Receive message from room group
    async def chat_message(self, event):
        message = event['message']

        # Send message to WebSocket
        await self.send(text_data=json.dumps({
            'message': message
        }))
```

6. Routing WebSocket Requests

In the `asgi.py` file, you already defined the routing for WebSockets. Here's a closer look:

```
python
Copy code
path('ws/chat/<str:room_name>/', ChatConsumer.as_asgi()),
```

This will match WebSocket requests at the URL `/ws/chat/<room_name>/`, where `room_name` can be any string that identifies the chat room.

7. Creating a Frontend with WebSocket

For the frontend, you can use JavaScript to connect to the WebSocket.

Here's an example of how to create a simple HTML template with a WebSocket connection:

```
html
Copy code
<!-- templates/chat/room.html -->
<!DOCTYPE html>
```

```
<html>
<head>
    <title>Real-time Chat</title>
    <style>
        #chat-log {
            border: 1px solid #ccc;
            height: 200px;
            overflow-y: scroll;
            margin-bottom: 10px;
        }
    </style>
</head>
<body>
    <h1>Chat Room: {{ room_name }}</h1>

 <div id="chat-log"></div>
    <input id="chat-message-input" type="text" placeholder="Type your message">
    <button id="chat-message-input-btn">Send</button>

    <script>
        const roomName = "{{ room_name }}";
        const chatSocket = new WebSocket('ws://' + window.location.host + '/ws/chat/'
+ roomName + '/');

        chatSocket.onmessage = function(e) {
            const data = JSON.parse(e.data);
            document.querySelector('#chat-log').innerHTML += '<p>' + data.message +
'</p>';
        };

        document.querySelector('#chat-message-input-btn').onclick = function(e) {
            const messageInputDom = document.querySelector('#chat-message-input');
            const message = messageInputDom.value;
            chatSocket.send(JSON.stringify({ 'message': message }));
            messageInputDom.value = '';
        };
    </script>
</body>
</html>
```

8. Views for Serving the Template

In your Django app, create a view to render the chat room page:

```
python
Copy code
# chat/views.py
from django.shortcuts import render

def room(request, room_name):
    return render(request, 'chat/room.html', {
        'room_name': room_name
    })
```

Add a URL pattern for this view in your app's `urls.py`:

```
python
Copy code
# chat/urls.py
from django.urls import path
from . import views

urlpatterns = [
```

```
    path('<str:room_name>/', views.room, name='room'),
]
```

And make sure to include the app's URLs in your main `urls.py`:

```
python
Copy code
# realtime_app/urls.py
from django.contrib import admin
from django.urls import path, include

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

9. Running the Application

Make sure Redis is running, and then run your Django project:

```
bash
Copy code
python manage.py runserver
```

Navigate to `http://127.0.0.1:8000/chat/some_room/` in your browser to see the real-time chat in action. Multiple users can open the same room in different browsers, and they will see messages in real-time.

15. Django with Redis for Caching and Session Management

Steps to Set Up Django with Redis for Caching and Session Management

1. Install Redis and Required Packages

First, you'll need to install the `redis` package for Python and the `django-redis` package which helps Django interact with Redis more easily.

```
bash
Copy code
pip install redis django-redis
```

Ensure you have Redis installed and running on your local machine or use a managed Redis service.

2. Configure Redis for Caching in Django

In Django, you can use Redis as a caching backend by configuring it in your `settings.py`.

Caching Configuration

In your `settings.py`, add the following cache configuration to use Redis:

```
python
```

```
Copy code
# settings.py

CACHES = {
    'default': {
        'BACKEND': 'django_redis.cache.RedisCache',
        'LOCATION': 'redis://127.0.0.1:6379/1',  # Redis server URL (127.0.0.1:6379,
database 1)
        'OPTIONS': {
            'CLIENT_CLASS': 'django_redis.client.DefaultClient',
        },
    }
}
```

Explanation of parameters:

- **BACKEND**: `django_redis.cache.RedisCache` is the backend class for caching using Redis.
- **LOCATION**: The Redis server location. Here it points to `localhost` (`127.0.0.1`) and uses the second Redis database (`/1`).
- **OPTIONS**: These are additional configurations. You can use `'CLIENT_CLASS': 'django_redis.client.DefaultClient'` to specify a default client class.

3. Configure Redis for Session Management

To store Django sessions in Redis (instead of the default database), you can set up session management like this:

Session Configuration

In your `settings.py`, add the following configuration for session management:

```
python
Copy code
# settings.py

# Session engine to use Redis as a session store
SESSION_ENGINE = "django.contrib.sessions.backends.cache"
SESSION_CACHE_ALIAS = "default"  # Use the default cache configuration
```

With the above settings, Django will store session data in Redis instead of the default database-backed session engine.

4. Using Redis for Cache in Views

Once Redis caching is set up, you can use it in your views to store and retrieve cached data.

Example: Caching a view in Django.

```
python
Copy code
# views.py
from django.shortcuts import render
from django.core.cache import cache

def cached_view(request):
    # Try to get the cached data
    cached_data = cache.get('some_key')

    if not cached_data:
```

```
        # If cache is empty, calculate the value and cache it
        cached_data = "This is some expensive data or result"
        # Set cache with a timeout of 60 seconds
        cache.set('some_key', cached_data, timeout=60)

    return render(request, 'cached_view.html', {'cached_data': cached_data})
```

5. Using Redis for Sessions

Once you have Redis configured for session management, you can use the Django session framework just like you normally would. The only difference is that session data will now be stored in Redis.

Example of setting and getting session data:

```
python
Copy code
# views.py
from django.shortcuts import render

def set_session(request):
    # Set session data
    request.session['user_name'] = 'John Doe'

return render(request, 'set_session.html', {'user_name': 'John Doe'})

def get_session(request):
    # Get session data
    user_name = request.session.get('user_name', 'Guest')
    return render(request, 'get_session.html', {'user_name': user_name})
```

In this example:

- **set_session**: Stores the `user_name` in the session.
- **get_session**: Retrieves the `user_name` from the session.

Optional: Using Redis for Other Caching Features

- **Cache versioning**: You can use cache versioning to prevent old versions of cache from being used.
- **Cache timeout**: Use `cache.set()` with a `timeout` argument to control how long the data should remain in the cache.
- **Cache keys**: Store and retrieve data from Redis using unique keys, e.g., `cache.get('user_data_%s' % user_id)`.

Full Example: Setting Up Redis with Caching and Session Management

Here is a full example integrating both caching and session management in Django with Redis.

1. **Install the packages**:

```
bash
Copy code
pip install redis django-redis
```

2. **Configure `settings.py`**:

```
python
Copy code
# settings.py
```

```
# Use Redis for caching
CACHES = {
    'default': {
        'BACKEND': 'django_redis.cache.RedisCache',
        'LOCATION': 'redis://127.0.0.1:6379/1',
        'OPTIONS': {
            'CLIENT_CLASS': 'django_redis.client.DefaultClient',
        },
    }
}

# Use Redis for session management
SESSION_ENGINE = "django.contrib.sessions.backends.cache"
SESSION_CACHE_ALIAS = "default"

# Optional: Use Redis as the default backend for the Django Channels (if you want
WebSockets support)
CHANNEL_LAYERS = {
    'default': {
        'BACKEND': 'channels_redis.core.RedisChannelLayer',
        'CONFIG': {
            'hosts': [('127.0.0.1', 6379)],
        },
    },
}
```

3. **Create Views for Caching and Session Management**:

```
python
Copy code

# views.py
from django.shortcuts import render
from django.core.cache import cache

# Example to cache a view's data
def cached_view(request):
    # Try to get the cached data
    cached_data = cache.get('some_key')

    if not cached_data:
        # If cache is empty, calculate the value and cache it
        cached_data = "This is some expensive data or result"
        # Set cache with a timeout of 60 seconds
        cache.set('some_key', cached_data, timeout=60)

    return render(request, 'cached_view.html', {'cached_data': cached_data})

# Example to set session data
def set_session(request):
    # Set session data
    request.session['user_name'] = 'John Doe'
    return render(request, 'set_session.html', {'user_name': 'John Doe'})

# Example to get session data
def get_session(request):
    # Get session data
    user_name = request.session.get('user_name', 'Guest')
    return render(request, 'get_session.html', {'user_name': user_name})
```

4. **Create Templates for the Views**:

```
html
```

```
Copy code
<!-- cached_view.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Cache Example</title>
</head>
<body>
    <h1>Cached Data: {{ cached_data }}</h1>
</body>
</html>
html
Copy code
<!-- set_session.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Set Session Example</title>
</head>
<body>
    <h1>Session Data: {{ user_name }}</h1>
</body>
</html>
html
Copy code
<!-- get_session.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Get Session Example</title>
</head>
<body>
    <h1>Session Data: {{ user_name }}</h1>
</body>
</html>
```

5. **Add URL Routing**:

```
python
Copy code
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('cached/', views.cached_view, name='cached_view'),
    path('set-session/', views.set_session, name='set_session'),
    path('get-session/', views.get_session, name='get_session'),
]
```

6. **Run the Project**:

Make sure Redis is running locally or via a remote server.

```
bash
Copy code
redis-server
python manage.py runserver
```

Now you can visit:

- `/cached/` to see the cached data.

- `/set-session/` to set session data.
- `/get-session/` to get session data.

16. AIM :Develop a Multi-Tenant SaaS Application

DESCRIPTION:

Steps to Develop a Multi-Tenant SaaS Application in Django

1. Create a Django Project and App

Let's start by creating a new Django project and a `tenant` app.

```
bash
Copy code
django-admin startproject saas_project
cd saas_project
python manage.py startapp tenant
```

2. Install Dependencies

You'll need to install `django-tenants`, a package designed for multi-tenancy, which helps with managing tenants in a shared database schema.

```
bash
Copy code
pip install django-tenants
```

3. Configure `settings.py`

Modify the `settings.py` to include the necessary configurations for multi-tenancy. Update `INSTALLED_APPS`, `DATABASES`, and middleware settings.

```
python
Copy code
# settings.py

INSTALLED_APPS = [
    # Default Django apps...
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # Tenant apps
    'tenant',
    'django_tenants',  # django-tenants app
]

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'saas_db',  # Single shared database
        'USER': 'your_db_user',
        'PASSWORD': 'your_db_password',
        'HOST': 'localhost',
        'PORT': '5432',
```

```
    }
}

# Use django-tenants middleware
MIDDLEWARE = [
    'django_tenants.middleware.TenantMiddleware',  # Handles tenant-specific routing
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]

# Define the domain for each tenant
TENANT_MODEL = "tenant.Tenant"  # The model to represent tenants
TENANT_DOMAIN_MODEL = "tenant.Domain"  # The model to represent tenant domains
```

4. Create Tenant Models

You will need to create models to represent your tenants and their domains. A tenant typically has a name and a domain that maps to the tenant's specific subdomain or URL.

```
python
Copy code
# tenant/models.py

from django.db import models
from django_tenants.models import TenantMixin

class Tenant(TenantMixin):
    name = models.CharField(max_length=255)
    created_on = models.DateField(auto_now_add=True)

    def __str__(self):
        return self.name

class Domain(models.Model):
    tenant = models.ForeignKey(Tenant, related_name='domains',
on_delete=models.CASCADE)
    domain = models.CharField(max_length=253)  # E.g., 'tenant1.myapp.com'
    is_primary = models.BooleanField(default=True)

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

The `Tenant` model inherits from `TenantMixin`, which is provided by `django-tenants` and manages tenant-specific configurations.

5. Define Tenant-Aware Models

To ensure that each tenant's data is isolated, you need to define tenant-specific models. You can do this by inheriting from `TenantModel` provided by `django-tenants`.

```
python
Copy code
# tenant/models.py (continued)

from django_tenants.models import TenantModel

class Product(TenantModel):
```

```
    tenant = models.ForeignKey(Tenant, on_delete=models.CASCADE)
    name = models.CharField(max_length=255)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    description = models.TextField()

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

In the `Product` model, the `tenant` field ensures that each product is associated with a particular tenant.

6. Set Up URLs for Each Tenant

To differentiate between tenants, you can use subdomains. You will need to modify the `urls.py` to handle different tenant URLs.

```
python
Copy code
# saas_project/urls.py

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('tenant/', include('tenant.urls')),  # Tenant-specific URLs
]
```

For each tenant, you can add a specific view and URL that is unique to the tenant.

7. Tenant-Specific Views

Create views that will serve tenant-specific data. For example, let's create a view that displays products.

```
python
Copy code
# tenant/views.py

from django.shortcuts import render
from .models import Product

def product_list(request):
    tenant = request.tenant  # Access the tenant from the request
    products = Product.objects.filter(tenant=tenant)
    return render(request, 'tenant/product_list.html', {'products': products})
```

8. Create Templates for Tenant Views

Create a template to render the products for the tenant.

```
html
Copy code
<!-- tenant/templates/tenant/product_list.html -->

<!DOCTYPE html>
<html>
<head>
    <title>Tenant Products</title>
</head>
<body>
    <h1>Products for {{ request.tenant.name }}</h1>
    <ul>
```

```
        {% for product in products %}
            <li>{{ product.name }} - {{ product.price }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

9. Set Up URLs for Tenant-Specific Views

Now, set up tenant-specific URLs to display the products.

```
python
Copy code
# tenant/urls.py

from django.urls import path
from . import views

urlpatterns = [
    path('products/', views.product_list, name='product_list'),
]
```

10. Create the Tenant Database Schema

When you run `migrate`, `django-tenants` will create separate schemas in the same database for each tenant.

```
bash
Copy code
python manage.py migrate_schemas --shared
```

This will create the shared schema (for models like `Tenant` and `Domain`), and then create individual schemas for each tenant when a new tenant is created.

11. Create a Tenant and Add Domain

To create a tenant, you can use the Django shell:

```
bash
Copy code
python manage.py shell
python
Copy code
from tenant.models import Tenant, Domain

# Create a new tenant
tenant = Tenant(name="Tenant 1")
tenant.save()

# Add a domain for the tenant
domain = Domain(tenant=tenant, domain="tenant1.myapp.com", is_primary=True)
domain.save()
```

12. Tenant Middleware

The `django-tenants` middleware will automatically route requests to the correct tenant based on the subdomain or domain. Make sure to have the middleware configured correctly, as shown in the `settings.py` earlier.

13. Run the Application

Once you've set up everything, you can start the Django development server.

```
bash
Copy code
python manage.py runserver
```

Now you can access tenant-specific data via URLs like:

- `http://tenant1.myapp.com/tenant/products/` for **Tenant 1**
- `http://tenant2.myapp.com/tenant/products/` for **Tenant 2** (if you add another tenant).

17. AIM :Create a Social Media Platform with Django

DESCRIPTION:

Step-by-Step Code

1. Setting Up Django Project and App

First, create a new Django project and app:

```
bash
Copy code
django-admin startproject social_media
cd social_media
python manage.py startapp accounts
python manage.py startapp posts
```

2. Install Dependencies

You will need `django` and `django-allauth` (for user authentication, optional but useful):

```
bash
Copy code
pip install django django-allauth
```

3. Configure `settings.py`

Update `settings.py` to include necessary apps and middleware:

```
python
Copy code
# settings.py

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

    'accounts',  # Custom user app
    'posts',     # App for posts
    'django.contrib.sites',  # Needed for allauth
    'allauth',  # Allauth for authentication
```

```
    'allauth.account',  # Account management via allauth
    'allauth.socialaccount',  # Social login (optional)
    'django.contrib.sites',
]

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

AUTHENTICATION_BACKENDS = (
    'allauth.account.auth_backends.AuthenticationBackend',
)

# Use email as the unique identifier for users
ACCOUNT_AUTHENTICATED_LOGIN_REDIRECTS = True


ACCOUNT_EMAIL_REQUIRED = True
ACCOUNT_EMAIL_VERIFICATION = "mandatory"
LOGIN_REDIRECT_URL = "/"

# Sites framework for handling multiple domains (needed for allauth)
SITE_ID = 1
```

Make sure to set up `urls.py` for `allauth`:

```
python
Copy code
# social_media/urls.py

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/', include('allauth.urls')),  # Add allauth urls for authentication
]
```

4. Create User Profile Model

In the `accounts` app, create a model for user profiles. Users can have additional information such as bio, profile picture, etc.

```
python
Copy code
# accounts/models.py

from django.contrib.auth.models import User
from django.db import models

class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    bio = models.TextField(blank=True, null=True)
    profile_picture = models.ImageField(upload_to='profiles/', blank=True, null=True)

    def __str__(self):
        return self.user.username
```

5. User Registration and Profile Update

We can use `allauth` for user registration, but you may want to allow users to update their profile after registration. Here is a form to handle the profile update:

```
python
Copy code
# accounts/forms.py

from django import forms
from .models import Profile

class ProfileUpdateForm(forms.ModelForm):
    class Meta:
        model = Profile
        fields = ['bio', 'profile_picture']
```

And a view to handle this:

```
python
Copy code
# accounts/views.py

from django.shortcuts import render, redirect
from .forms import ProfileUpdateForm
from django.contrib.auth.decorators import login_required

@login_required
def profile_view(request):
    profile, created = Profile.objects.get_or_create(user=request.user)
    if request.method == 'POST':
        form = ProfileUpdateForm(request.POST, request.FILES, instance=profile)
        if form.is_valid():
            form.save()
            return redirect('profile')
    else:
        form = ProfileUpdateForm(instance=profile)

    return render(request, 'accounts/profile.html', {'form': form, 'profile':
profile})
```

Create the template for displaying and updating profiles:

```
html
Copy code
<!-- accounts/templates/accounts/profile.html -->

{% extends 'base_generic.html' %}

{% block content %}
<h2>{{ user.username }}'s Profile</h2>
<img src="{{ profile.profile_picture.url }}" alt="Profile Picture">
<form method="POST" enctype="multipart/form-data">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Update Profile</button>
</form>
{% endblock %}
```

6. Create the Post Model

Now, create the `Post` model in the `posts` app. Each post should belong to a user and have a text content.

```
python
Copy code
# posts/models.py

from django.db import models
from django.contrib.auth.models import User

class Post(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"{self.user.username}'s post"

    class Meta:
        ordering = ['-created_at']
```

7. Create Views for Posts

Now create views to allow users to post content and display posts.

```
python
Copy code
# posts/views.py

from django.shortcuts import render, redirect
from .models import Post
from django.contrib.auth.decorators import login_required

@login_required
def post_create(request):
    if request.method == 'POST':
        content = request.POST.get('content')
        if content:
            Post.objects.create(user=request.user, content=content)
            return redirect('post_list')
    return render(request, 'posts/post_create.html')

def post_list(request):
    posts = Post.objects.all()
    return render(request, 'posts/post_list.html', {'posts': posts})
```

8. Create Post Templates

For displaying posts and creating new ones, you need templates.

```
html
Copy code
<!-- posts/templates/posts/post_create.html -->
{% extends 'base_generic.html' %}

{% block content %}
<h2>Create a Post</h2>
<form method="POST">
    {% csrf_token %}
    <textarea name="content" placeholder="What's on your mind?"></textarea>
    <button type="submit">Post</button>
</form>
{% endblock %}
html
Copy code
<!-- posts/templates/posts/post_list.html -->
```

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

{% block content %}
<h2>All Posts</h2>
{% for post in posts %}
    <div>
        <h4>{{ post.user.username }}:</h4>
        <p>{{ post.content }}</p>
        <small>{{ post.created_at }}</small>
    </div>
{% endfor %}
{% endblock %}
```

9. Create the Follow System

For a simple follow/unfollow system, you can create a `Follow` model:

```
python
Copy code
# accounts/models.py (continued)

class Follow(models.Model):
    user = models.ForeignKey(User, related_name='followers', on_delete=models.CASCADE)
    following = models.ForeignKey(User, related_name='following',
on_delete=models.CASCADE)

    def __str__(self):
        return f"{self.user.username} follows {self.following.username}"
```

Add views to follow and unfollow users:

```
python
Copy code
# accounts/views.py (continued)

from .models import Follow

@login_required
def follow_user(request, username):
    user_to_follow = User.objects.get(username=username)
    if user_to_follow != request.user:
        Follow.objects.get_or_create(user=request.user, following=user_to_follow)
    return redirect('post_list')

@login_required
def unfollow_user(request, username):
    user_to_unfollow = User.objects.get(username=username)
    Follow.objects.filter(user=request.user, following=user_to_unfollow).delete()
    return redirect('post_list')
```

10. Add URLs for Follow Views

Now, set up the URLs for the follow/unfollow actions:

```
python
Copy code
# accounts/urls.py

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

```
urlpatterns = [
    path('profile/', views.profile_view, name='profile'),
    path('follow/<str:username>/', views.follow_user, name='follow_user'),
    path('unfollow/<str:username>/', views.unfollow_user, name='unfollow_user'),
]
```

11. Set Up URL Routing

Set up URLs for the post views in `posts/urls.py`:

```
python
Copy code
# posts/urls.py

from django.urls import path
from . import views

urlpatterns = [
    path('', views.post_list, name='post_list'),
    path('create/', views.post_create, name='post_create'),
]
```

Include these URLs in the main `urls.py`:

```
python
Copy code
# social_media/urls.py

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