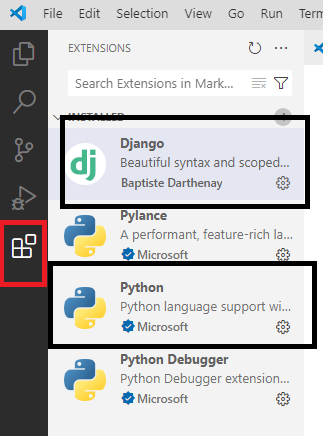
01-june-2024

**VS Code**

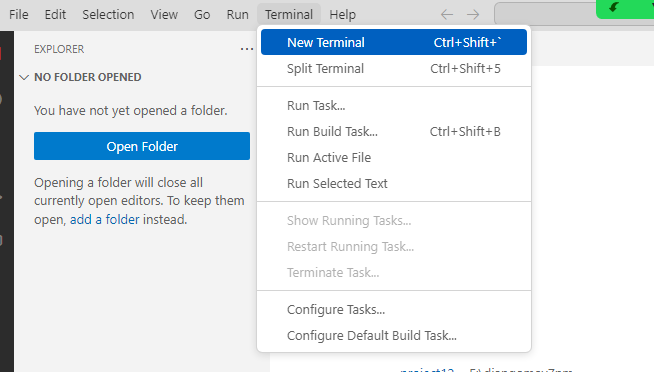
1. Download VS Code

<https://code.visualstudio.com/download>

1. Add Extensions

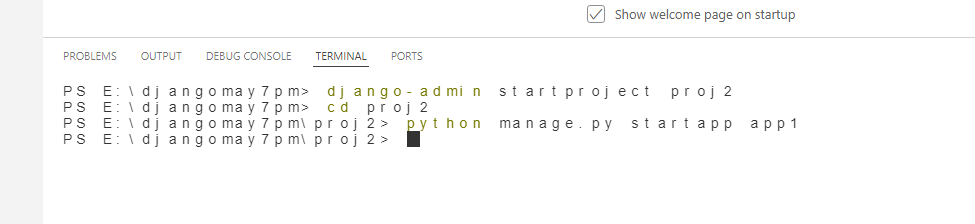


1. Select Terminal

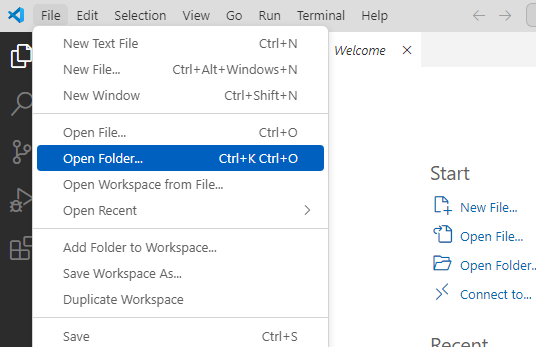


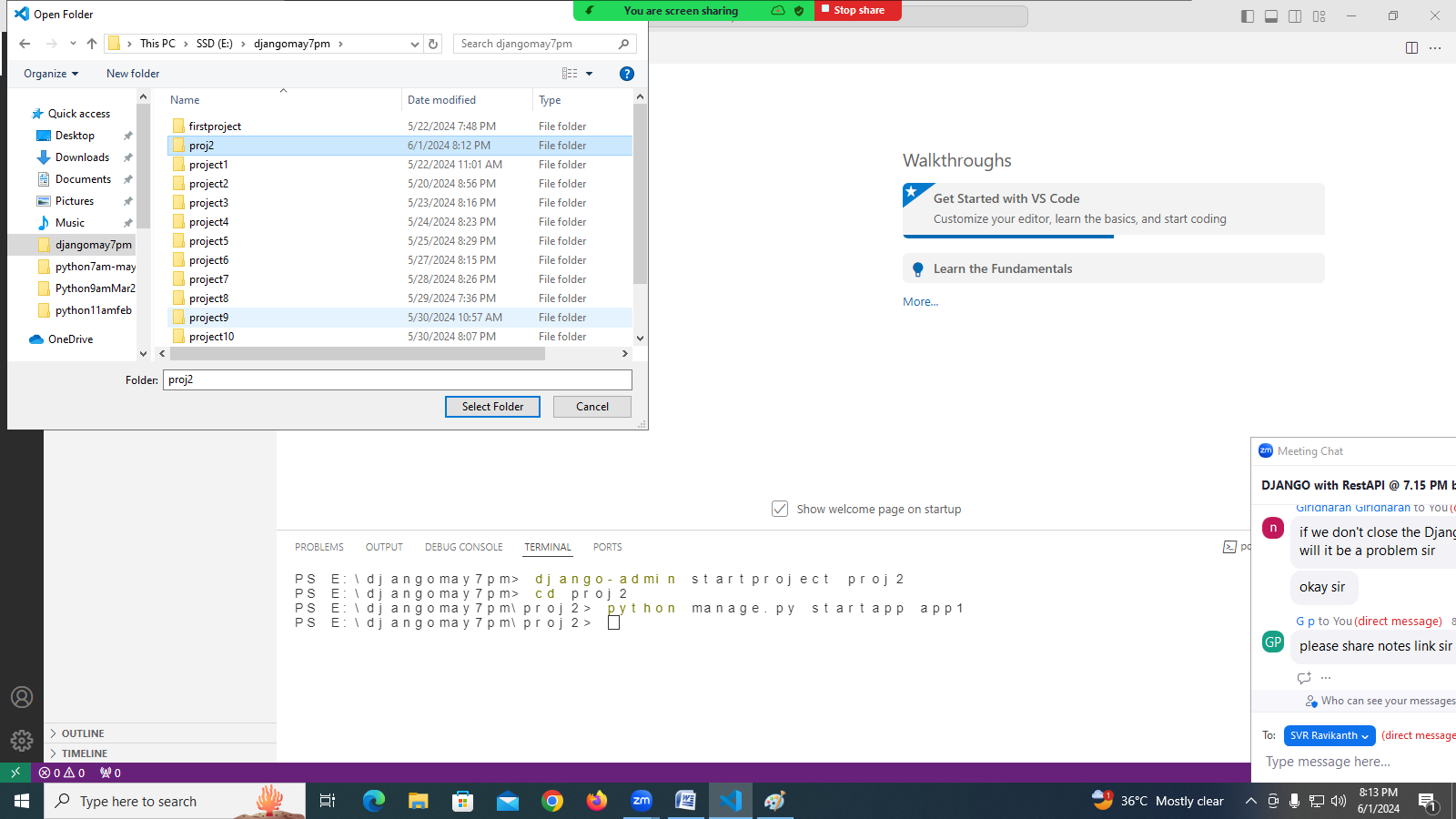
4. Create project

5. Create application

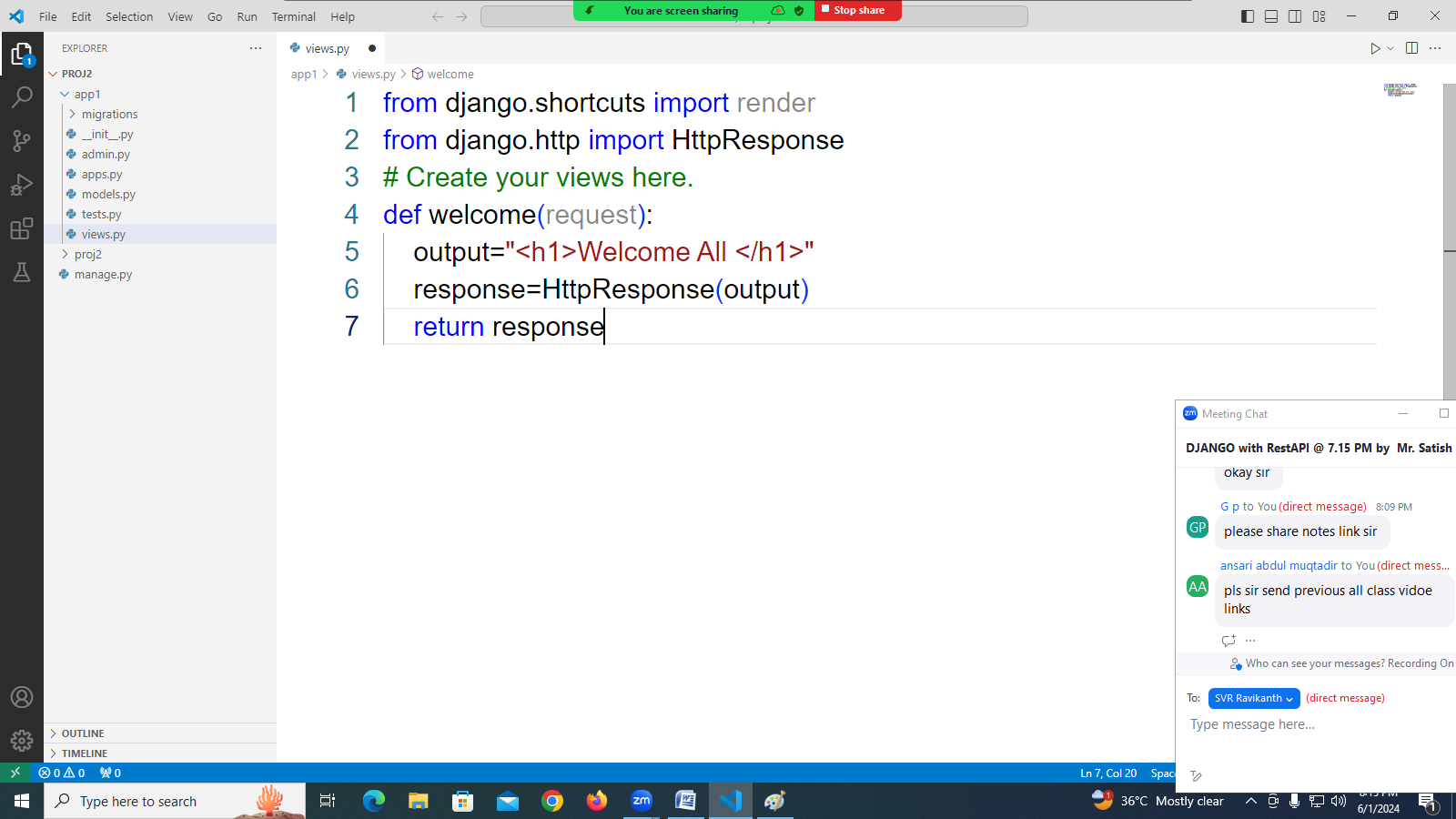


6. Open Project folder

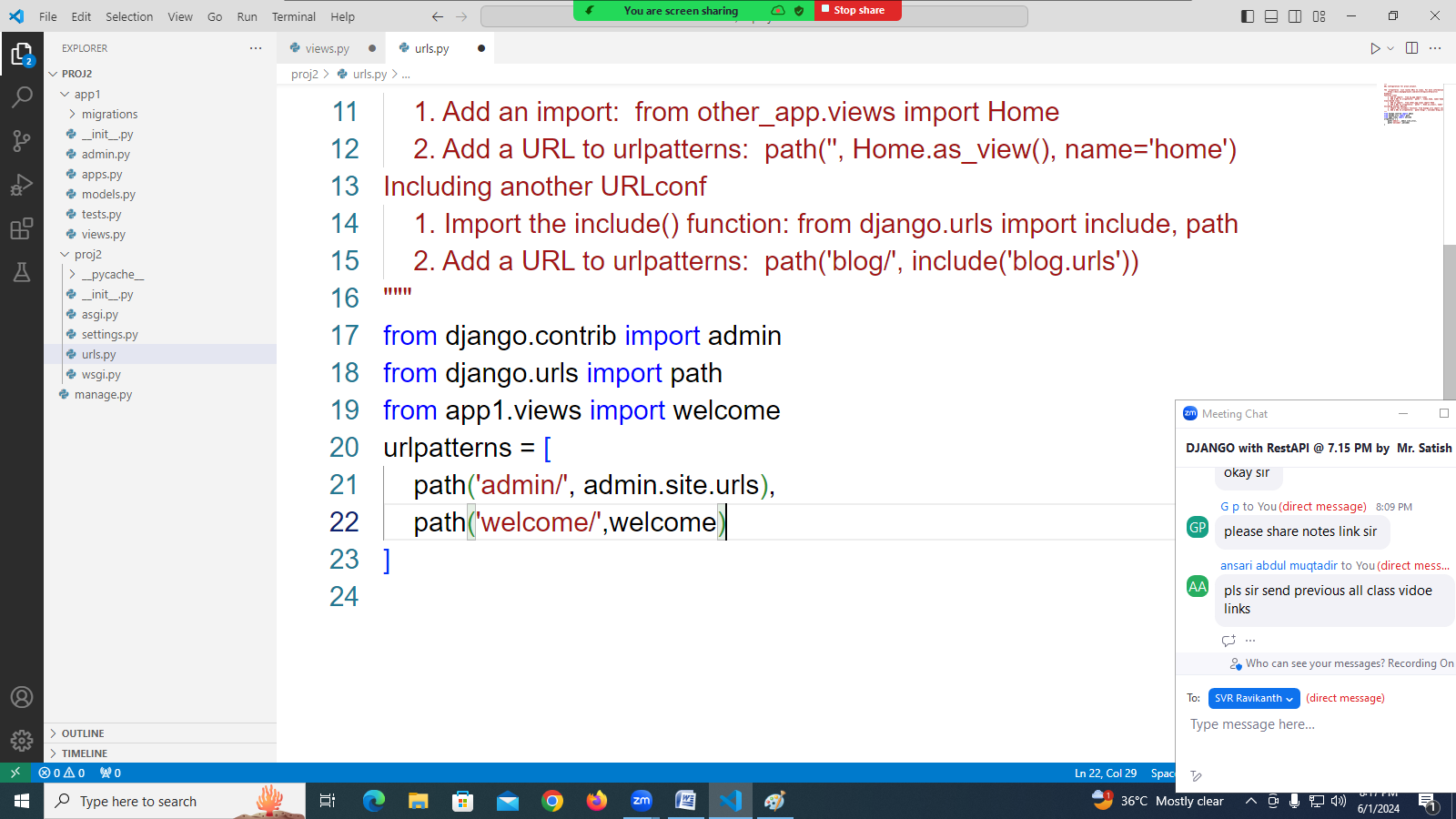




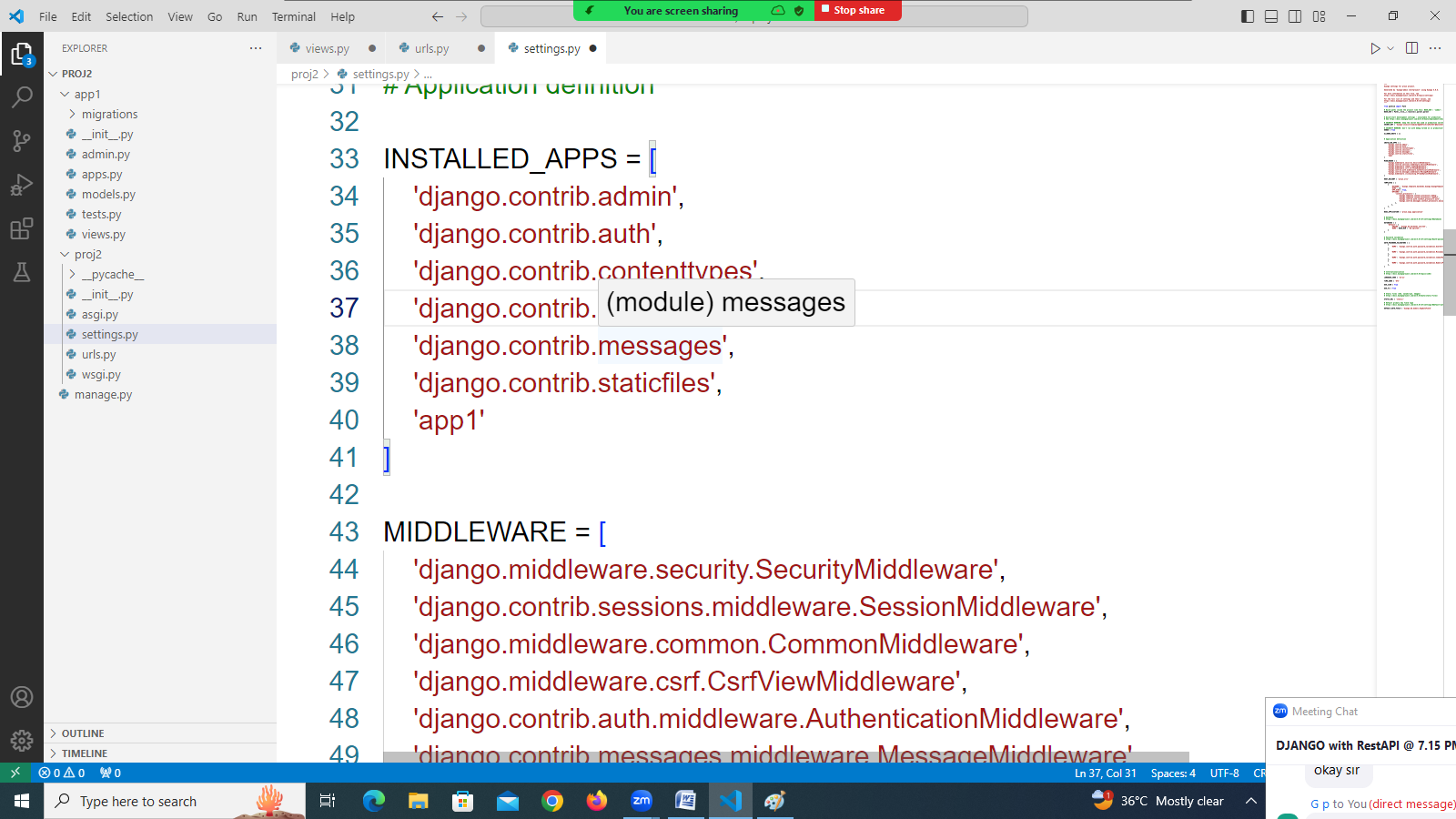
7. Open Views.py and write view function



8. Open urls.py

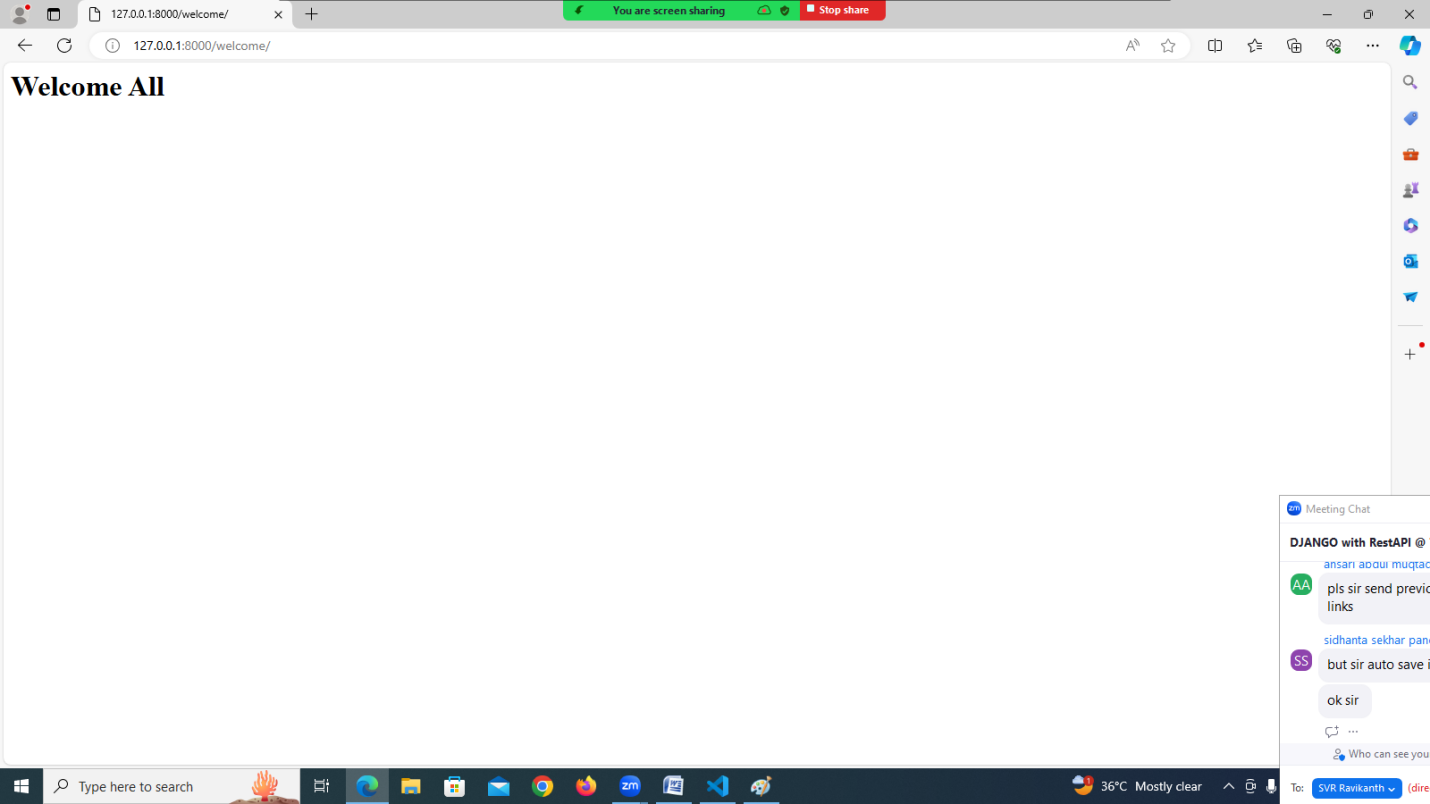


9. Open settings.py



10. File 🡪 Save All

11. Run Server

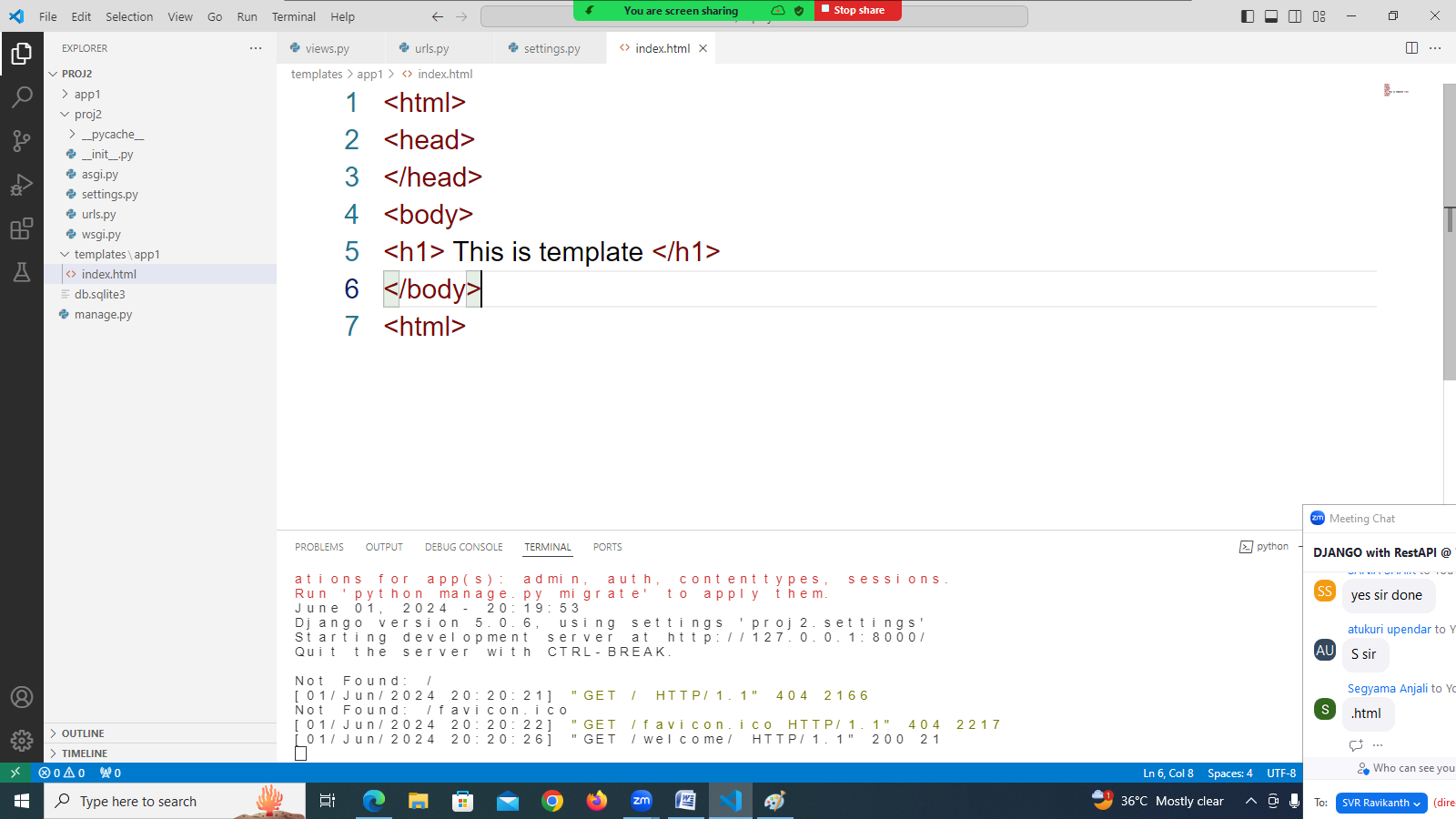


12. Adding templates

13. Create templates folder inside container folder

14. Inside templates folder create folder with application-name

15. Inside templates/application-folder create html file



16. Configure temples in settings.py

TEMPLATES\_DIR=BASE\_DIR/'templates'

TEMPLATES = [

    {

        'BACKEND': 'django.template.backends.django.DjangoTemplates',

        'DIRS': [TEMPLATES\_DIR],

18. Runserver

03-06-2024

**Django Template Language (DTL)**

Django Template Language is a markup language. It provides special syntax for making template as dynamic.

Template text file or html file, HTML is static it is used for presentation of data or create user interface.

In order make template is dynamic we are using DTL. It provides the features programming language (variables, control statements,…).

DTL is provides various syntax for working with templates.

1. Template Variables
2. Template Tags
3. Template filters
4. Template Inheritance

**Template Variables**

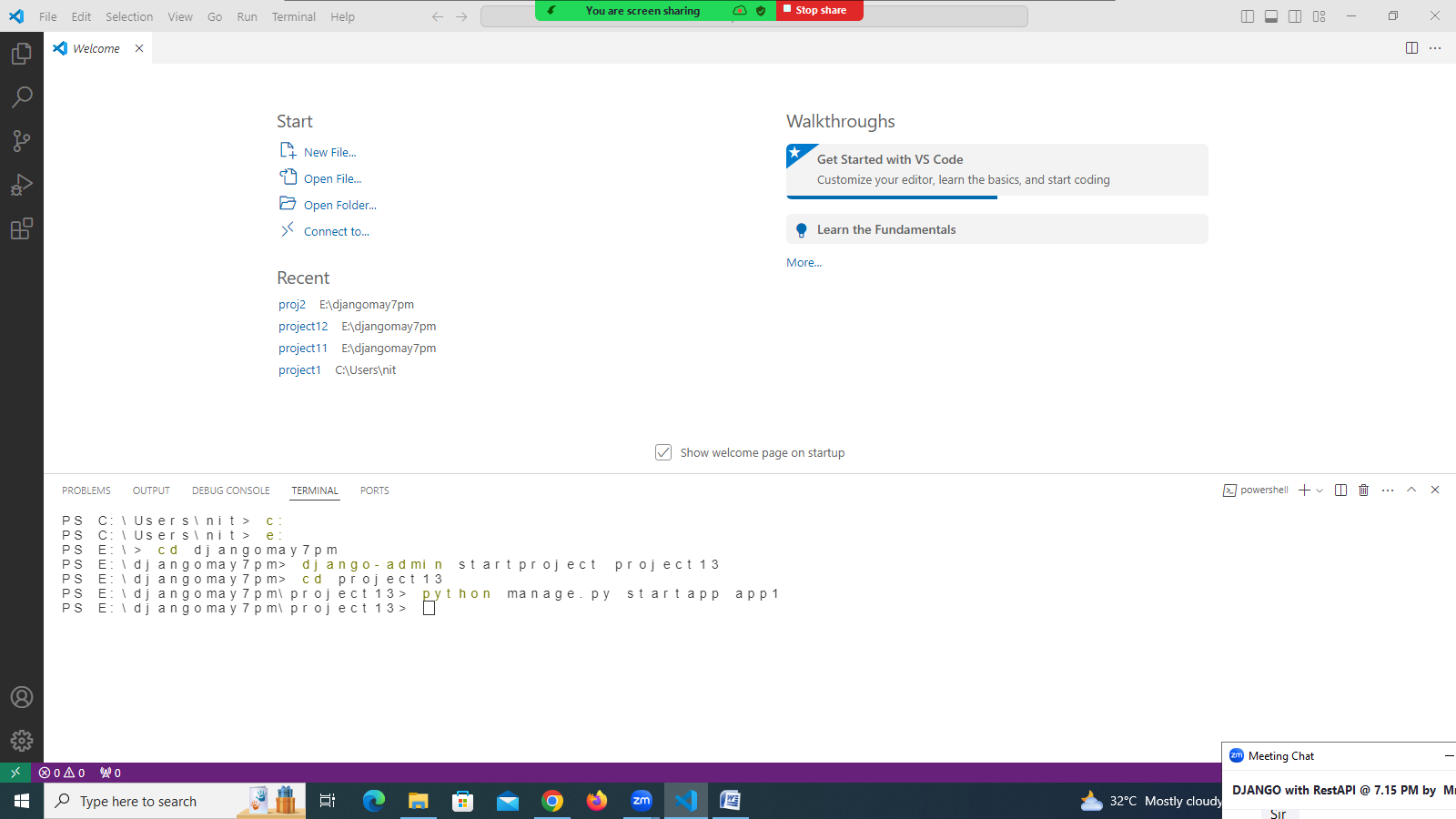
Within template variables can be inserted using template variables syntax.

1. Variable
2. List
3. Object
4. Dictionary

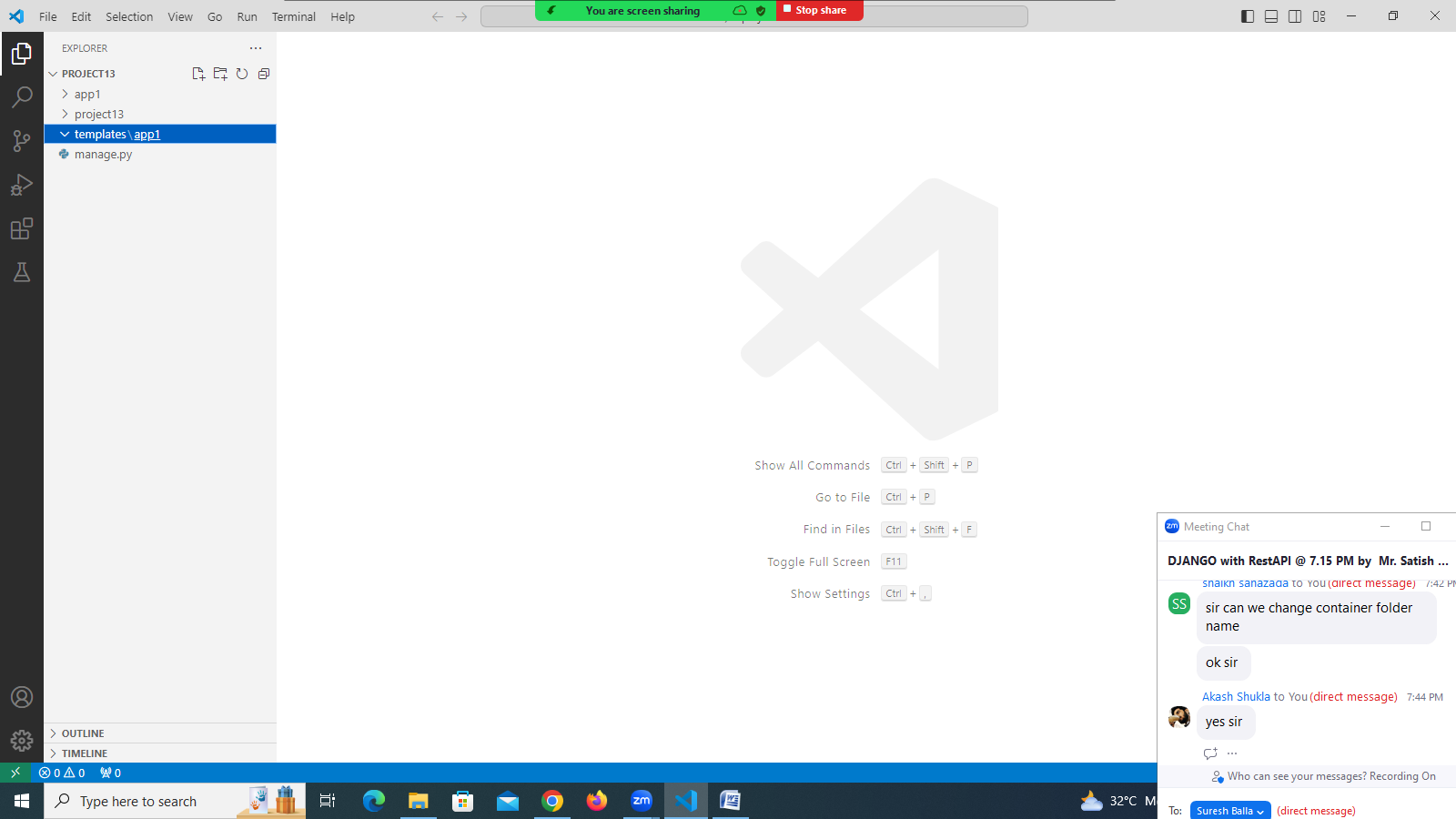
**Syntax: {{ variable }}**

**Example:**

1. **Open terminal in vscode and create project and applictation**



1. Open project within VS Code
   1. File
   2. Open Folder
2. Create templates folder inside container folder
3. Create folder inside templates folder with application name



1. Open views.py

from django.shortcuts import render

# Create your views here.

def test\_view(request):

    a=10 # Local Variables

    b=20 # Local Variables

    c=a+b

    c={'n1':a,'n2':b,'n3':c}

    return render(request,'app1/result.html',context=c)

**result.html**

<html>

<head>

    <title> Django Template Langauge </title>

</head>

<body>

    <h1> Sum of Two Numbers </h1>

    <h2> Value of a is {{n1}} </h2>

    <h2> Value of b is {{n2}} </h2>

    <h2> Sum is {{n3}} </h2>

</body>

</html>

**Urls.py**

from django.contrib import admin

from django.urls import path

from app1.views import test\_view

urlpatterns = [

    path('admin/', admin.site.urls),

    path('sum/',test\_view)

]

**Settings.py**

BASE\_DIR = Path(\_\_file\_\_).resolve().parent.parent

TEMPLATES\_DIR=BASE\_DIR/'templates'

INSTALLED\_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    'app1'

]

TEMPLATES = [

    {

        'BACKEND': 'django.template.backends.django.DjangoTemplates',

        'DIRS': [TEMPLATES\_DIR],

**How to read list values inside template**

1. Index
2. for loop

**using index**

This allows reading values from list sequential and random.

**{{list-name.index}}**

**Views.py**

from django.shortcuts import render

# Create your views here.

def test\_view(request):

    courses\_list=["PYTHON","JAVA",".NET","ORACLE"]

    c={'list1':courses\_list}

    return render(request,'app1/result.html',context=c)

**result.html**

<html>

<head>

    <title> Django Template Langauge </title>

</head>

<body>

    <h1>

    <OL>

        <LI> {{list1.0}} </LI>

        <LI> {{list1.1}} </LI>

        <LI> {{list1.2}} </LI>

        <LI> {{list1.3}} </LI>

    </OL>

</h1>

</body>

</html>

**How to values from object?**

Object values inside template are accessed using property name or instance variable name.

**Syntax:**

{{obect-name.property-name}}

**Views.py**

from django.shortcuts import render

# Create your views here.

class Employee:

    def \_\_init\_\_(self):

        self.empno=101

        self.ename="naresh"

        self.salary=5000

def test\_view(request):

    emp=Employee()

    c={'e':emp}

    return render(request,'app1/result.html',context=c)

**result.html**

<html>

<head>

    <title> Django Template Langauge </title>

</head>

<body>

    <h2> Employee Details </h2>

    <table border="1">

        <tr>

            <th>Empno</th>

            <th>Ename</th>

            <th>Salary</th>

        </tr>

        <tr>

            <td>{{e.empno}}</td>

            <td>{{e.ename}}</td>

            <td>{{e.salary}}</td>

        </tr>

    </table>

</body>

</html>

**How to read values from dictionary?**

Within template dictionary values can be read using key.

**Syntax: {{dict-name.key}}**

**Views.py**

from django.shortcuts import render

# Create your views here.

def test\_view(request):

    stud\_data={'rollno':101,

               'name':'naresh',

               'course':'python'}

    c={'stud':stud\_data}

    return render(request,'app1/result.html',context=c)

**result.html**

<html>

<head>

    <title> Django Template Langauge </title>

</head>

<body>

    <h2> Student Details </h2>

    <table border="1">

        <tr>

            <th>Rollno</th>

            <th>Name</th>

            <th>Course</th>

        </tr>

        <tr>

            <td>{{stud.rollno}}</td>

            <td>{{stud.name}}</td>

            <td>{{stud.course}}</td>

        </tr>

    </table>

</body>

</html>

**Template tags**

Template tags are used to perform various operations.

Example: condition statement, Looping statement,…

**Syntax:**

**{% tag-name %}**

**Example: {%if test %}, {%for variable in iterable %}**

**If(<condition>)**

**{**

**}**

**Template tags**

Template tags are used to perform various operations.

Example: condition statement, Looping statement,…

**Syntax:**

**{% tag-name %}**

**Example: {%if test %}, {%for variable in iterable %}**

**Control statement template tags**

1. **Conditional template tags**
2. **Looping template tags**

**Conditional template tags**

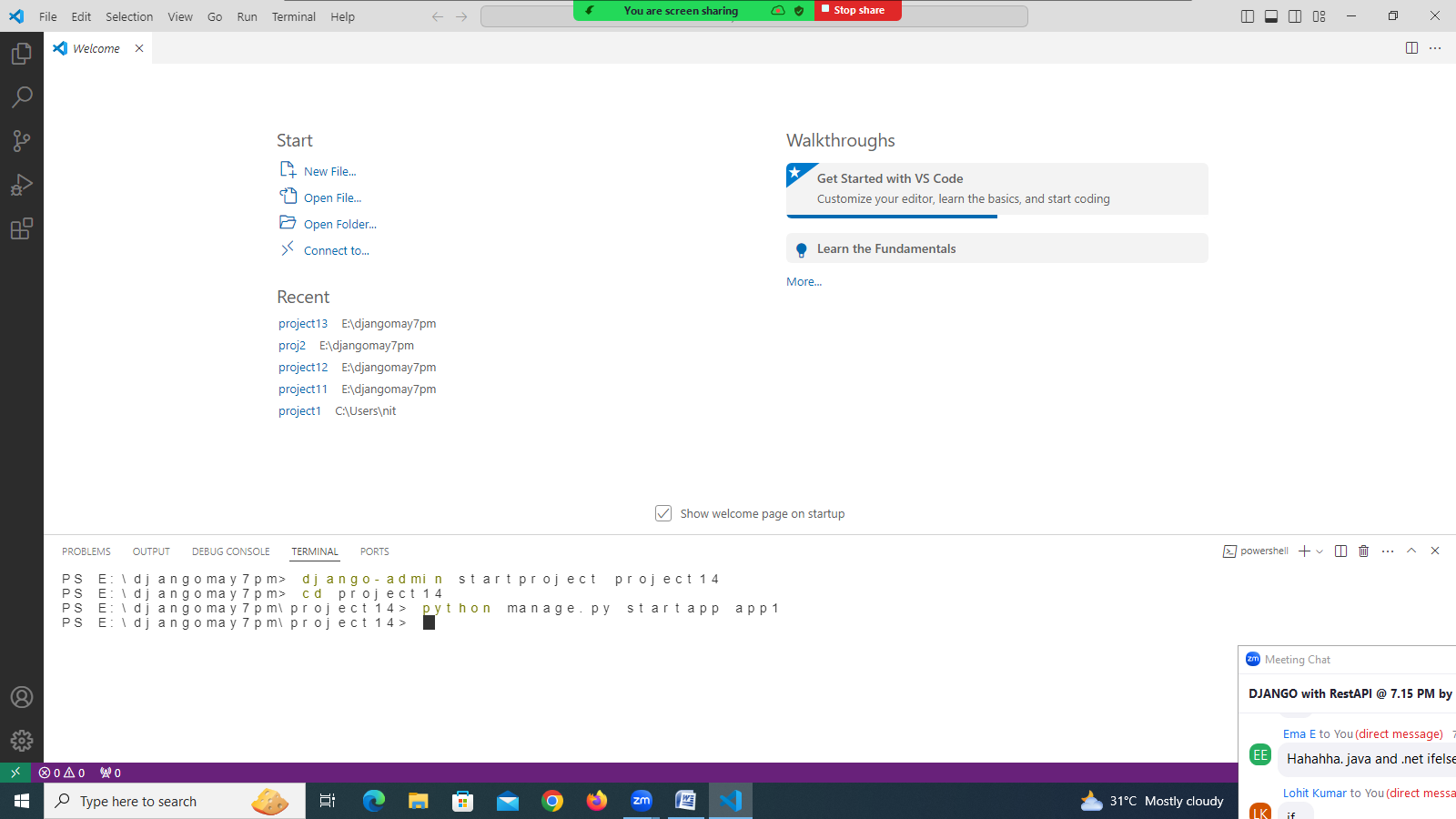
Conditional tags are used to execute block of tags based on condition or test.

1. {%if%}
2. {%else%}
3. {%elif%}
4. {%endif%}

|  |  |  |
| --- | --- | --- |
| **Simple if**  {%if test%}  Tags  {%endif%} | **If..else**  {%if test%}  Tags  {%else%}  Tags  {%endif%} | **If..elif..else (if..else ladder)**  {%if test %}  Tags  {%elif test %}  Tags  {%else%}  Tags  {%endif%} |

if tag must be end with another tag {%endif%}

**Create project and inside project create application**



**Create templates folder inside project container folder**

**Inside templates folder create folder with application name**

**Views.py**

from django.shortcuts import render

# Create your views here.

def test\_view(request,age):

    return render(request,"app1/find.html",context={'age':age})

**templates/app1/find.html**

<html>

<body>

<h1> Voter Elg </h1>

<h2>

{%if age >= 18 %}

Elg to Vote

{%else%}

Not Elg to Vote

{% endif %}

</h2>

</body>

</html>

**Urls.py**

from django.contrib import admin

from django.urls import path

from app1.views import test\_view

urlpatterns = [

    path('admin/', admin.site.urls),

    path('test/<int:age>/',test\_view)

]

**Settings.py**

BASE\_DIR = Path(\_\_file\_\_).resolve().parent.parent

TEMPLATE\_DIR=BASE\_DIR/'templates'

INSTALLED\_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    'app1'

]

TEMPLATES = [

    {

        'BACKEND': 'django.template.backends.django.DjangoTemplates',

        'DIRS': [TEMPLATE\_DIR],

        'APP\_DIRS': True,

        'OPTIONS': {

**Example of if else ladder**

**Views.py**

from django.shortcuts import render

# Create your views here.

def test\_view(request,name,marks):

    return render(request,"app1/find.html",context={'name':name,'marks':marks})

**find.html**

<html>

<body>

<h1> Displaying Grade </h1>

<h2>

Name {{name}} <br>

{%if marks >= 90%}

Grade A

{%elif marks >= 70 and marks < 90 %}

Grade B

{%elif marks >= 50 and marks < 70 %}

Grade C

{%else%}

Grade D

{%endif%}

</h2>

</body>

</html>

**Urls.py**

from django.contrib import admin

from django.urls import path

from app1.views import test\_view

urlpatterns = [

    path('admin/', admin.site.urls),

    path('test/<name>/<int:marks>/',test\_view)

]

**Looping template tag**

Django Template Language provides one looping template tag

{%for variable in iterable%}

{%endfor%}

Looping tag is used to iterate the content of iterables (list, set or dictionary or any other collection)

**Syntax:**

{%for variable in iterable%}

Tags

{%endfor%}

**Reading data from list**

**Views.py**

from django.shortcuts import render

# Create your views here.

def test\_view(request):

    names\_list=["naresh","suresh","ramesh","kishore"]

    return render(request,"app1/find.html",context={'names':names\_list})

**find.html**

<html>

<body>

<h1> Displaying Names </h1>

<h2>

    <UL>

        {%for name in names %}

            <li> {{name}} </li>

        {%endfor%}

    </UL>

</h2>

</body>

</html>

**Urls.py**

from django.contrib import admin

from django.urls import path

from app1.views import test\_view

urlpatterns = [

    path('admin/', admin.site.urls),

    path('test/',test\_view)

]

**Reading data from dictionary**

**Views.py**

from django.shortcuts import render

# Create your views here.

def test\_view(request):

    sales\_dict={2000:45000,2001:57000,2002:65000,2003:78000}

    return render(request,"app1/find.html",context={'sales':sales\_dict})

**find.html**

<html>

<body>

<h1> Displaying Names </h1>

<h2>

    <table border="2">

        <tr>

            <th>Year</th>

            <th>Sales</th>

        </tr>

        {%for year,s in sales.items  %}

        <tr>

            <td>{{year}}</td>

            <td>{{s}}</td>

        </tr>

        {%endfor%}

    </table>

</h2>

</body>

</html>

06-06-2024

**{%include%}**

Include tag allow include content of another template.

This allows reusing the content one template inside many templates.

Syntax

{% include template %}

or

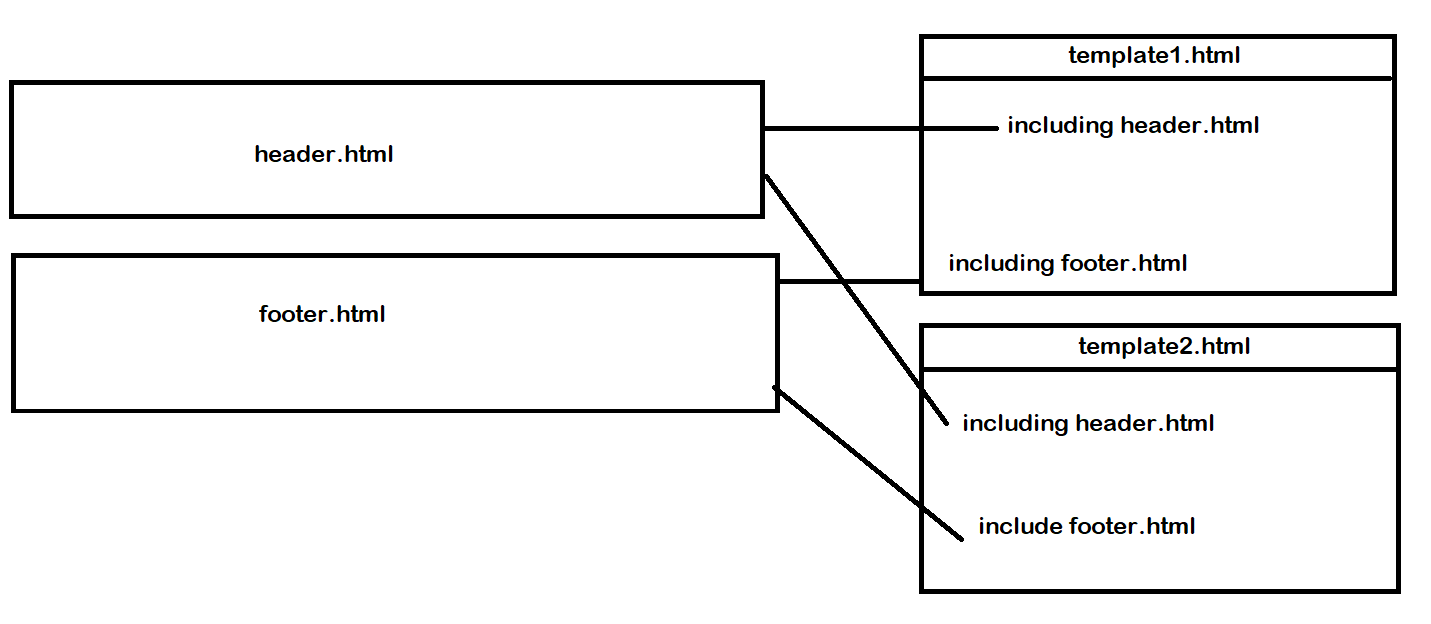
{% include template with key=value%}

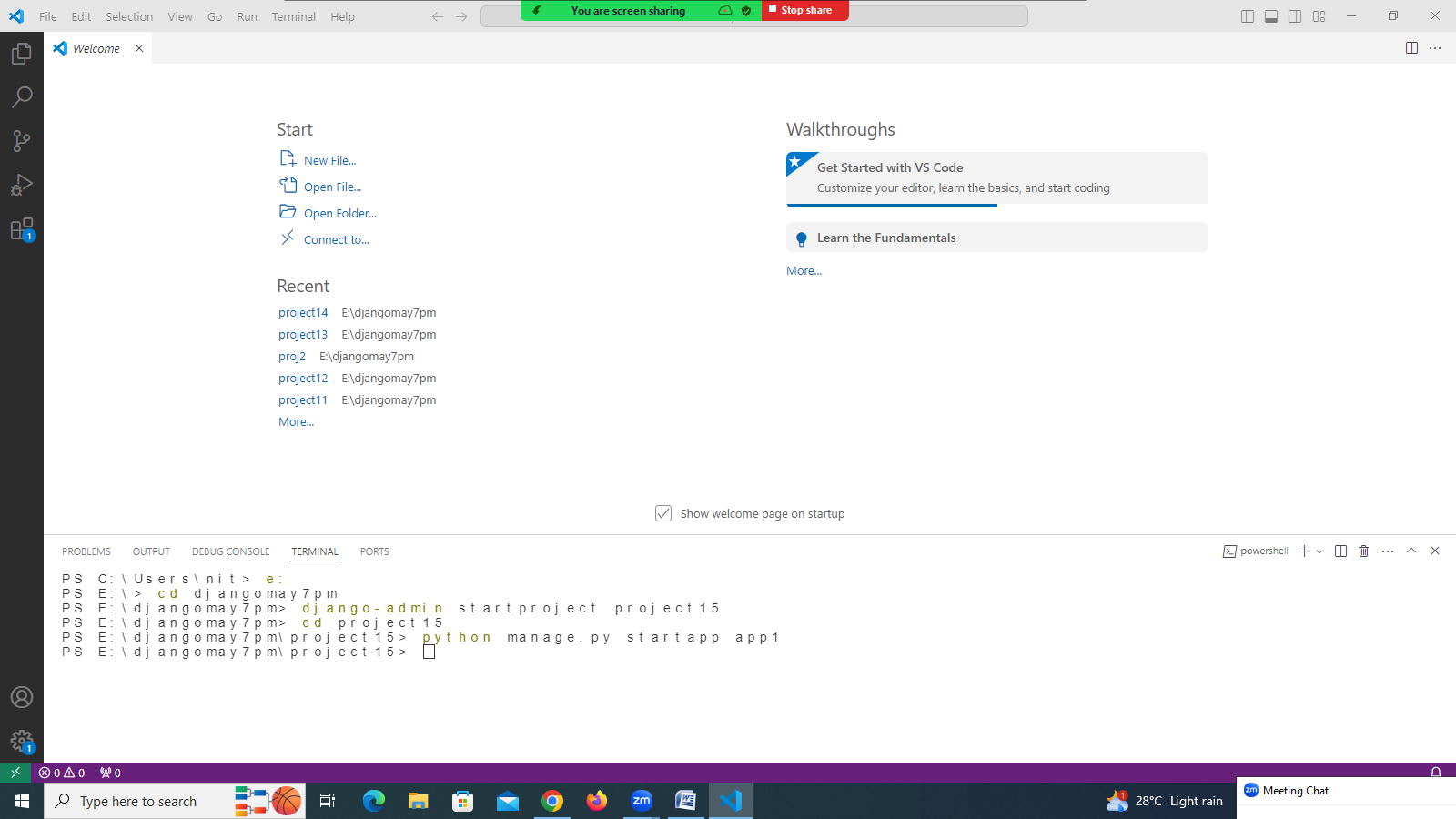
Parameters

|  |  |
| --- | --- |
| Value | Description |
| template | Required. The filename of the template. Either a string or a variable. |
| key=value | Optional. A variable name and value to send into the include file. Used together with the with keyword. You can have as many key/value pairs as you like. |

**This provides**,

1. Reusability
2. Modularity
3. Maintaining code is easy





Inside project container folder create templates folder

Inside templates folder create folder with application name

**header.html**

<html>

<head>

    <style>

        h1{

            color:red;

        }

    </style>

</head>

<body>

    <h1> Naresh I Technologies </h1>

    <h2> Hello {{name}} </h2>

</body>

</html>

**Footer.html**

<html>

<head>

    <style>

        h2{

            color:blue;

        }

    </style>

</head>

<body>

    <h2> This is footer.html </h2>

</body>

</html>

**Content.html**

<html>

<body>

{%include "app1/header.html" with name="XYZ"%}

This is content template

{%include "app1/footer.html" %}

</body>

</html>

**Views.py**

from django.shortcuts import render

# Create your views here.

def display\_view(request):

    return render(request,"app1/content.html")

**urls.py**

from app1.views import display\_view

urlpatterns = [

    path('admin/', admin.site.urls),

    path('display/',display\_view)

]

**Settings.py**

BASE\_DIR = Path(\_\_file\_\_).resolve().parent.parent

TEMPLATE\_DIR=BASE\_DIR/"templates"

INSTALLED\_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    'app1'

]

TEMPLATES = [

    {

        'BACKEND': 'django.template.backends.django.DjangoTemplates',

        'DIRS': [TEMPLATE\_DIR],

**{%url%} tag**

**{%url%} tag return url of the view.**

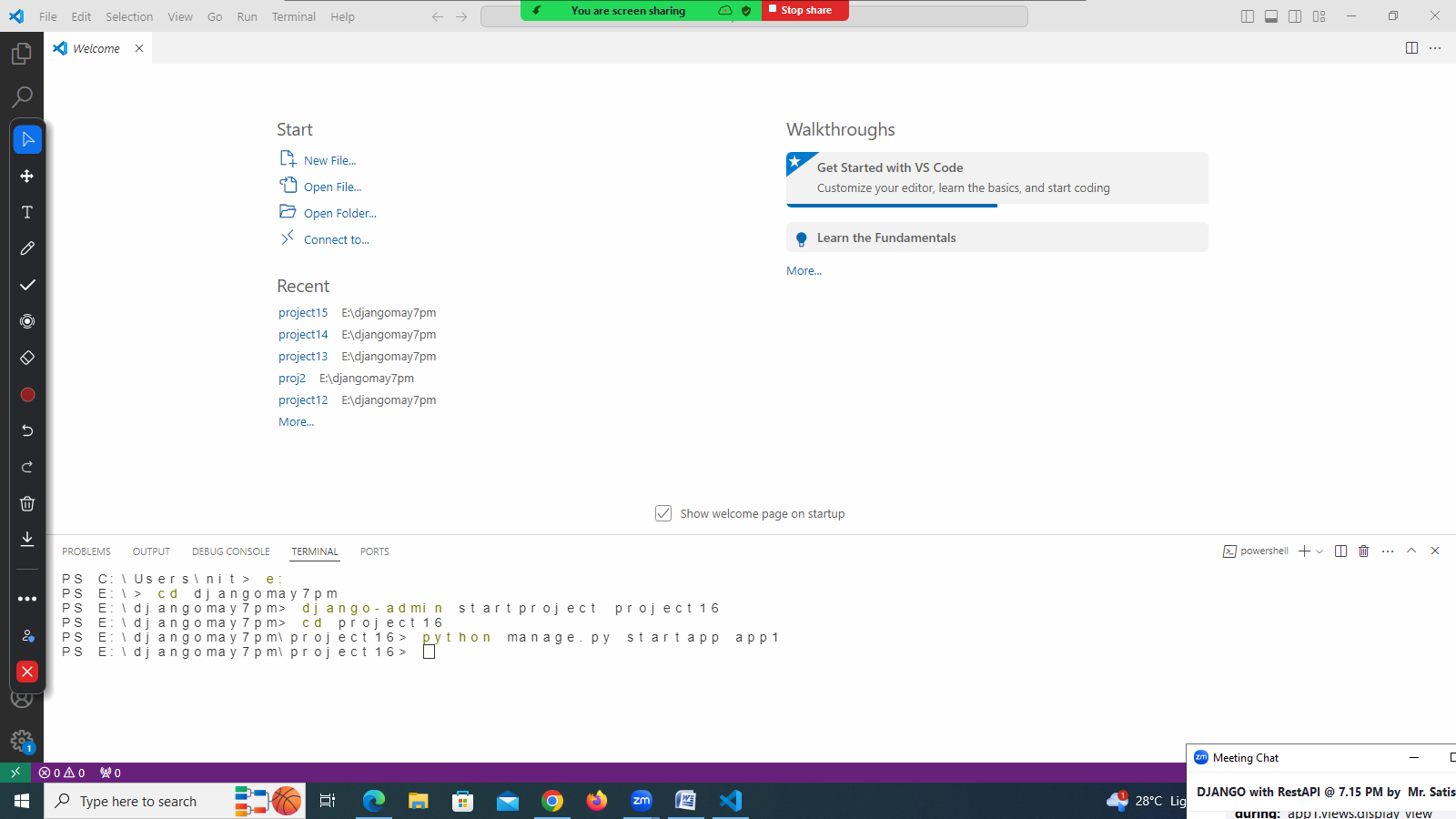
**{%url%} tag is used inside template** for calling or invoking view.

**Syntax:**

**{%url “url-name” %}**

This name of url defined inside url-configuration file (urls.py).

**Create Project**



Create templates folder inside project container folder

Create folder with application name inside templates

**display.html**

<html>

<body>

<a href="{%url 'view1' %}">View1</a><br>

<a href="{%url 'view2' %}">View2</a><br>

</body>

</html>

**Views.py**

def view1(request):

    output="<h1>This is view1 </h1>"

    response=HttpResponse(output)

    return  response

def view2(request):

    output="<h1> This is view2 </h1>"

    response=HttpResponse(output)

    return response

def view3(request):

    return render(request,"app1/display.html")

**urls.py**

urlpatterns = [

    path('admin/', admin.site.urls),

    path('view1/',view1,name='view1'),

    path('view2/',view2,name='view2'),

    path('view3/',view3)

]

**Settings.py**

BASE\_DIR = Path(\_\_file\_\_).resolve().parent.parent

TEMPATE\_DIR=BASE\_DIR/'templates'

INSTALLED\_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

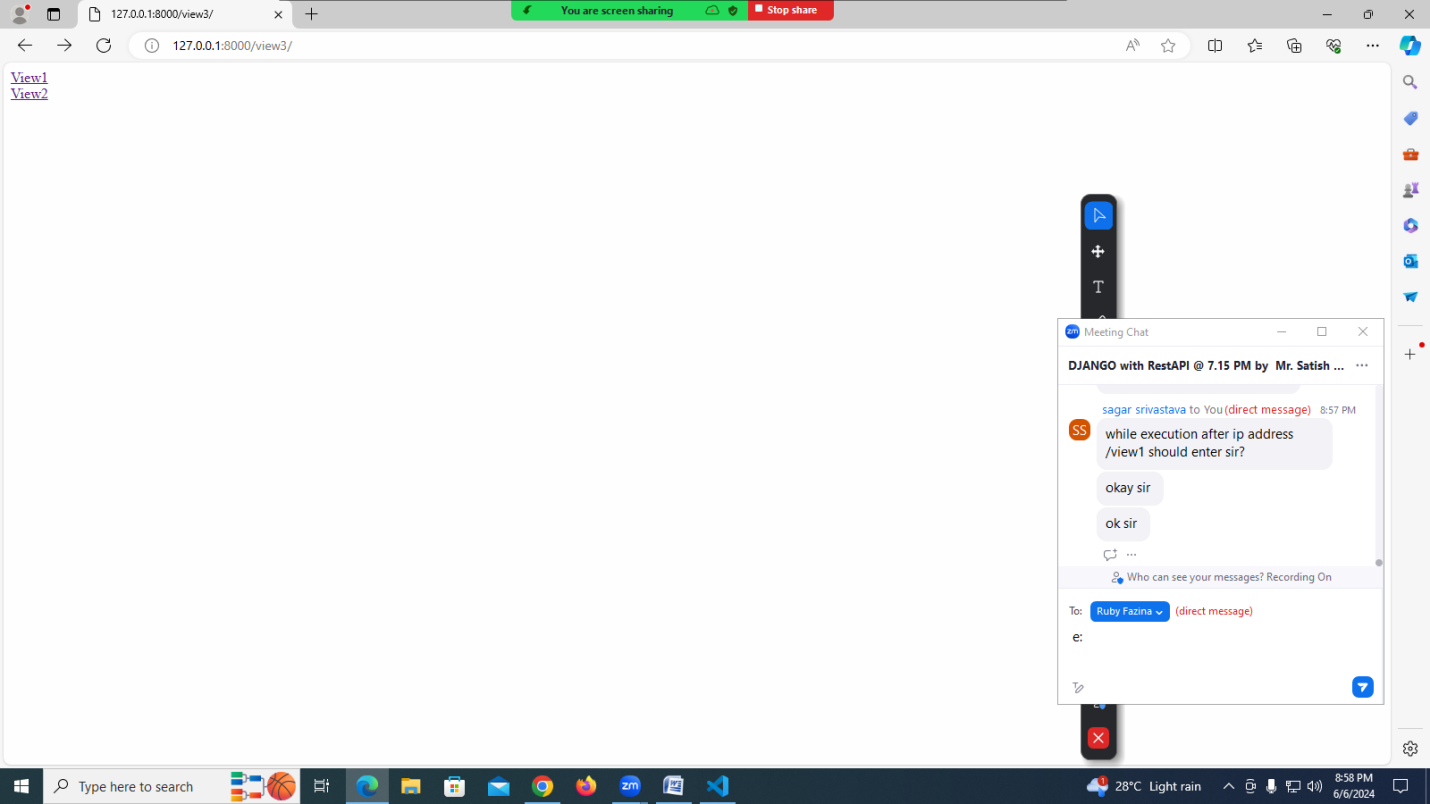
    'app1']

TEMPLATES = [

    {

        'BACKEND': 'django.template.backends.django.DjangoTemplates',

        'DIRS': [TEMPATE\_DIR],



[pythonbygupta@gmail.com](mailto:pythonbygupta@gmail.com)

codewithsatishgupta

07-june-2024

**Template Inheritance**

Template inheritance allows acquiring the properties and behavior of one template inside another template.

Template inheritance provides,

1. Reusability
2. Extensibility

Template inheritance the following django template tags

1. {%extends %}
2. {%block%} {%endblock%}
3. {%block.super%}

**{%extends%}**

This tag is used inside template to inherit another template.

**Syntax: {%extends template-name%}**

**{%block name%}…{%endblock%}**

This tag defines block inside base template, this block can be modified or override inside child template or derived template.

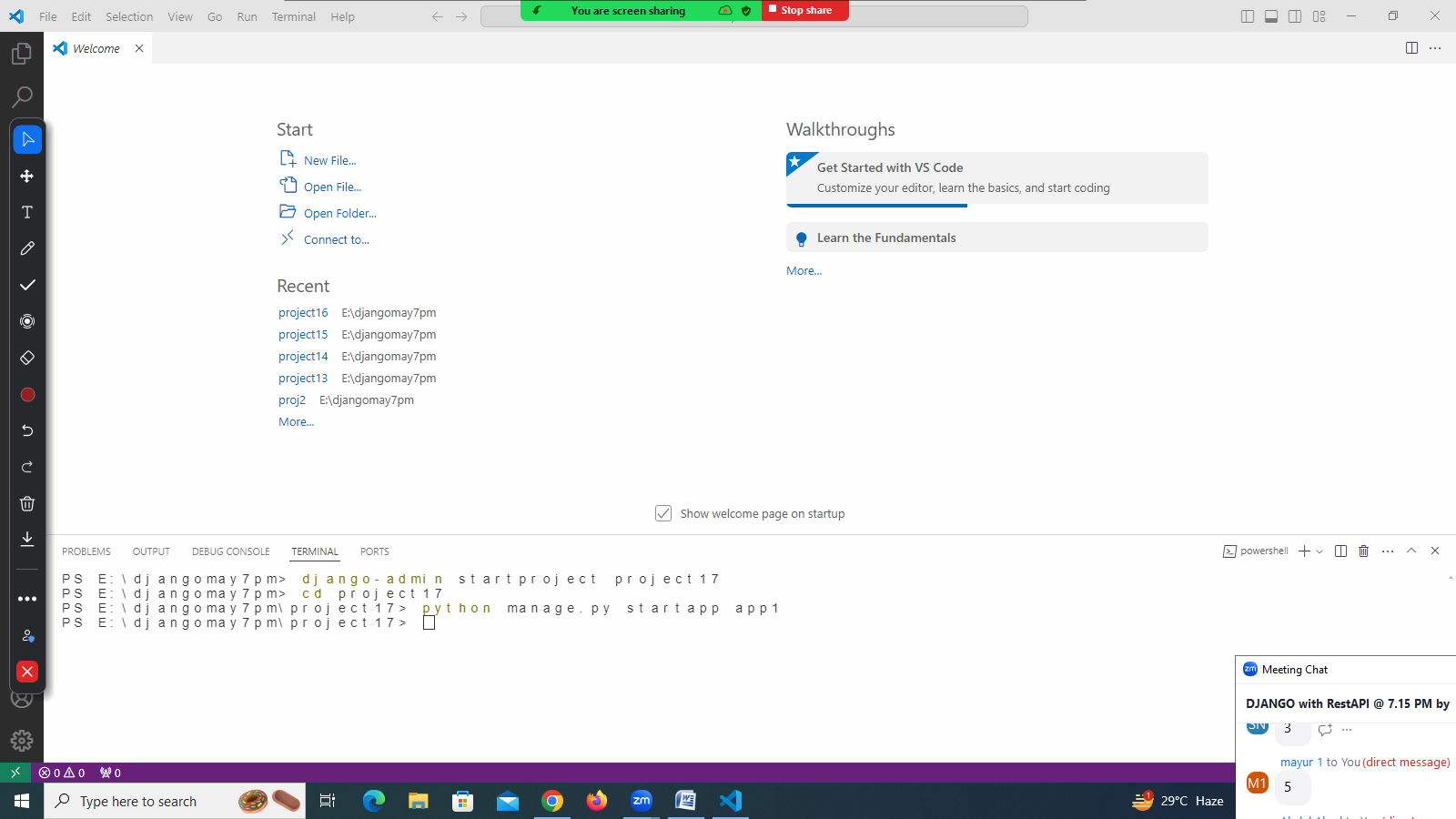
This block tag defines the structure of child template.

The content of child template is defined using block inside base template.

**{{block.super}}**

This variable includes content of base template block within derived template or child template.

**Create Project**

****

**Create templates folder inside project container folder**

**Create application folder within template folder**

**base.html**

<html>

<head>

    {%block title%}<title>Base Template</title>{%endblock%}

</head>

<body>

    <header>

        <nav>

            <ul>

                <li><a href="#"> Home </a> </li>

                <li><a href="#"> Content </a></li>

                <li><a href="#"> ContactUs </a></li>

            </ul>

        </nav>

    </header>

    {%block content%}Base Template{% endblock %}

</body>

</html>

**Child1.html**

{% extends "app1/base.html" %}

{%block title%}<title>Child Template1 </title>{%endblock%}

{%block content%} <h1> Child Template1 </h1>

{{block.super}}

{%endblock%}

**Child2.html**

{% extends "app1/base.html" %}

{%block title%}<title>Child Template2 </title> {%endblock%}

{%block content%}<h2> This Child Template2 </h2>{%endblock%}

**Views.py**

def display\_view(request,n):

    if n==1:

        return render(request,"app1/child1.html")

    elif n==2:

        return render(request,"app1/child2.html")

urls.py

from django.contrib import admin

from django.urls import path

from app1.views import display\_view

urlpatterns = [

    path('admin/', admin.site.urls),

    path('display/<int:n>/',display\_view)

]

Settings.py

TEMPLATE\_DIR=BASE\_DIR/"templates"

INSTALLED\_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    'app1'

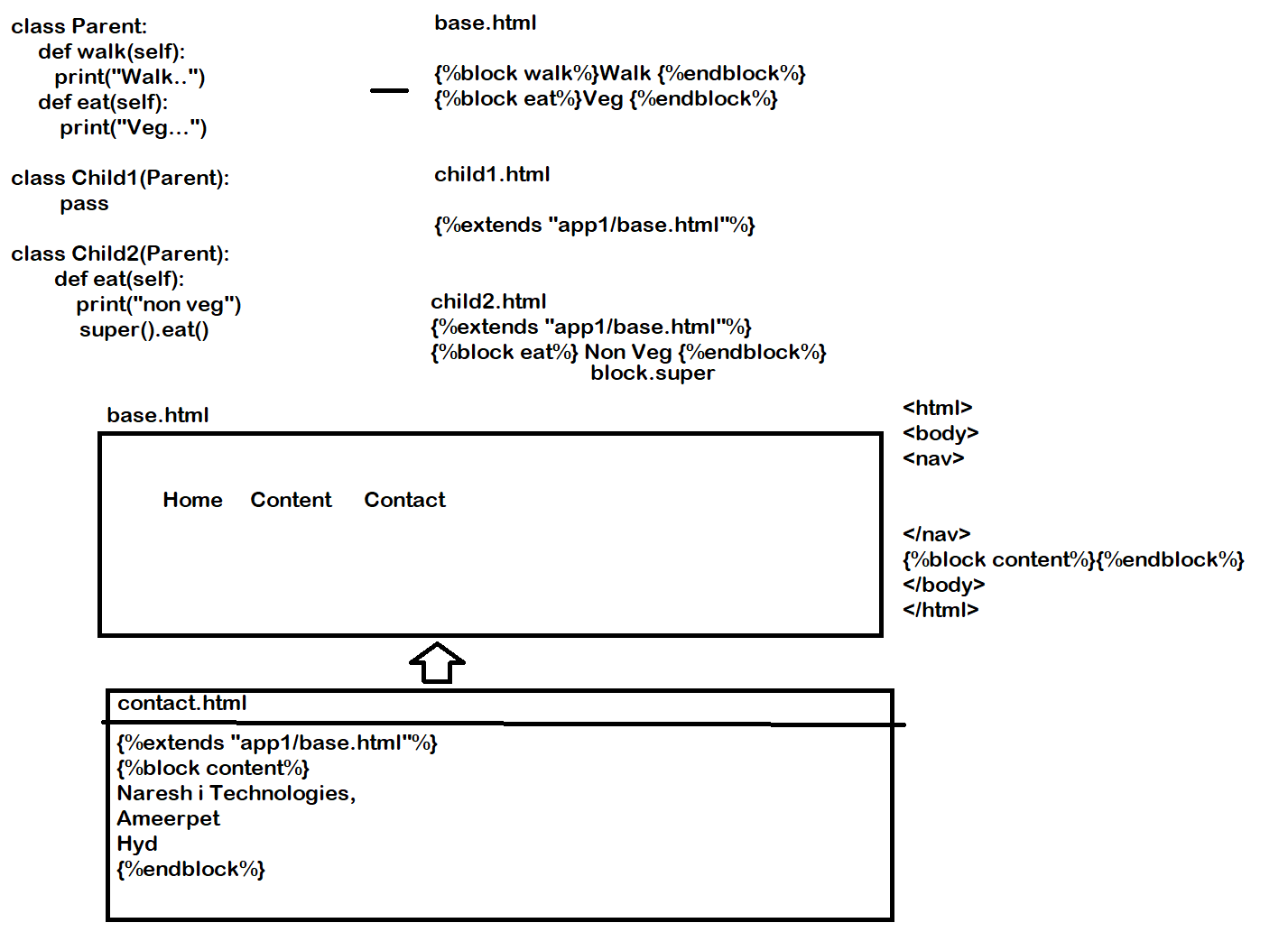
]

TEMPLATES = [

    {

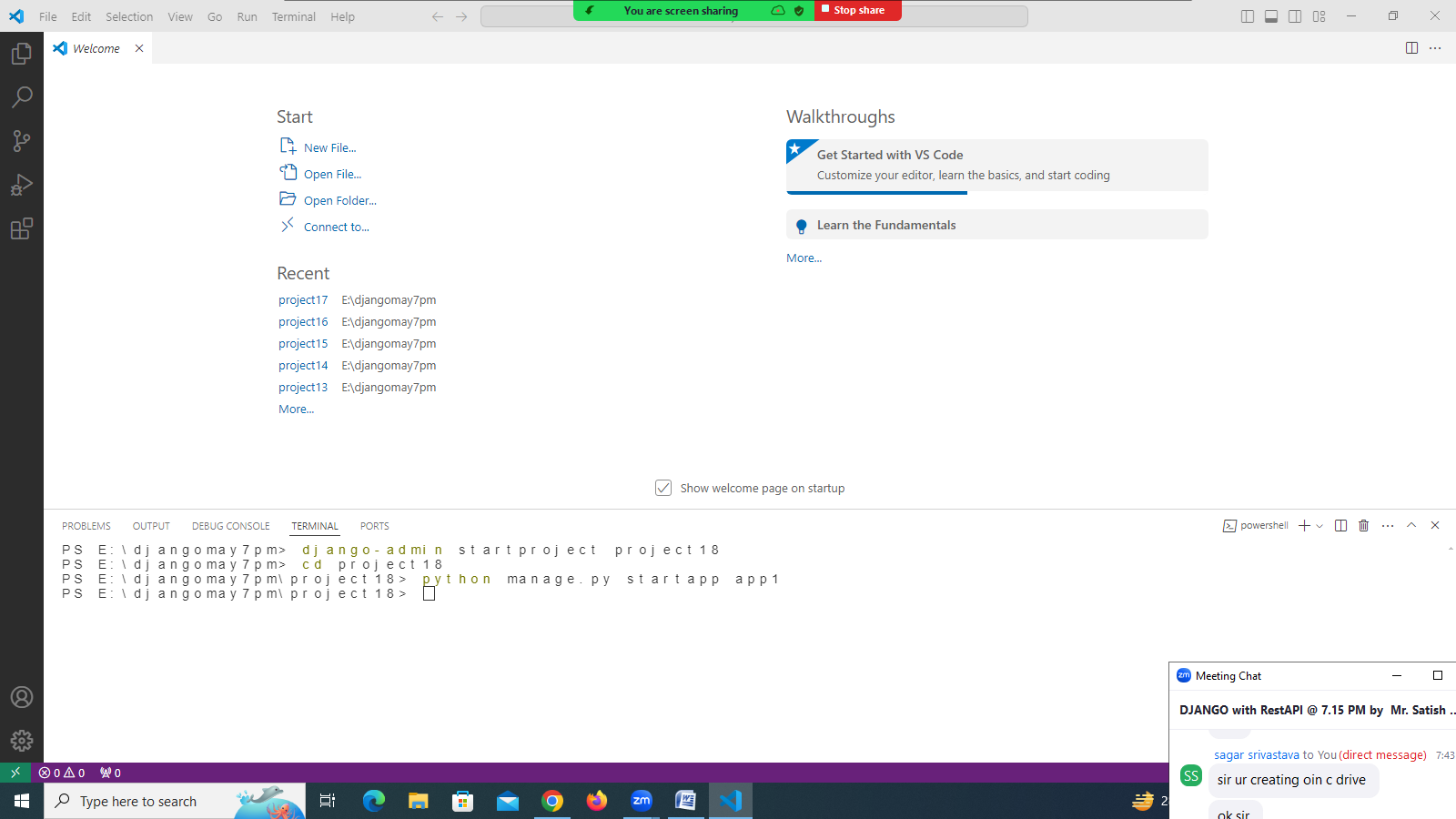
        'BACKEND': 'django.template.backends.django.DjangoTemplates',

        'DIRS': [TEMPLATE\_DIR],



08-june-2024

**Case Study (E-Commerce Applications)**



Index.html

<!DOCTYPE html>

<html lang="en">

<head>

<title>Online Shoping Site in India</title>

<style>

    body {

        margin: 0;

        font-family: Arial, sans-serif;

      }

      nav {

        background-color: #333;

        overflow: hidden;

      }

      nav a {

        float: left;

        display: block;

        color: white;

        text-align: center;

        padding: 14px 20px;

        text-decoration: none;

      }

      nav a:hover {

        background-color: #ddd;

        color: black;

      }

      table {

        border-collapse: collapse;

        width: 100%;

      }

      th, td {

        border: 1px solid #dddddd;

        padding: 8px;

        text-align: left;

      }

      th {

        background-color: #f2f2f2;

      }

      tr:nth-child(even) {

        background-color: #f2f2f2;

      }

</style>

</head>

<body>

<h2>Amazon.in</h2>

<nav>

  <a href="{% url "mobiles" %}">Mobiles</a>

  <a href="{% url "fashions" %}">Fashions</a>

  <a href="{% url "electronics" %}">Electronics</a>

  <a href="{% url "computers" %}">Computers</a>

</nav>

{%block content%}{% endblock %}

</body>

</html>

**Computers.html**

{% extends "app1/index.html" %}

{%block content%}

<table>

     <tr>

        <th>Device Name </th>

        <th> Price </th>

    </tr>

    {%for name,price in computers\_data.items %}

    <tr>

        <td>{{name}}</td>

        <td>{{price}}</td>

    </tr>

    {%endfor%}

</table>

{%endblock%}

**Fashions.html**

{% extends "app1/index.html" %}

{%block content%}

<table>

     <tr>

        <th>Product Name</th>

        <th> Price </th>

    </tr>

    {%for name,price in fashions\_data.items %}

    <tr>

        <td>{{name}}</td>

        <td>{{price}}</td>

    </tr>

    {%endfor%}

</table>

{%endblock%}

**Electronics.html**

{% extends "app1/index.html" %}

{%block content%}

<table>

     <tr>

        <th>Product Name </th>

        <th> Price </th>

    </tr>

    {%for name,price in electronics\_data.items %}

    <tr>

        <td>{{name}}</td>

        <td>{{price}}</td>

    </tr>

    {%endfor%}

</table>

{%endblock%}

**Mobiles.html**

{% extends "app1/index.html" %}

{%block content%}

<table>

     <tr>

        <th>Mobile Name </th>

        <th> Price </th>

    </tr>

    {%for mname,price in mobiles\_data.items %}

    <tr>

        <td>{{mname}}</td>

        <td>{{price}}</td>

    </tr>

    {%endfor%}

</table>

{%endblock%}

Urls.py

"""

URL configuration for project18 project.

from django.contrib import admin

from django.urls import path

from app1.views import \*

urlpatterns = [

    path('admin/', admin.site.urls),

    path('index/',index),

    path('mobiles/',mobiles,name="mobiles"),

    path('electronics/',electronics,name="electronics"),

    path('fashions/',fashions,name='fashions'),

    path('computers/',computers,name='computers')

]

Settings.py

BASE\_DIR = Path(\_\_file\_\_).resolve().parent.parent

TEMPLATES\_DIR=BASE\_DIR/"templates"

INSTALLED\_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    'app1'

]

TEMPLATES = [

    {

        'BACKEND': 'django.template.backends.django.DjangoTemplates',

        'DIRS': [TEMPLATES\_DIR],

        'APP\_DIRS': True,

**Views.py**

from django.shortcuts import render

# Create your views here.

def index(request):

    return render(request,"app1/index.html")

def mobiles(request):

    mobiles\_data={'Oneplus Nord CE4':'₹24,999',

                  'Redmi 12 5G Jade Black':'₹12,499',

                  'iQOO Z9x 5G':'₹12,999',

                  'realme GT 6T ':'₹32,999',

                  'Samsung Galaxy M15':'₹12,999'}

    return render(request,'app1/mobiles.html',context={'mobiles\_data':mobiles\_data})

def fashions(request):

    fashions\_data={'TIMEWEAR':'₹289',

                   'SELLORIA':'₹185',

                   'LOUIS DEVIN':'₹369 ',

                  'Fire-Boltt Ninja Call Pro Plus ':' ₹1,099' }

    return render(request,"app1/fashions.html",context={"fashions\_data":fashions\_data})

def electronics(request):

    electronics\_data={

            'GoPro HERO10 Black': '₹27,490',

            'Fujifilm Instax Mini 12 Happiness':'₹8,999',

            'Sony Alpha ILCE-6100Y 24.2':'₹79,999',

            'Sony Alpha ZV-E10L 24.2':'₹61,489' }

    return render(request,'app1/electronics.html',context={'electronics\_data':electronics\_data})

def computers(request):

    computers\_data={

            'Logitech K480 Wireless Multi-Device Keyboard':'₹1,999',

            'iClever BK10 Multi-Device Bluetooth Keyboard':'₹2,599',

            'Logitech Mx Mechanical Wireless':' ₹17,995',

            'Amazon Basics Wireless':'₹1,199'}

    return render(request,'app1/computers.html',context={'computers\_data':computers\_data})

**Static files**

1. june-2024

**Static files or static resources**

Web application is collection of resources.

1. Static resources
   1. Images
   2. CSS files
   3. Java Scripts
   4. Audio
   5. Videos
2. Dynamic resources
   1. Template (Text File .html)
   2. Views (.py)

**Basic steps for working with static files**

1. Inside project root folder or container folder
2. Create folder with name “static”
3. Create folder with name “images” inside static folder
4. Inside this folder copy images

**Create project**

e:\djangomay7pm>django-admin startporject project19

e:\djangomay7pm>cd project19

e:\djangomay7pm\project19> python manage.py startapp app1

Open project within vscode

1. Inside project root folder or container folder
2. Create folder with name “static”
3. Create folder with name “images” inside static folder
4. Inside this folder copy images

Configure static resources within settings.py

**Settings.py**

BASE\_DIR = Path(\_\_file\_\_).resolve().parent.parent

STATIC\_PATH=BASE\_DIR/"static"

Create templates folder configure in settings.py

TEMPLATE\_DIR=BASE\_DIR/"templates"

STATIC\_URL = 'static/'

STATICFILES\_DIRS=[STATIC\_PATH]

Static files are managed by an application called django.contrib.staticfiles application

**test.html**

<html>

{%load static%}

<head>

    <title> Static Resources Demo </title>

</head>

<body>

<h2>Python</h2>

<img src="{%static "images/python.jpeg" %}">

</body>

</html>

**Views.py**

from django.shortcuts import render

# Create your views here.

def display\_image(request):

    return render(request,"app1/test.html")

**urls.py**

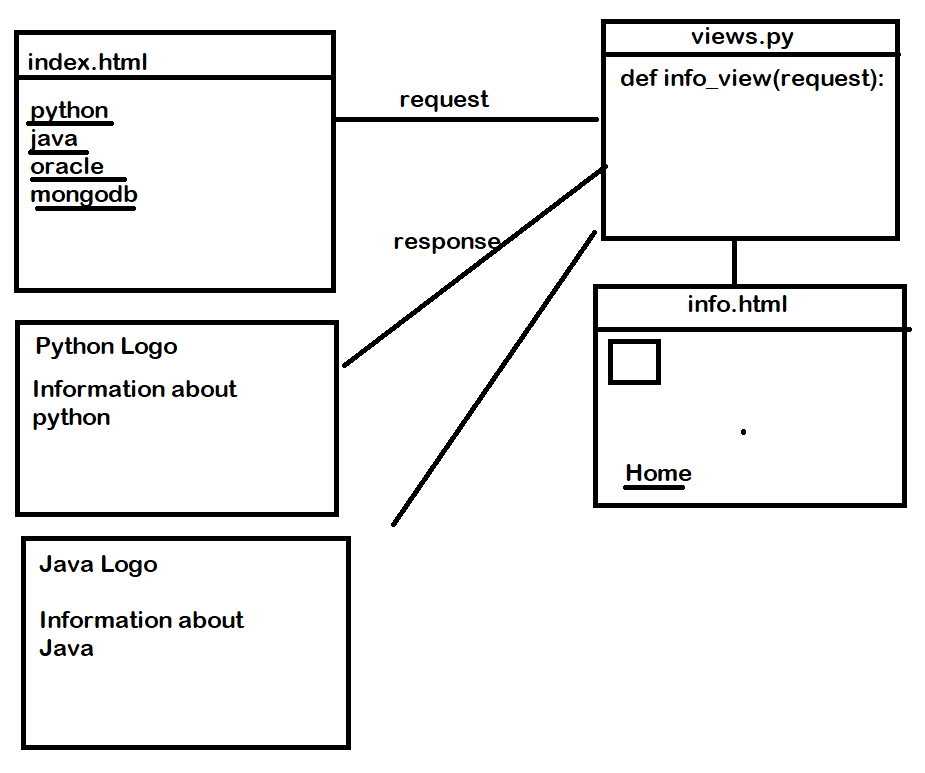
from app1.views import display\_image

urlpatterns = [

    path('admin/', admin.site.urls),

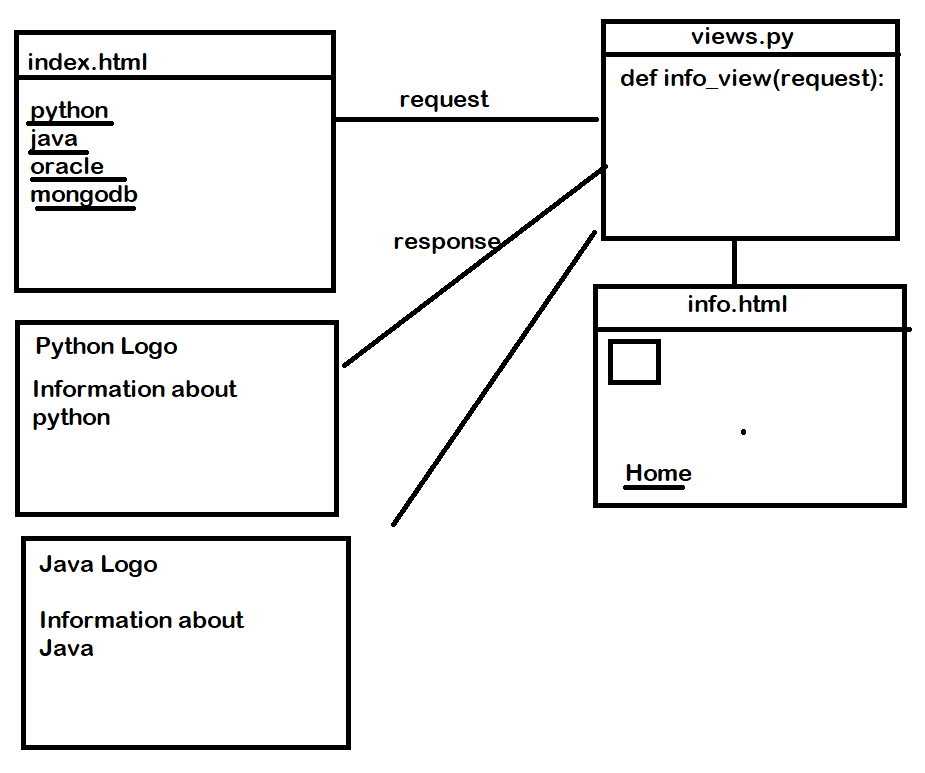
    path('display/',display\_image)

]

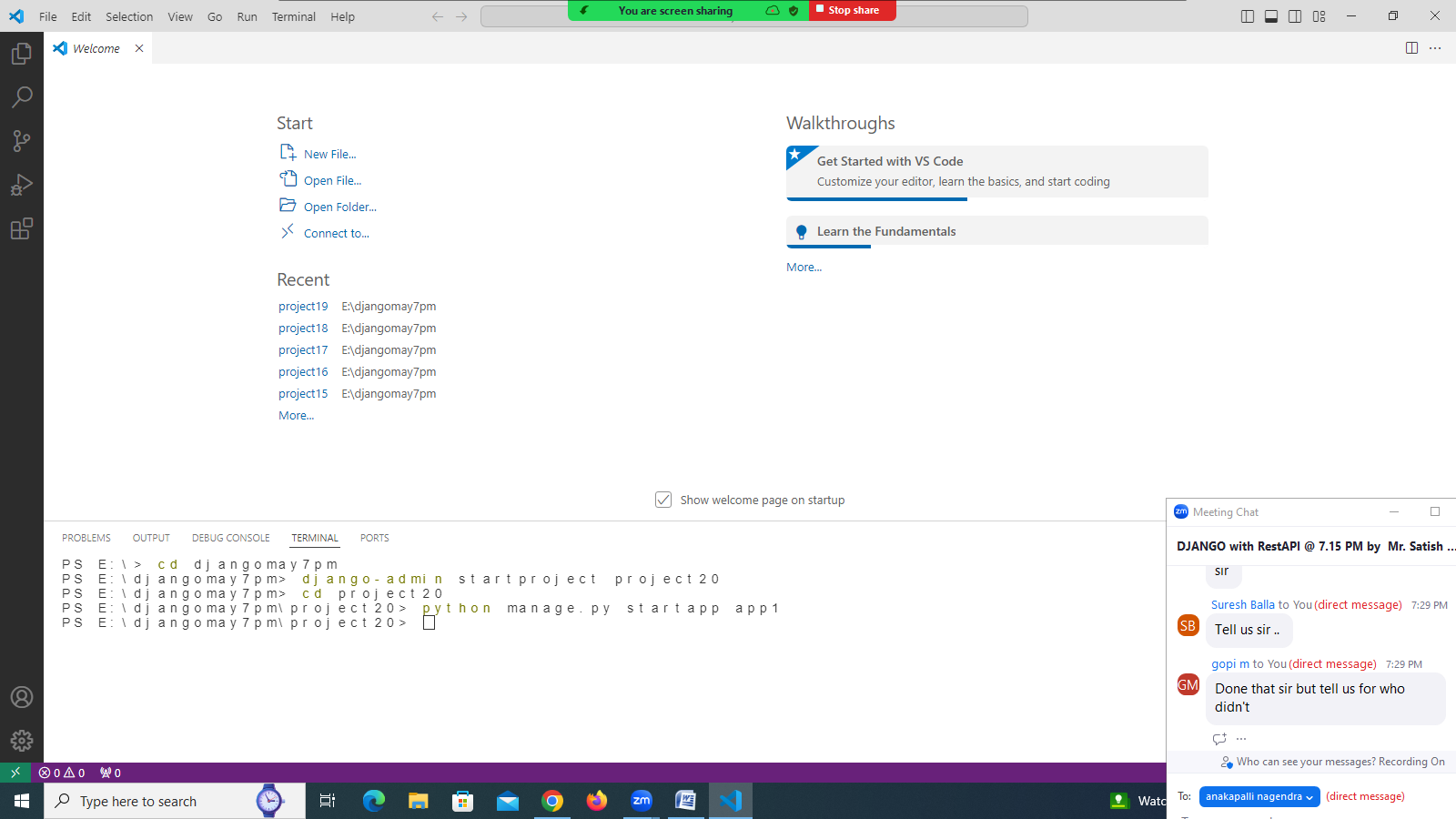


**CSS files**

1. june-2023



1. Create Project
2. Inside Project create Application



1. Create static folder inside project root folder/container folder
2. Inside static folder create images folder and copy images
3. Create templates folder inside project root folder
4. Crate application folder inside templates folder

**Index.html**

<html>

<head>

    <title>Index</title>

</head>

<body>

<a href="{%url "info" "python"%}">Python</a><br>

<a href="{%url "info" "java" %}">Java</a><br>

<a href="{%url "info" "oracle"%}">Oracle</a><br>

</body>

</html>

**Urls.py**

from django.contrib import admin

from django.urls import path

from app1.views import \*

urlpatterns = [

    path('admin/', admin.site.urls),

    path('index/',display\_index,name="index"),

    path('info/<course>/',info\_view,name="info")

]

**Settings.py**

TEMPLATE\_DIR=BASE\_DIR/"templates"

STATIC\_DIR=BASE\_DIR/"static"

INSTALLED\_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    'app1'

]

TEMPLATES = [

    {

        'BACKEND': 'django.template.backends.django.DjangoTemplates',

        'DIRS': [TEMPLATE\_DIR],

STATIC\_URL = 'static/'

STATICFILES\_DIRS=[STATIC\_DIR]

**Views.py**

def display\_index(request):

    return render(request,"app1/index.html")

def info\_view(request,course):

    match(course):

        case "python":

            desc='''Python is generaal purpose programming langauge

            This langauge is used for developing various types of applications

            Python is easy language'''

            data={'desc':desc,'course':course}

            return render(request,"app1/info.html",context=data)

        case "java":

            desc='''Java is an object oriented programming langauge

            This langauge is platform indepedent and protable langague

            This language is used for develop mobile and web applications'''

            data={'desc':desc,'course':course}

            return render(request,"app1/info.html",context=data)

        case "oracle":

            desc='''Oracle is large database, which allows to store large amount

            data, this database support cloud and AI features,

            Versions of oracle is 21ai'''

            data={'desc':desc,'course':course}

            return render(request,"app1/info.html",context=data)

**info.html**

<HTML>

<head>

    <title>Courses Information</title>

    {%load static%}

</head>

<body>

{%if course == "python" %}

<img src="{% static "images/python.jpeg" %}">

<h2>{{desc}} </h2>

{% elif course == "java" %}

<img src="{% static "images/java.jpeg" %}">

<h2>{{desc}}</h2>

{%elif course == "oracle" %}

<img src="{% static "images/oracle.jpeg" %}">

<h2>{{desc}}</h2>

{% endif %}

<br>

<a href="{%url "index" %}">Home</a>

</body>

</html>

**css files (external css file)**

It is similar to images.

Inside static folder create css folder.

Add css file (OR) create on css file

**Style1.css**

body{

    background-color: aqua;

    text-align: center;

}

h2{

    color: blue;

}

img{

    width: 200px;

    height: 200px;

    border: 2px;

}

**Info.html**

<HTML>

<head>

    <title>Courses Information</title>

    {%load static%}

    <link rel="stylesheet" href="{%static "css/style1.css" %}">

</head>

<body>

{%if course == "python" %}

<img src="{% static "images/python.jpeg" %}">

<h2>{{desc}} </h2>

{% elif course == "java" %}

<img src="{% static "images/java.jpeg" %}">

<h2>{{desc}}</h2>

{%elif course == "oracle" %}

<img src="{% static "images/oracle.jpeg" %}">

<h2>{{desc}}</h2>

{% endif %}

<br>

<a href="{%url "index" %}">Home</a>

</body>

</html>

**How to send data to view?**

1. url parameter
2. request parameter
3. using html form (UI)

**URL parameter**

URL parameter is defined inside urls.py with URL pattern

Syntax: “url-pattern/param1/param2/”

Syntax: “url-pattern/<datatype:param1>/”

display/10/20

def view\_fun(request,param1,param2):

statement

**Request Parameters OR Query String**

Request parameters are concatenated with URL while sending request.

Syntax: inside browser url?<param-name>=value&<param-name>=<value>

The values which are concatenated with URL are called query string. These values are stored inside request object and send view.

Request parameters are strings.

HTTP Request Methods

1. GET
2. POST

Default request method is GET

In request method GET, the data is send by concatenating with URL.

def view1(request):

request.GET.get(“parameter-name”)

request.GET.get(“parameter-name”)

1. june-2024

**Request Parameters OR Query String**

Request parameters are concatenated with URL while sending request.

Syntax: inside browser url?<param-name>=value&<param-name>=<value>

The values which are concatenated with URL are called query string. These values are stored inside request object and send view.

Request parameters are strings.

HTTP Request Methods

1. GET
2. POST

Default request method is GET

In request method GET, the data is send by concatenating with URL.

def view1(request):

request.GET.get(“parameter-name”)

request.GET.get(“parameter-name”)

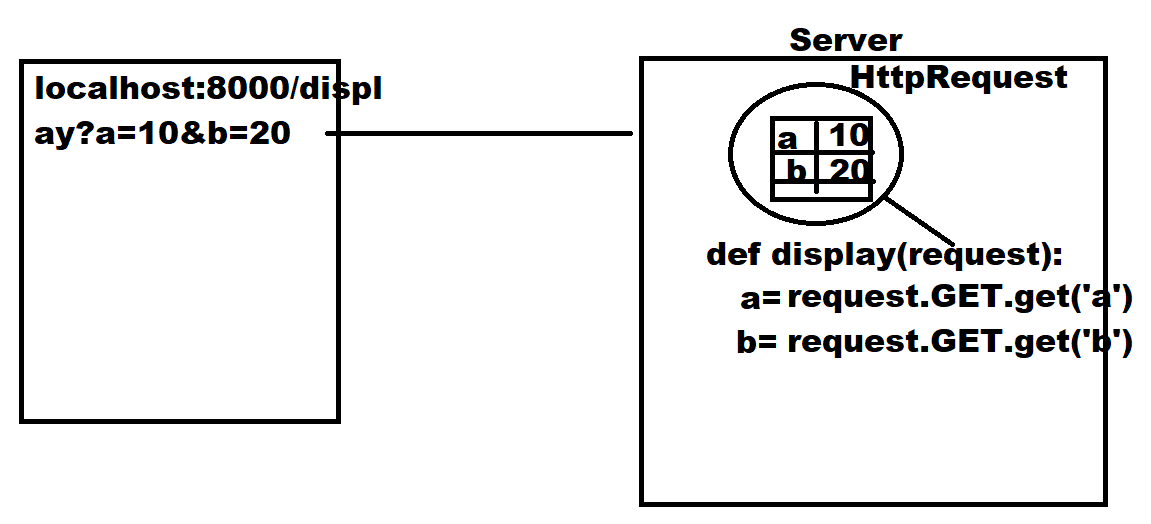
1. Create Project
2. Create application within project

E:\\djangomay7pm>django-admin startproject project21

E:\\ djangomay7pm>cd project21

E:\\ djangomay7pm\project21>python manage.py startapp app1

Open this project in vscode.



**Views.py**

from django.shortcuts import render

from django.http import HttpResponse

# Create your views here.

def display(request):

    a=request.GET.get('a')

    b=request.GET.get('b')

    data=f'<h1>the value of a={a} and b={b}  </h1>'

    response=HttpResponse(data)

    return response

**settings.py**

INSTALLED\_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    'app1'

]

from django.contrib import admin

from django.urls import path

from app1.views import display

urlpatterns = [

    path('admin/', admin.site.urls),

    path('display/',display)

]

**Views.py**

from django.shortcuts import render

from django.http import HttpResponse

# Create your views here.

def display(request):

    a=request.GET.get('a')

    b=request.GET.get('b')

    opr=request.GET.get('opr')

    match(opr):

        case '+':

            res=eval(a)+eval(b)

        case '-':

            res=eval(a)-eval(b)

        case '\*':

            res=eval(a)\*eval(b)

        case '/':

            res=eval(a)/eval(b)

        case \_:

            res=None

    result={'res':res}

    return render(request,"app1/result.html",context=result)

**result.html**

<html>

<head>

</head>

<body>

<h1> Result is </h1>

<h2> {{res}} </h2>

</body>

</html>

**Settings.py**

BASE\_DIR = Path(\_\_file\_\_).resolve().parent.parent

TEMPLATE\_PATH=BASE\_DIR/"templates"

TEMPLATES = [

    {

        'BACKEND': 'django.template.backends.django.DjangoTemplates',

        'DIRS': [TEMPLATE\_PATH],

**Urls.py**

from django.contrib import admin

from django.urls import path

from app1.views import display

urlpatterns = [

    path('admin/', admin.site.urls),

    path('display/',display)

]

**HTML form**

HTML form is user interface where end user/client input data and submit to backend (view).

Data can be send view or submit to view using two methods.

1. GET
2. POST

In GET request the data is send to server by concatenating with URL as query string.

Limitation of GET method

1. GET method is for reading data from server
2. GET method data is concatenated with URL as string, It is visible on URL bar. Secured information cannot send using GET method
3. Large amount of data cannot send using GET method, because of URL length
4. GET method is used for sending string.

In POST request method data is send as part of request body or stream.

POST is a request for writing data or information to server.

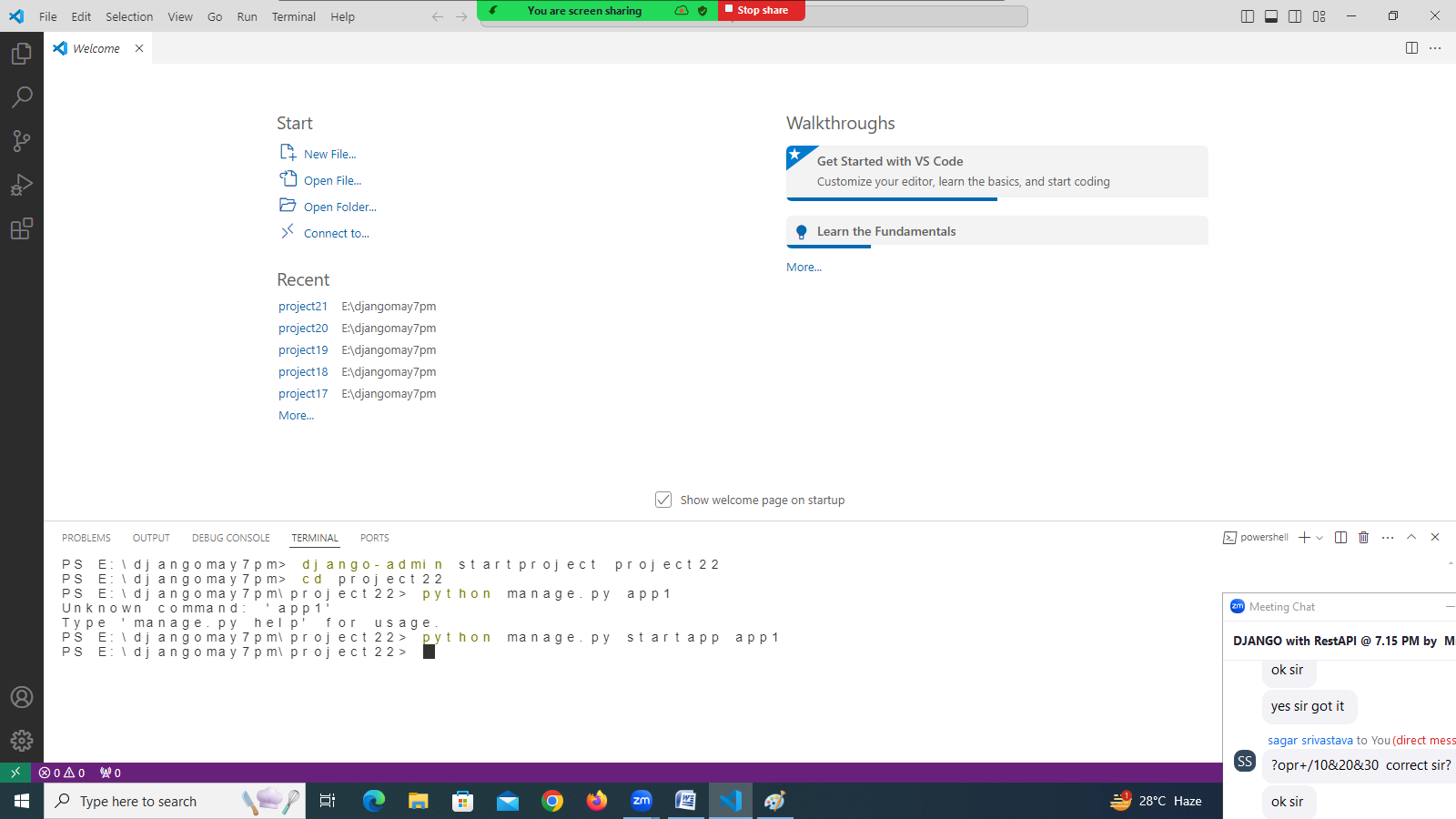
<form method=”GET/POST” action=”url”>

Form element

</form>

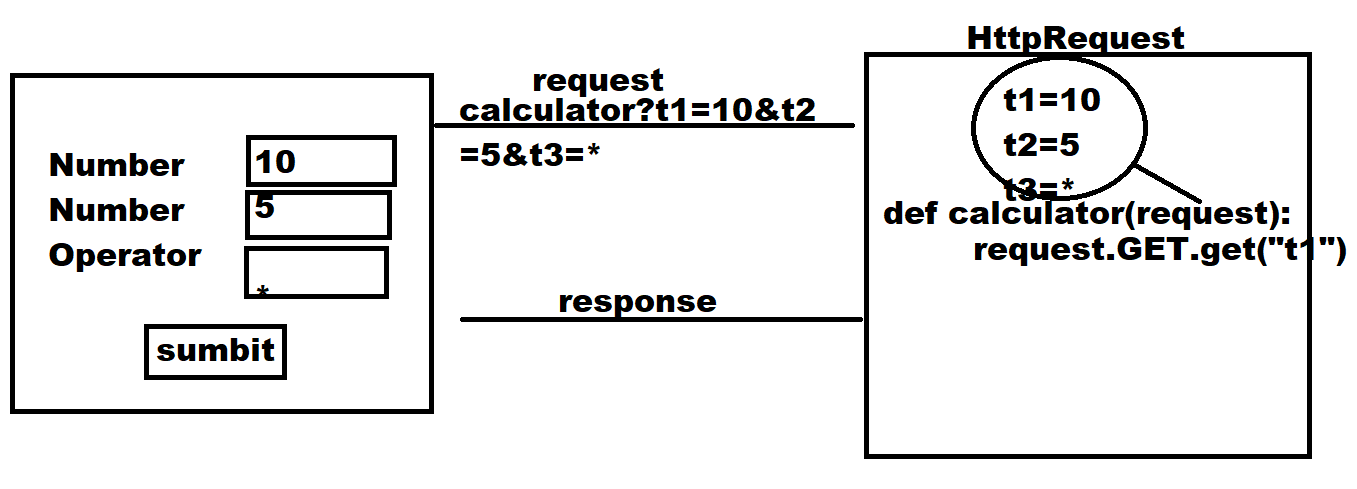
Create Project

Crate Application within project



Create templates folder within project container folder/root folder

Create application folder with templates



calc.html

<html>

<head>

    <title>Calculator</title>

</head>

<body>

<form method="GET" action="{%url "calculator" %}">

<h2>

Number <input type="text" name="t1"><br>

Number <input type="text" name="t2"><br>

Operator <input type="text" name="opr"><br>

<input type="submit" value="submit">

</h2>

</form>

</body>

</html>

**Views.py**

from django.shortcuts import render

from django.http import HttpResponse

# Create your views here.

def index(request):

    return render(request,"app1/calc.html")

def calculator(request):

    a=request.GET.get("t1")

    b=request.GET.get("t2")

    c=request.GET.get("t3")

    return HttpResponse("calculator")

urls.py

urlpatterns = [

    path('admin/', admin.site.urls),

    path("index/",index),

    path("calculator/",calculator,name="calculator")

]

**Settings.py**

INSTALLED\_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    'app1'

]

BASE\_DIR = Path(\_\_file\_\_).resolve().parent.parent

TEMPLATE\_DIR=BASE\_DIR/"templates"

TEMPLATES = [

    {

        'BACKEND': 'django.template.backends.django.DjangoTemplates',

        'DIRS': [TEMPLATE\_DIR],

codewithsatishgupta

1. june-2024

**POST Request Method**

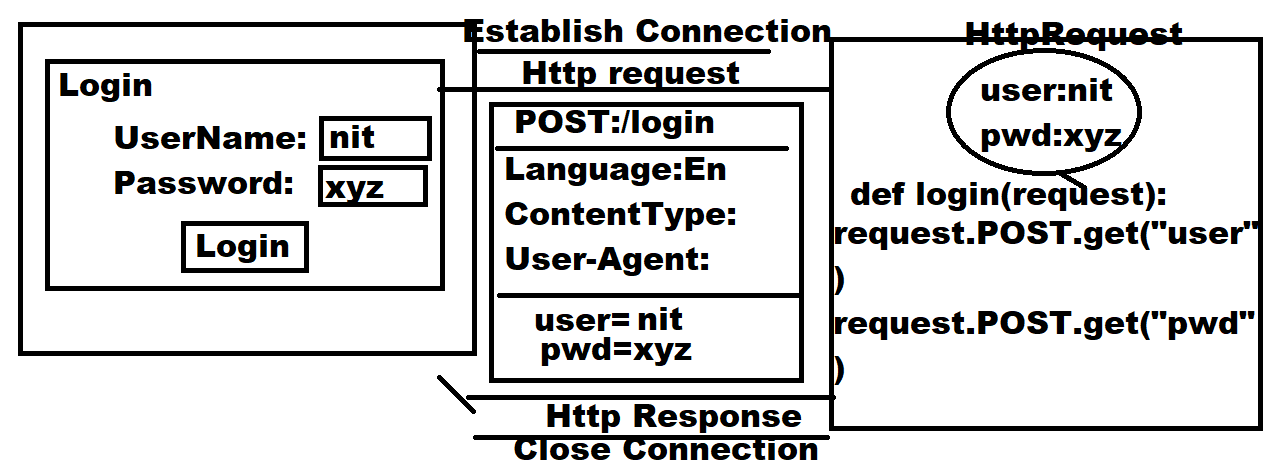
In request method post, data is send to server as part of request body or stream. It is not visible on URL bar.

Request method Post, allows sending any type of data.

The data which is send using POST method is secured.

<form method=”POST” action=”url”>

</form>



1. Create Project
2. Create Application within project
3. Create templates folder with project root folder
4. Create application folder within template folder

Login.html

<html>

<head>

<title>Login</title>

</head>

<body>

<form method="POST" action="{%url "login" %}">

    {% csrf\_token %}

<h2>

UserName <input type="text" name="user"><br>

Password <input type="password" name="pwd"><br>

<input type="submit" value="Login">

</h2>

</form>

{{msg}}

</body>

</html>

**Views.py**

from django.shortcuts import render

from django.http import HttpResponse

# Create your views here.

def login\_template(request):

    return render(request,"app1/login.html")

def login(request):

    users={'nit':'nit123',

           'naresh':'n321',

           'suresh':'s567'}

    user=request.POST.get("user")

    pwd=request.POST.get("pwd")

    if user in users and pwd==users[user]:

        msg=f'<h2>{user} welcome</h2>'

        response=HttpResponse(msg)

        return response

    else:

        msg="invalid username or password"

        return render(request,"app1/login.html",context={"msg":msg})

**urls.py**

from django.contrib import admin

from django.urls import path

from app1.views import \*

urlpatterns = [

    path('admin/', admin.site.urls),

    path('home/',login\_template),

    path('login/',login,name="login")

]

**Settings.py**

BASE\_DIR = Path(\_\_file\_\_).resolve().parent.parent

TEMPLATE\_PATH=BASE\_DIR/"templates"

INSTALLED\_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    'app1'

]

TEMPLATES = [

    {

        'BACKEND': 'django.template.backends.django.DjangoTemplates',

        'DIRS': [TEMPLATE\_PATH],

**Model**

Every web application required persistence.

Persistence is nothing but storing data permanently.

This persistence or saving can be done using two systems.

1. File System
2. Database System

**Database Applications or software’s**

1. Oracle
2. MySQL
3. PostgreSQL
4. MongoDB
5. SQLLite
6. SQL Server

How Django application communicate with database?

1. with SQL

2. without SQL (ORM)

1. june-2024

**Model**

Every web application required persistence.

Persistence is nothing but storing data permanently.

This persistence or saving can be done using two systems.

1. File System
2. Database System

**Database Applications or software’s**

1. Oracle
2. MySQL
3. PostgreSQL
4. MongoDB
5. SQLLite
6. SQL Server

How Django application communicate with database?

1. with SQL

2. without SQL (ORM)

**Limitations with SQL**

1. Programmer need to know SQL language
2. SQL Syntax is database dependent

**ORM (Object Relational Mapping)**

**ORM is tool/library or technology**

Object Relational Mapping/Mapper (ORM) is a technique used in creating a "bridge" between object-oriented programs and, in most cases, relational databases.

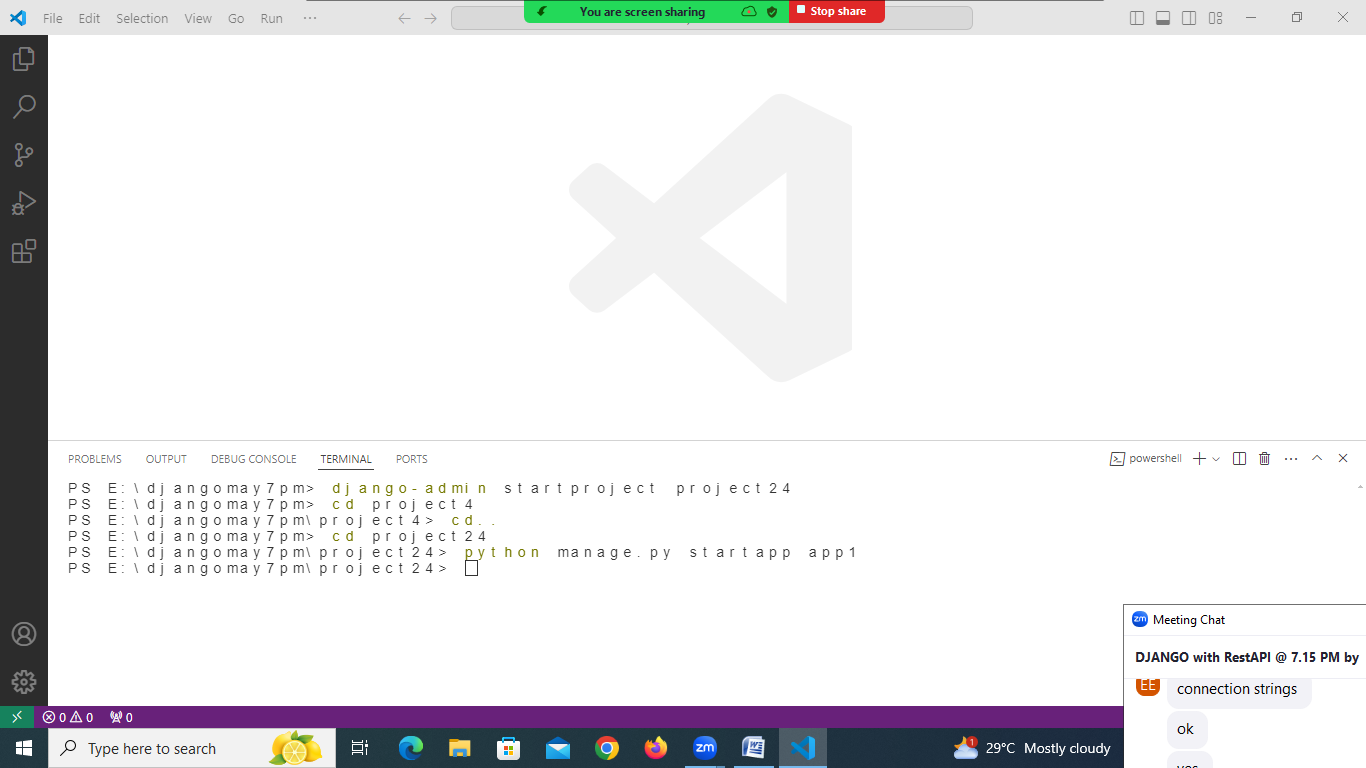
Mapping object oriented programming with relational database.

**Advantage:**

1. No SQL
2. Database independent applications (SQL Queries)

ORM tool generate queries related with object oriented programming.

Django framework comes with default ORM tool.



1. **Install Database**
   1. Django provides default database software, SQLlite
2. **Database Configuration (settings.py)**

DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.sqlite3',

        'NAME': BASE\_DIR / 'db.sqlite3',

    }

}

Database configuration is used by django for establishing connection to database.

1. ENGINE
2. NAME
3. USER
4. PASSWORD
5. HOST
6. PORTNO

**ENGINE:** It is an application or library is responsible in communicating with database.

**NAME:** Database Name

**USER:** username of the database

**PASSWORD:** password of database

**HOST:** IP/Address or host of system where database server is running.

**PRORTNO:** Database server is identified with portno

**Database configuration for SQLLite3**

DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.sqlite3',

        'NAME': BASE\_DIR / 'db.sqlite3',

    }

}

**Database configuration for Oracle**

DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.oracle',

        'NAME':'XE',

'USER':'SYSTEM',

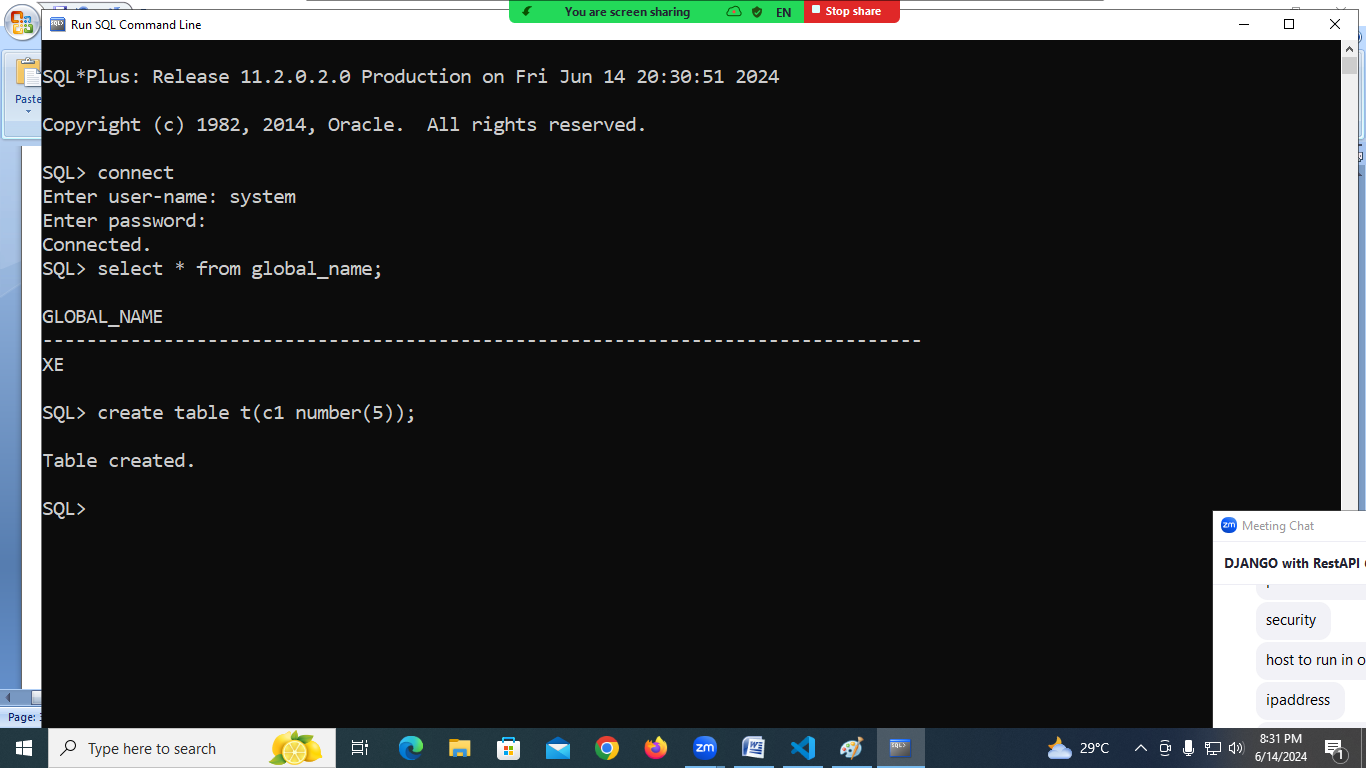
‘PASSWORD’:’MANAGER’,

‘HOST’:’localhost’,

‘PORTNO’:’1521’

    }

}



**Database configuration for Mysql**

DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.mysql',

        'NAME':'DB1',

'USER':'root',

‘PASSWORD’:’root’,

‘HOST’:’127.0.0.1’,

‘PORTNO’:’3306’

    }

}

**How to communicate with django application without writing code?**

1. Open terminal
2. PS E:\djangomay7pm\project24> python manage.py shell

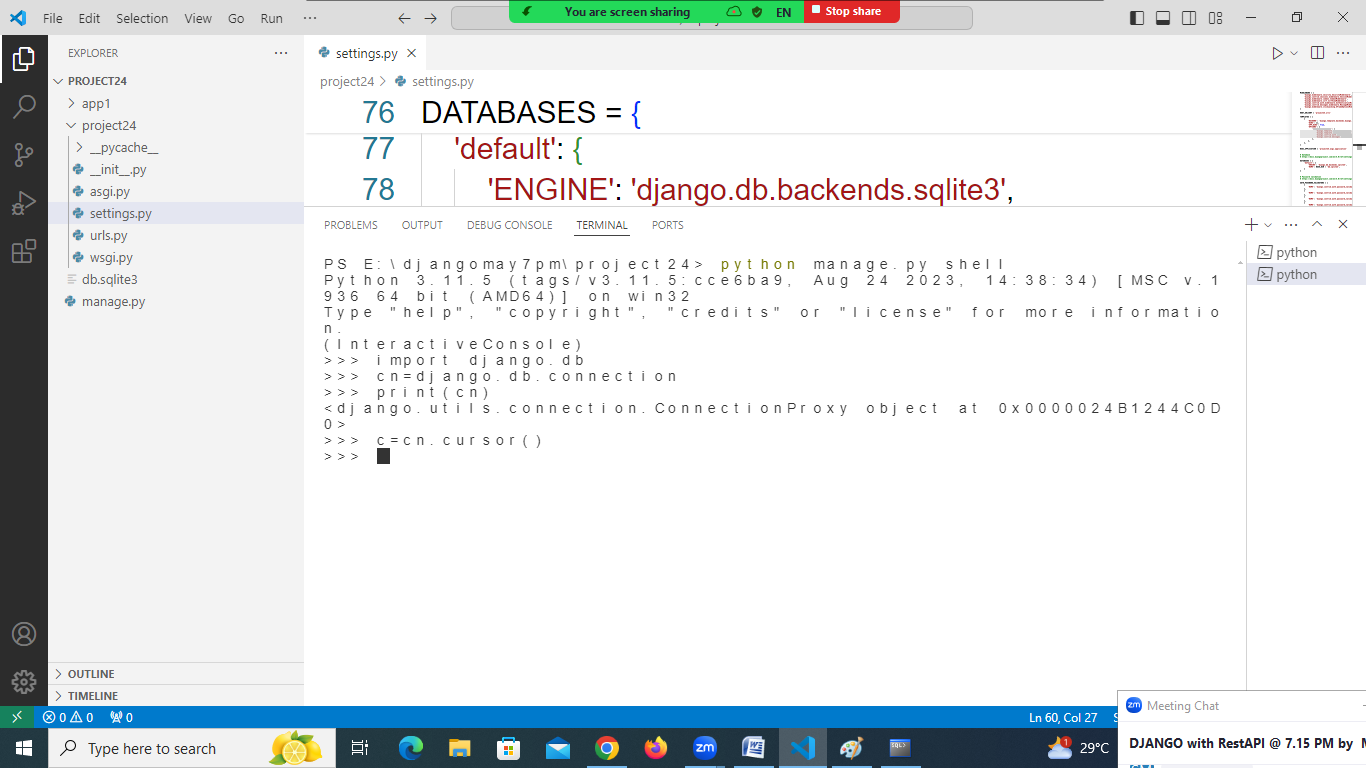
Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

(InteractiveConsole)

>>>

This interactive shell is used to communicate with django application.



1. june-2024

DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.mysql',

        'NAME':'DB1',

'USER':'root',

‘PASSWORD’:’root’,

‘HOST’:’127.0.0.1’,

‘PORT’:’3306’

    }

}

  raise ImproperlyConfigured(  
django.core.exceptions.ImproperlyConfigured: Error loading MySQLdb module.  
Did you install mysqlclient?

To avoid this error the following packages has to installed.

pip install mysqlclient

pip install PyMysql

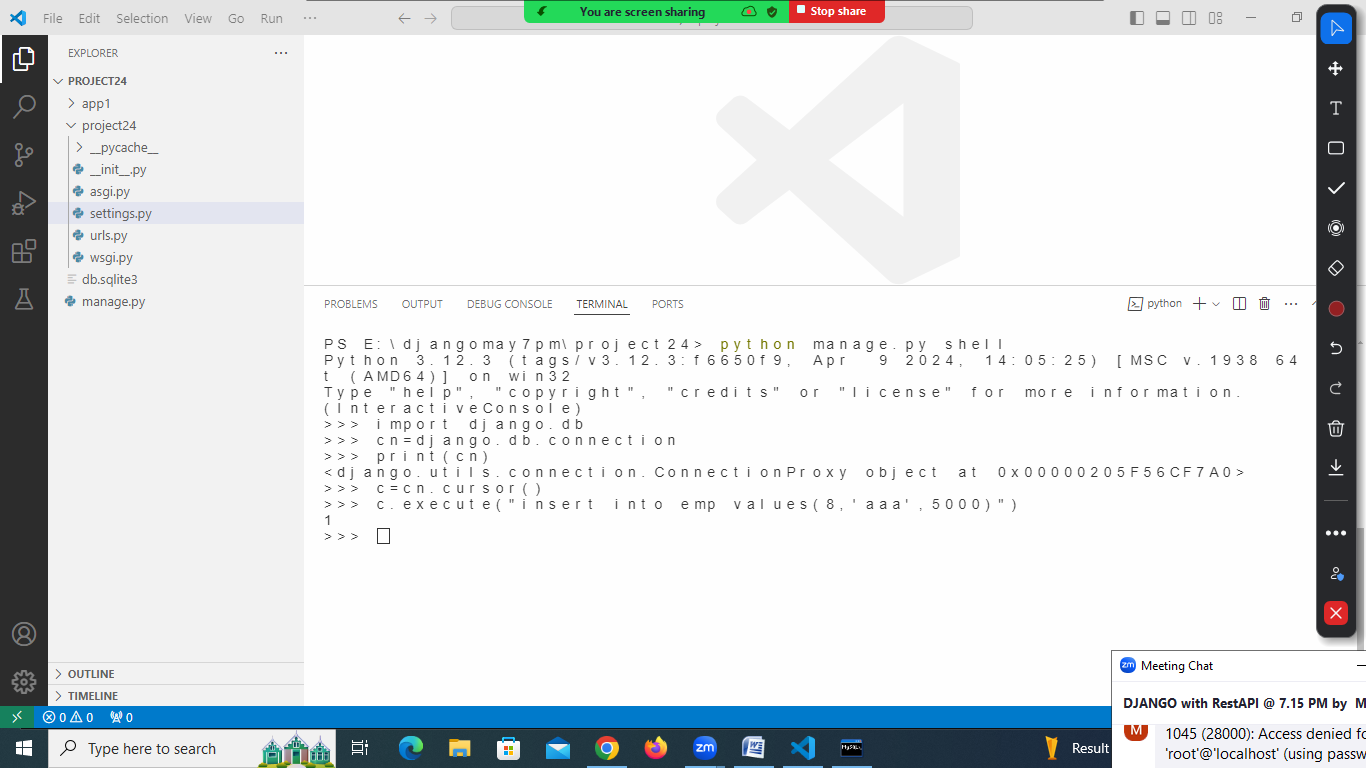
inside project folder

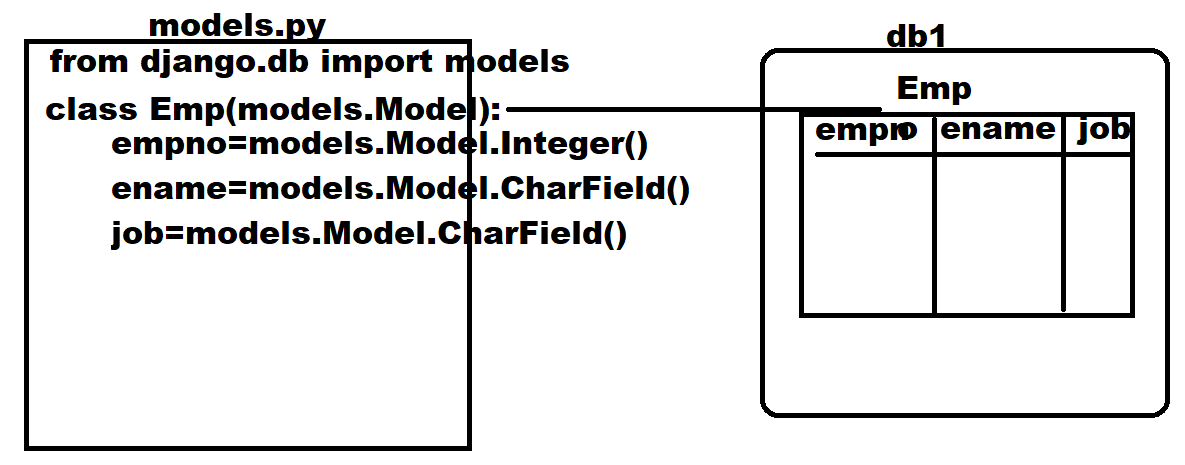
**\_\_init\_\_.py (package configuration)**

import pymysql

pymysql.install\_as\_MySQLdb()

PyMySQL is a pure-Python MySQL client library, which means it is a Python package that creates an API interface for us to access MySQL relational databases.



****

**Model**

In object relational mapping, a model represents class, which is equal to table in relational database.

Django framework provides Model class to develop model classes.

This Model class is available in django.db.models module.

Any class which inherit the properties and behavior of Model class is called model class.

This model class contains fields/variables/attributes, which represents field in database table.

**Syntax:**

class <model-class-name>(models.Model):

variables

methods

This model class is defined inside models.py (module), this module is exists in application folder.

Create Project

Create Application within project

**Settings.py**

DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.mysql',

        'NAME': 'db1',

        'USER': 'root',

        'PASSWORD': 'root',

        'HOST': 'localhost',

        'PORT': '3306',

    }

}

**Models.py**

class Student(models.Model):

    pass

PS E:\djangomay7pm\project24> python manage.py makemigrations

Migrations for 'app1':

app1\migrations\0001\_initial.py

- Create model Student

PS E:\djangomay7pm\project24> python manage.py sqlmigrate app1 0001

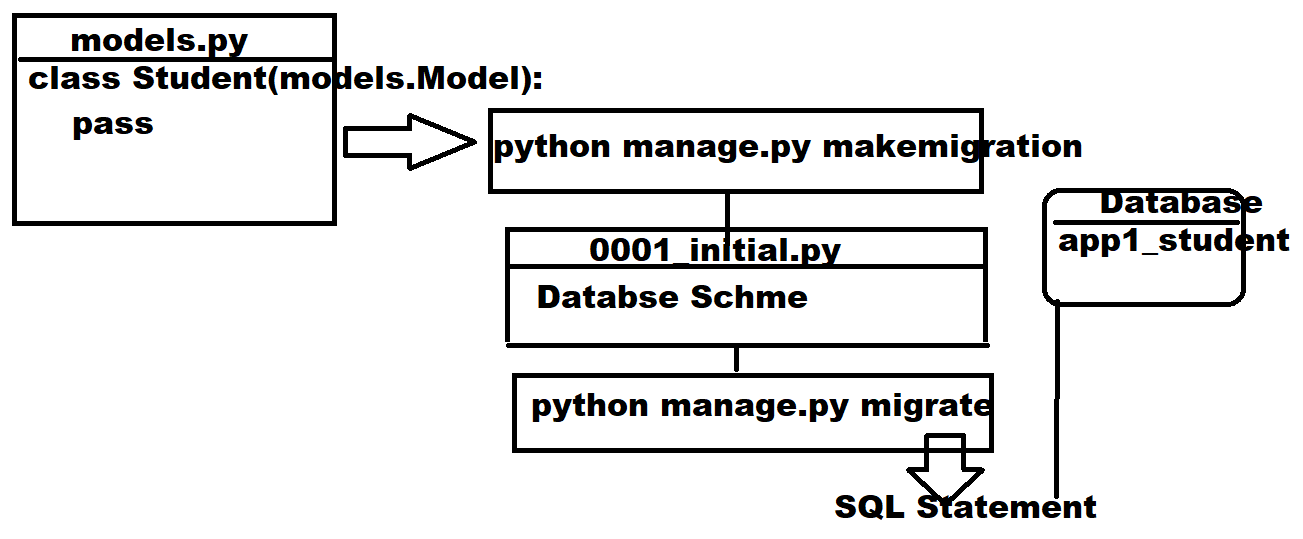
--

-- Create model Student

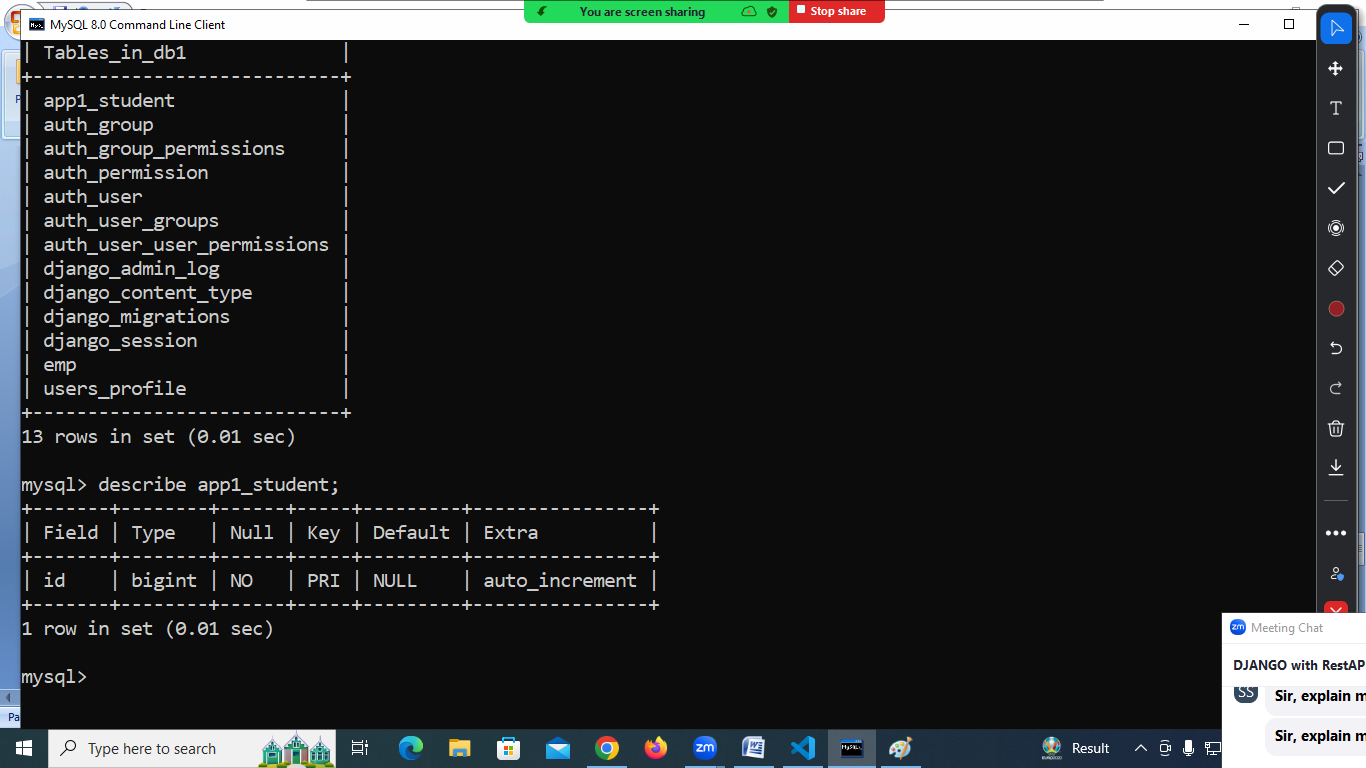
--

CREATE TABLE `app1\_student` (`id` bigint AUTO\_INCREMENT NOT NULL PRIMARY KEY);

PS E:\djangomay7pm\project24>



PS E:\djangomay7pm\project24> python manage.py migrate



**What is difference between makemigartions and migrate?**

Makemigration is a command for manage.py, which is used for creation migration file (application folder/migrations/0001\_intial.py).

This file or module contain information about database schme (structure).

Migrate is a command of manage.py, which is used for generating and executing SQL command. Migrate uses sqlmigrate for generating equal SQL statements. Migrate uses migration file for generating SQL statements.

1. june-2024

Model class represents a table in relational database.

Table is collection of columns (fields) and rows (data)

Model class defined fields, each field is bind with data type which called behavior.

**Field Types**

**IntegerField**

An integer. Values from -2147483648 to 2147483647 are safe in all databases supported by Django.

**FloatField**

A floating-point number represented in Python by a **float** instance.

**CharField**

A string field, for small- to large-sized strings

**TextField**

A large text field. The default form widget for this field is a [Textarea](https://docs.djangoproject.com/en/5.0/ref/forms/widgets/" \l "django.forms.Textarea" \o "django.forms.Textarea).

**DateField**

A date, represented in Python by a datetime.date

**DateTimeField**

A date and time, represented in Python by a datetime.datetime instance.

**\_\_Init\_\_.py of project**

import pymysql

pymysql.install\_as\_MySQLdb()

**settings.py**

DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.mysql',

        'NAME': 'db1',

        'USER': 'root',

        'PASSWORD': 'root',

        'HOST': 'localhost',

        'PORT': '3306',

    }

}

from django.db import models

# Create your models here.

class Employee(models.Model):

    empno=models.IntegerField()

    ename=models.CharField(max\_length=20)

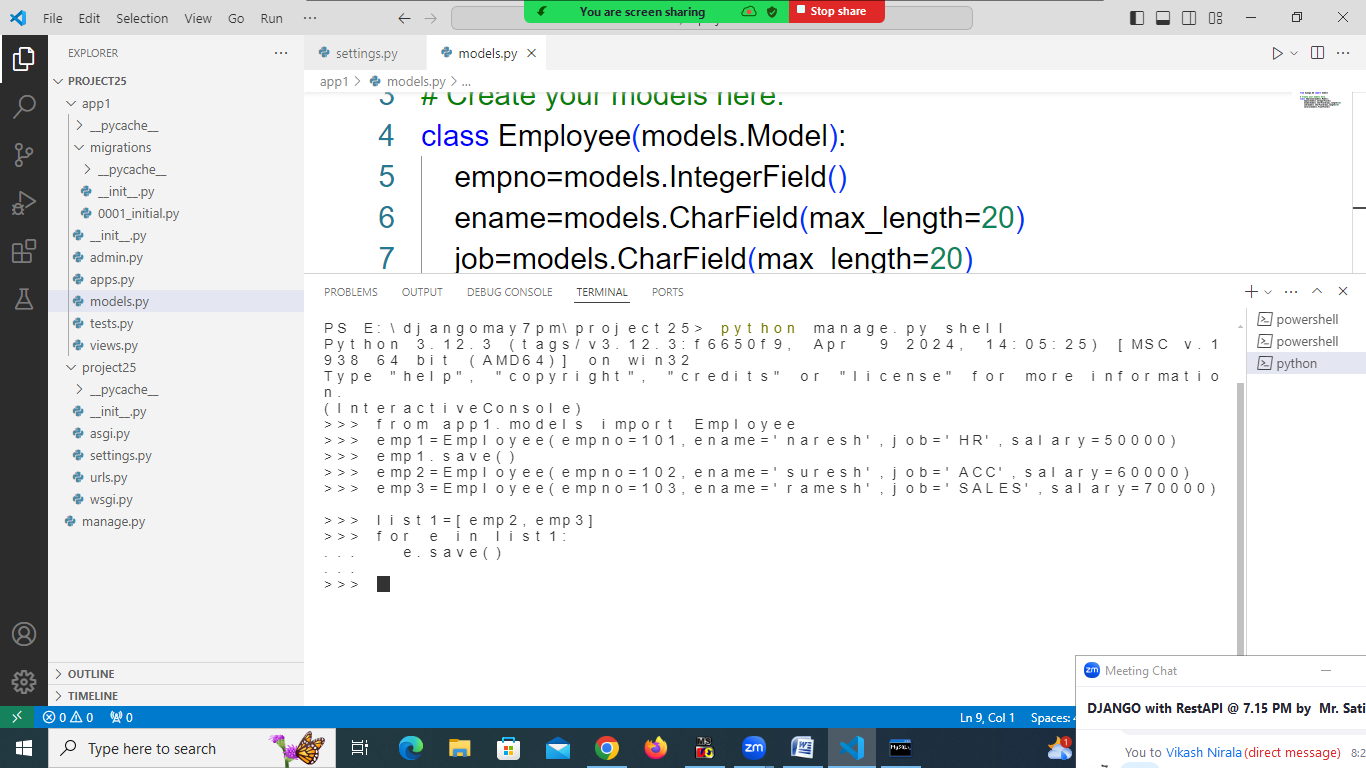
    job=models.CharField(max\_length=20)

    salary=models.FloatField()

**Inserting data/Create row**

Each row is an instance of Employee model class.

1. Open terminal
2. Open django Shell

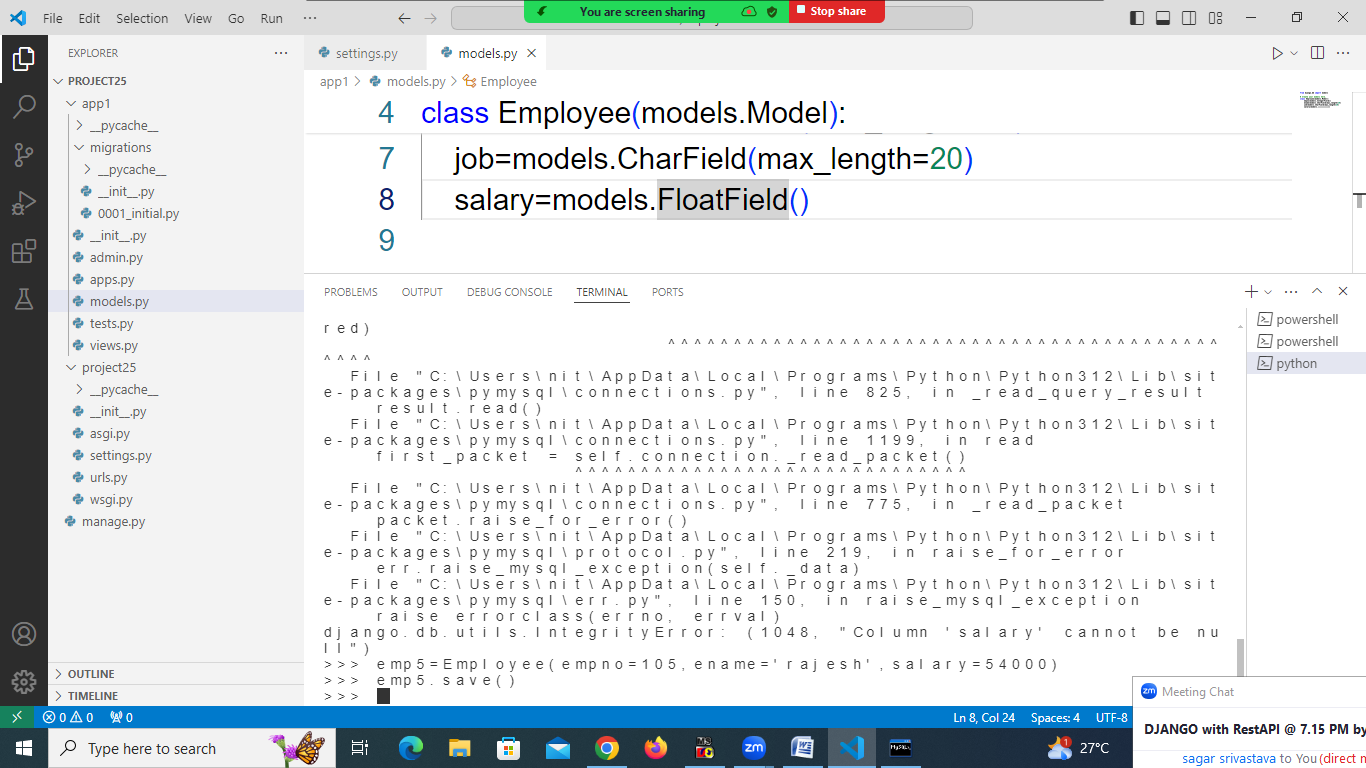


**Syntax:** class-name(field-name=value,field-name=value,

field-name=value,field-name=value,….)

After creating object of model class.

This object is saved within database table using save ()



**Reading data from database table**

Select \* from app1\_employee

Employee.object.all()

1. june-2024

**Reading data from database table**

In ORM, Model class represents table and data is represented as objects.

**Syntax1: model-name.objects.all()**

This all() method of objects returns QuerySet.

QuerySet is object which contains result of executed query.

>>> from app1.models import Employee

>>> a=Employee.objects.all()

>>> type(a)

<class ‘django.db.models.query.QuerySet’>

>>> print(a)

<QuerySet [<Employee: Employee object (1)>, <Employee: Employee object (2)>,

<Employee: Employee object (3)>, <Employee: Employee object (4)>, <Employee:

Employee object (5)>]>

>>> a[1]

<Employee: Employee object (2)>

>>> a[0]

<Employee: Employee object (1)>

>>> a[2]

<Employee: Employee object (3)>

>>> emp1=a[0]

>>> print(emp1.empno)

101

>>> print(emp1.ename)

naresh

>>> print(emp1.job)

HR

>>> print(emp1.salary)

50000.0

>>>

>>> for emp in a:

… print(emp)

…

Employee object (1)

Employee object (2)

Employee object (3)

Employee object (4)

Employee object (5)

>>> for emp in a:

… print(emp.empno,emp.ename,emp.job,emp.salary)

…

101 naresh HR 50000.0

102 suresh ACC 60000.0

103 ramesh SALES 70000.0

104 kishore HR 45000.0

105 rajesh 54000.0

>>>

**Syntax2: Model-class-name.objects.values()**

This method also returns queryset , this queryset contains data.

Each row is return as dictionary, which contains key and value.

>>> b=Employee.objects.values()

>>> print(b)

<QuerySet [{'id': 1, 'empno': 101, 'ename': 'naresh', 'job': 'HR', 'salary':

50000.0}, {'id': 2, 'empno': 102, 'ename': 'suresh', 'job': 'ACC', 'salary':

60000.0}, {'id': 3, 'empno': 103, 'ename': 'ramesh', 'job': 'SALES', 'salary': 70000.0}, {'id': 4, 'empno': 104, 'ename': 'kishore', 'job': 'HR', 'salary': 45000.0}, {'id': 5, 'empno': 105, 'ename': 'rajesh', 'job': '', 'salary': 54000.0}]>

>>>

>>> for row in b:

... print(row)

...

{'id': 1, 'empno': 101, 'ename': 'naresh', 'job': 'HR', 'salary': 50000.0}

{'id': 2, 'empno': 102, 'ename': 'suresh', 'job': 'ACC', 'salary': 60000.0}

{'id': 3, 'empno': 103, 'ename': 'ramesh', 'job': 'SALES', 'salary': 70000.0}{'id': 4, 'empno': 104, 'ename': 'kishore', 'job': 'HR', 'salary': 45000.0}

{'id': 5, 'empno': 105, 'ename': 'rajesh', 'job': '', 'salary': 54000.0}

>>>

>>> for row in b:

... print(row['empno'],row['ename'],row['job'],row['salary'])

...

101 naresh HR 50000.0

102 suresh ACC 60000.0

103 ramesh SALES 70000.0

104 kishore HR 45000.0

105 rajesh 54000.0

>>>

>>> row=Employee.objects.all()[1]

>>> print(row)

Employee object (2)

>>> print(row.empno,row.ename,row.job,row.salary)

102 suresh ACC 60000.0

>>>

**Add \_\_str\_\_ method for returning string representation of object**

from django.db import models

# Create your models here.

class Employee(models.Model):

    empno=models.IntegerField()

    ename=models.CharField(max\_length=20)

    job=models.CharField(max\_length=20)

    salary=models.FloatField()

    def \_\_str\_\_(self):

        return f'{self.empno},{self.ename},{self.job},{self.salary}'

>>> emp1=Employee.objects.all()[0]

>>> print(emp1)

101,naresh,HR,50000.0

>>> print(emp1.\_\_str\_\_())

101,naresh,HR,50000.0

>>>

**Python Code**

class Employee:

def \_\_init\_\_(self,eno,en,j,sal):

self.empno=eno

self.ename=en

self.job=j

self.salary=sal

def \_\_str\_\_(self):

return f'{self.empno},{self.ename},{self.job},{self.salary}'

def \_\_repr\_\_(self):

return f'Employee({self.empno},"{self.ename}","{self.job}",{self.salary})'

emp1=Employee(101,"naresh","HR",60000)

print(emp1)

print(emp1.\_\_str\_\_())

print(repr(emp1))

emp2=eval(repr(emp1))

print(emp2)

A QuerySet represents a collection of objects from your database. It can have zero, one or many filters. Filters narrow down the query results based on the given parameters. In SQL terms, a QuerySet equates to a SELECT statement, and a filter is a limiting clause such as WHERE or LIMIT .

**filter(\*\*kwargs)**

Returns a new [QuerySet](https://docs.djangoproject.com/en/5.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet) containing objects that match the given lookup parameters.

Syntax1: model-class-name.objects.filter(condition)

Syntax2: model-class-name.objects.all().filter(condition)

>>> emp2=Employee.objects.filter(empno=101)

>>> print(emp2)

<QuerySet [<Employee: 101,naresh,HR,50000.0>]>

>>> emp3=Employee.objects.all().filter(empno=102)

>>> print(emp2)

<QuerySet [<Employee: 101,naresh,HR,50000.0>]>

>>> print(emp3)

<QuerySet [<Employee: 102,suresh,ACC,60000.0>]>

>>>

>>> rows=Employee.objects.all().filter(salary=50000)

>>> print(rows)

<QuerySet [<Employee: 101,naresh,HR,50000.0>]>

>>>

**exclude(\*\*kwargs)**

Returns a new [QuerySet](https://docs.djangoproject.com/en/5.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet) containing objects that do *not* match the given lookup parameters.

>>> rows=Employee.objects.exclude(salary=50000)

>>> print(rows)

<QuerySet [<Employee: 102,suresh,ACC,60000.0>, <Employee: 103,ramesh,SALES,70000.0>, <Employee: 104,kishore,HR,45000.0>, <Employee: 105,rajesh,,54000.0>]>>>>

=======================================================

>>> from app1.models import Employee

>>> a=Employee.objects.all()

>>> print(a)

<QuerySet [<Employee: 101,naresh,HR,50000.0>, <Employee: 102,suresh,ACC,60000.0>, <Employee: 103,ramesh,SALES,70000.0>, <Employee: 104,kishore,HR,45000.0>, <Employee: 105,rajesh,,54000.0>]>

>>> for emp in a:

... if emp.salary>50000:

... print(emp)

...

102,suresh,ACC,60000.0

103,ramesh,SALES,70000.0

105,rajesh,,54000.0

>>>

**Chaining filters**

The result of refining a [QuerySet](https://docs.djangoproject.com/en/5.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet) is itself a [QuerySet](https://docs.djangoproject.com/en/5.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet), so it’s possible to chain refinements together.

1. june-2024

**Reading data from database table**

In ORM, Model class represents table and data is represented as objects.

**Syntax1: model-name.objects.all()**

This all() method of objects returns QuerySet.

QuerySet is object which contains result of executed query.

>>> from app1.models import Employee

>>> a=Employee.objects.all()

>>> type(a)

<class ‘django.db.models.query.QuerySet’>

>>> print(a)

<QuerySet [<Employee: Employee object (1)>, <Employee: Employee object (2)>,

<Employee: Employee object (3)>, <Employee: Employee object (4)>, <Employee:

Employee object (5)>]>

>>> a[1]

<Employee: Employee object (2)>

>>> a[0]

<Employee: Employee object (1)>

>>> a[2]

<Employee: Employee object (3)>

>>> emp1=a[0]

>>> print(emp1.empno)

101

>>> print(emp1.ename)

naresh

>>> print(emp1.job)

HR

>>> print(emp1.salary)

50000.0

>>>

>>> for emp in a:

… print(emp)

…

Employee object (1)

Employee object (2)

Employee object (3)

Employee object (4)

Employee object (5)

>>> for emp in a:

… print(emp.empno,emp.ename,emp.job,emp.salary)

…

101 naresh HR 50000.0

102 suresh ACC 60000.0

103 ramesh SALES 70000.0

104 kishore HR 45000.0

105 rajesh 54000.0

>>>

**Syntax2: Model-class-name.objects.values()**

This method also returns queryset , this queryset contains data.

Each row is return as dictionary, which contains key and value.

>>> b=Employee.objects.values()

>>> print(b)

<QuerySet [{'id': 1, 'empno': 101, 'ename': 'naresh', 'job': 'HR', 'salary':

50000.0}, {'id': 2, 'empno': 102, 'ename': 'suresh', 'job': 'ACC', 'salary':

60000.0}, {'id': 3, 'empno': 103, 'ename': 'ramesh', 'job': 'SALES', 'salary': 70000.0}, {'id': 4, 'empno': 104, 'ename': 'kishore', 'job': 'HR', 'salary': 45000.0}, {'id': 5, 'empno': 105, 'ename': 'rajesh', 'job': '', 'salary': 54000.0}]>

>>>

>>> for row in b:

... print(row)

...

{'id': 1, 'empno': 101, 'ename': 'naresh', 'job': 'HR', 'salary': 50000.0}

{'id': 2, 'empno': 102, 'ename': 'suresh', 'job': 'ACC', 'salary': 60000.0}

{'id': 3, 'empno': 103, 'ename': 'ramesh', 'job': 'SALES', 'salary': 70000.0}{'id': 4, 'empno': 104, 'ename': 'kishore', 'job': 'HR', 'salary': 45000.0}

{'id': 5, 'empno': 105, 'ename': 'rajesh', 'job': '', 'salary': 54000.0}

>>>

>>> for row in b:

... print(row['empno'],row['ename'],row['job'],row['salary'])

...

101 naresh HR 50000.0

102 suresh ACC 60000.0

103 ramesh SALES 70000.0

104 kishore HR 45000.0

105 rajesh 54000.0

>>>

>>> row=Employee.objects.all()[1]

>>> print(row)

Employee object (2)

>>> print(row.empno,row.ename,row.job,row.salary)

102 suresh ACC 60000.0

>>>

**Add \_\_str\_\_ method for returning string representation of object**

from django.db import models

# Create your models here.

class Employee(models.Model):

    empno=models.IntegerField()

    ename=models.CharField(max\_length=20)

    job=models.CharField(max\_length=20)

    salary=models.FloatField()

    def \_\_str\_\_(self):

        return f'{self.empno},{self.ename},{self.job},{self.salary}'

>>> emp1=Employee.objects.all()[0]

>>> print(emp1)

101,naresh,HR,50000.0

>>> print(emp1.\_\_str\_\_())

101,naresh,HR,50000.0

>>>

**Python Code**

class Employee:

def \_\_init\_\_(self,eno,en,j,sal):

self.empno=eno

self.ename=en

self.job=j

self.salary=sal

def \_\_str\_\_(self):

return f'{self.empno},{self.ename},{self.job},{self.salary}'

def \_\_repr\_\_(self):

return f'Employee({self.empno},"{self.ename}","{self.job}",{self.salary})'

emp1=Employee(101,"naresh","HR",60000)

print(emp1)

print(emp1.\_\_str\_\_())

print(repr(emp1))

emp2=eval(repr(emp1))

print(emp2)

A QuerySet represents a collection of objects from your database. It can have zero, one or many filters. Filters narrow down the query results based on the given parameters. In SQL terms, a QuerySet equates to a SELECT statement, and a filter is a limiting clause such as WHERE or LIMIT .

**filter(\*\*kwargs)**

Returns a new [QuerySet](https://docs.djangoproject.com/en/5.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet) containing objects that match the given lookup parameters.

Syntax1: model-class-name.objects.filter(condition)

Syntax2: model-class-name.objects.all().filter(condition)

>>> emp2=Employee.objects.filter(empno=101)

>>> print(emp2)

<QuerySet [<Employee: 101,naresh,HR,50000.0>]>

>>> emp3=Employee.objects.all().filter(empno=102)

>>> print(emp2)

<QuerySet [<Employee: 101,naresh,HR,50000.0>]>

>>> print(emp3)

<QuerySet [<Employee: 102,suresh,ACC,60000.0>]>

>>>

>>> rows=Employee.objects.all().filter(salary=50000)

>>> print(rows)

<QuerySet [<Employee: 101,naresh,HR,50000.0>]>

>>>

**exclude(\*\*kwargs)**

Returns a new [QuerySet](https://docs.djangoproject.com/en/5.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet) containing objects that do *not* match the given lookup parameters.

>>> rows=Employee.objects.exclude(salary=50000)

>>> print(rows)

<QuerySet [<Employee: 102,suresh,ACC,60000.0>, <Employee: 103,ramesh,SALES,70000.0>, <Employee: 104,kishore,HR,45000.0>, <Employee: 105,rajesh,,54000.0>]>>>>

=======================================================

>>> from app1.models import Employee

>>> a=Employee.objects.all()

>>> print(a)

<QuerySet [<Employee: 101,naresh,HR,50000.0>, <Employee: 102,suresh,ACC,60000.0>, <Employee: 103,ramesh,SALES,70000.0>, <Employee: 104,kishore,HR,45000.0>, <Employee: 105,rajesh,,54000.0>]>

>>> for emp in a:

... if emp.salary>50000:

... print(emp)

...

102,suresh,ACC,60000.0

103,ramesh,SALES,70000.0

105,rajesh,,54000.0

>>>

**Chaining filters**

The result of refining a [QuerySet](https://docs.djangoproject.com/en/5.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet) is itself a [QuerySet](https://docs.djangoproject.com/en/5.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet), so it’s possible to chain refinements together.

1. june-2024

>>> q1=Employee.objects.filter(salary\_\_gt=50000)

>>> print(q1)

<QuerySet [<Employee: 102,suresh,ACC,60000.0>, <Employee: 103,ramesh,SALES,70000.0>, <Employee: 105,rajesh,,54000.0>]>

>>> q2=Employee.objects.filter(salary\_\_lt=50000)

>>> print(q2)

<QuerySet [<Employee: 104,kishore,HR,45000.0>]>

**Operators uses with query set**

1. exact
2. iexact
3. contains
4. icontains
5. in
6. gt
7. gte
8. lt
9. lte
10. startswith
11. istartswith
12. endswith
13. iendswith

>>> q3=Employee.objects.filter(salary\_\_gte=50000)

>>> print(q3)

<QuerySet [<Employee: 101,naresh,HR,50000.0>, <Employee: 102,suresh,ACC,60000.0>, <Employee: 103,ramesh,SALES,70000.0>, <Employee: 105,rajesh,,54000.0>]>

>>>

>>> q4=Employee.objects.filter(salary\_\_lte=50000)

>>> print(q4)

<QuerySet [<Employee: 101,naresh,HR,50000.0>, <Employee: 104,kishore,HR,45000.0>]>

>>>

>>> q5=Employee.objects.filter(ename\_\_startswith='n')

>>> print(q5)

<QuerySet [<Employee: 101,naresh,HR,50000.0>]>

>>>

>>> q6=Employee.objects.filter(ename\_\_endswith='h')

>>> print(q6)

<QuerySet [<Employee: 101,naresh,HR,50000.0>, <Employee: 102,suresh,ACC,60000.0>, <Employee: 103,ramesh,SALES,70000.0>, <Employee: 105,rajesh,,54000.0>]>

>>>

>>> q7=Employee.objects.filter(job\_\_in=['HR','ACC'])

>>> print(q7)

<QuerySet [<Employee: 101,naresh,HR,50000.0>, <Employee: 102,suresh,ACC,60000.0>, <Employee: 104,kishore,HR,45000.0>]>

>>>

>>> q8=Employee.objects.filter(ename\_\_contains='a')

>>> print(q8)

<QuerySet [<Employee: 101,naresh,HR,50000.0>, <Employee: 103,ramesh,SALES,70000.0>, <Employee: 105,rajesh,,54000.0>]>

>>>

>>> q9=Employee.objects.filter(empno\_\_range=(101,103))

>>> print(q9)

<QuerySet [<Employee: 101,naresh,HR,50000.0>, <Employee: 102,suresh,ACC,60000.0>, <Employee: 103,ramesh,SALES,70000.0>]>

>>>

**Filter Chaining**

The result of refining a [QuerySet](https://docs.djangoproject.com/en/5.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet) is itself a [QuerySet](https://docs.djangoproject.com/en/5.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet), so it’s possible to chain refinements together.

>>> q10=Employee.objects.filter(salary\_\_gt=50000).exclude(job='HR')

>>> print(q10)

<QuerySet [<Employee: 102,suresh,ACC,60000.0>, <Employee: 103,ramesh,SALES,70000.0>, <Employee: 105,rajesh,,54000.0>]>

>>>

**Update data**

>>> emp1=Employee.objects.all()[0]

>>> print(emp1)

101,naresh,HR,50000.0

>>> emp1.salary=emp1.salary+100

>>> emp1.save()

>>>

1. june-2024

**Retrieving a single object with get()**

[filter()](https://docs.djangoproject.com/en/5.0/ref/models/querysets/" \l "django.db.models.query.QuerySet.filter" \o "django.db.models.query.QuerySet.filter) will always give you a [QuerySet](https://docs.djangoproject.com/en/5.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet), even if only a single object matches the query - in this case, it will be a [QuerySet](https://docs.djangoproject.com/en/5.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet) containing a single element.

If you know there is only one object that matches your query, you can use the [get()](https://docs.djangoproject.com/en/5.0/ref/models/querysets/" \l "django.db.models.query.QuerySet.get" \o "django.db.models.query.QuerySet.get) method on a [Manager](https://docs.djangoproject.com/en/5.0/topics/db/managers/" \l "django.db.models.Manager" \o "django.db.models.Manager) which returns the object directly:

>>> from app1.models import Employee

>>> emp1=Employee.objects.get(empno=102)

>>> print(emp1)

102,suresh,ACC,60000.0

>>> print(type(emp1))

<class 'app1.models.Employee'>

>>>

>>> emp1.salary=emp1.salary+500

>>> emp1.save()

>>>

>>> emp=Employee.objects.all()

>>> for e in emp:

... e.salary=e.salary+100

... e.save()

...

>>>

>>> emp=Employee.objects.all().filter(job='HR')

>>> for e in emp:

... e.salary+=100

... e.save()

...

>>>

**Delete object or deleting row**

The delete method, conveniently, is named [delete()](https://docs.djangoproject.com/en/5.0/ref/models/instances/" \l "django.db.models.Model.delete" \o "django.db.models.Model.delete). This method immediately deletes the object and returns the number of objects deleted and a dictionary with the number of deletions per object type.

>>> emp=Employee.objects.get(empno=105)

>>> x=emp.delete()

>>> print(x)

(1, {'app1.Employee': 1})

>>>

>>> y=Employee.objects.filter(job='HR').delete()

>>> print(y)

(2, {'app1.Employee': 2})

>>>

You can also delete objects in bulk. Every [QuerySet](https://docs.djangoproject.com/en/5.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet) has a [delete()](https://docs.djangoproject.com/en/5.0/ref/models/querysets/" \l "django.db.models.query.QuerySet.delete" \o "django.db.models.query.QuerySet.delete) method, which deletes all members of that [QuerySet](https://docs.djangoproject.com/en/5.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet).

**Updating multiple objects at once**

Sometimes you want to set a field to a particular value for all the objects in a **[QuerySet](https://docs.djangoproject.com/en/5.0/ref/models/querysets/" \l "django.db.models.query.QuerySet" \o "django.db.models.query.QuerySet)**. You can do this with the **[update()](https://docs.djangoproject.com/en/5.0/ref/models/querysets/" \l "django.db.models.query.QuerySet.update" \o "django.db.models.query.QuerySet.update)** method.

>>> emp=Employee.objects.all().update(salary=9000)

>>>

>>> emp=Employee.objects.get(empno=102)

>>> emp.salary=emp.salary+500

>>> emp.save()

>>>

**Aggregation**

Aggregation is nothing applying group operations OR applying aggregate functions (sum, avg, min, max,…)

aggregate() is a terminal clause for a QuerySet that, when invoked, returns a dictionary of name-value pairs. The name is an identifier for the aggregate value; the value is the computed aggregate. The name is automatically generated from the name of the field and the aggregate function. If you want to manually specify a name for the aggregate value, you can do so by providing that name when you specify the aggregate clause:

>>> from django.db.models import Sum,Avg,Min,Max

>>> Employee.objects.aggregate(Sum('salary'))

{'salary\_\_sum': 36700.0}

>>>

>>> Employee.objects.aggregate(Min('salary'))

{'salary\_\_min': 4600.0}

>>> Employee.objects.aggregate(Max('salary'))

{'salary\_\_max': 9500.0}

>>> Employee.objects.aggregate(Avg('salary'))

{'salary\_\_avg': 7340.0}

>>>

>>> Employee.objects.filter(job='HR').aggregate(Sum('salary'))

{'salary\_\_sum': 13600.0}

>>>

1. june-2024

**Q Objects in Django**

The Django Q object (django.db.models.Q) is a container for keyword arguments. It is primarily used for complex queries that require logical operations. The keyword arguments are encapsulated by the Q object and passed to query methods such as filter, exclude, and get. Multiple Q objects can be combined using the AND, OR, or NOT operators.

& 🡪 AND

| 🡪 OR

~ 🡪 NOT

Syntax: Q(kwargs)

**Example:**

>>> from django.db.models import Q

>>> emp=Employee.objects.filter(Q(job='HR')|Q(job='SALES'))

>>> print(emp)

<QuerySet [<Employee: 103,ramesh,SALES,9000.0>, <Employee: 101,naresh,HR,5600.0>, <Employee: 104,kishore,HR,8000.0>]>

>>>

>>> emp=Employee.objects.filter(Q(job='HR')&Q(salary\_\_gt=50000))

>>> print(emp)

<QuerySet []>

>>> emp=Employee.objects.filter(Q(job='HR')&Q(salary\_\_gt=5000))

>>> print(emp)

<QuerySet [<Employee: 101,naresh,HR,5600.0>, <Employee: 104,kishore,HR,8000.0>]>

>>>

>>> emp=Employee.objects.filter(~Q(job='HR'))

>>> print(emp)

<QuerySet [<Employee: 102,suresh,ACC,9500.0>, <Employee: 103,ramesh,SALES,9000.0>, <Employee: 105,kiran,ACC,4600.0>]>

>>>

>>> emp=Employee.objects.filter(~Q(job='HR')&Q(salary\_\_gt=5000))

>>> print(emp)

<QuerySet [<Employee: 102,suresh,ACC,9500.0>, <Employee: 103,ramesh,SALES,9000.0>]>

>>>

**F objects**

An F() object represents the value of a model field, its transformed value, or an annotated column. It is primarily used to compare the model class's A and B field attributes, i.e. to operate on the value of a column in the database.

Syntax: F(‘field-name’)

>>> from django.db.models import F

>>> c=Employee.objects.all().update(salary=F('salary')+500)

>>> print(c)

5

>>>

>>> emp=Employee.objects.filter(salary\_\_lt=F('salary')\*2)

>>> print(emp)

<QuerySet [<Employee: 102,suresh,ACC,10000.0>, <Employee: 103,ramesh,SALES,9500.0>, <Employee: 101,naresh,HR,6100.0>, <Employee: 104,kishore,HR,8500.0>, <Employee: 105,kiran,ACC,5100.0>]>

>>>

**Model Relationships**

A project consists of one or more than one table (model)

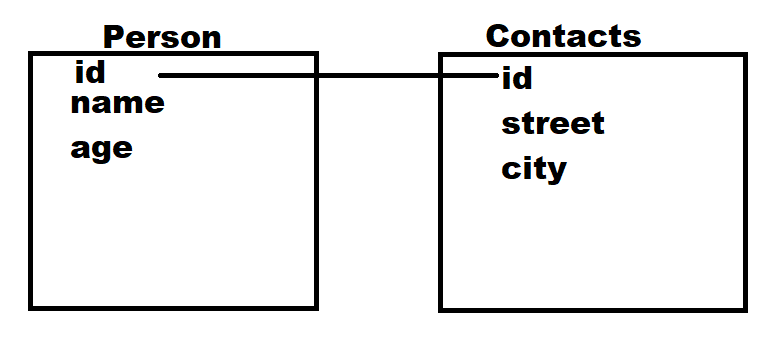
The relationship between models is done to avoid redundancy and inconsistency.

**Type of relationships between models**

1. One to one relationship
2. One to many relationship
3. Many to many relationship

**One to One relationship**

One record in a table is associated with one and only one record in another table.



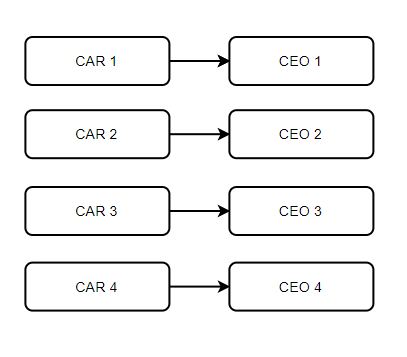
**Syntax:**

**Django.db.models.OneToOneField(Model-class,on\_delete=**models.CASCADE)

1. june-2024

**One-To-One Relationship (OneToOneField)**

One record in a table is associated with one and only one record in another table.



*- Car company can have only one CEO.  
- CEO can work only one Car company.*

|----------|--------------|  
|**CarCompany**| **CEO** |  
|----------|--------------|  
|Mercedes | Adam Smith |  
|BMW | Jack Ryan |  
|Audi | Kevin Bishop |  
|----------|--------------|

from django.db import models

class Car(models.Model):

name = models.CharField(max\_length=255)

def \_\_str\_\_(self):

return self.name

class Ceo(models.Model):

car = models.OneToOneField(Car, on\_delete=models.CASCADE)

name = models.CharField(max\_length=255, blank=True)

def \_\_str\_\_(self):

return self.name

## **Add Car:**

**>>> from app1.models import Car, Ceo**

**>>> mercedes = Car.objects.create(name=”Mercedes”)  
>>> audi = Car.objects.create(name=”Audi”)  
>>>bmw = Car.objects.create(name=”BMW”)**

**>>>Car.objects.all()**<QuerySet [  
<Car: Mercedes>,   
<Car: Audi>,   
<Car: BMW>  
]>

## **Add Ceo:**

**>>>mercedes\_ceo = Ceo.objects.create(name=”Adam Smith”, car=mercedes)  
>>> audi\_ceo = Ceo.objects.create(name=”Kevin Bishop”, car=audi)  
>>> bmw\_ceo = Ceo.objects.create(name=”Jack Ryan”, car=bmw)  
  
>>> Ceo.objects.all()**<QuerySet [  
<Ceo: Adam Smith>,   
<Ceo: Kevin Bishop>,   
<Ceo: Jack Ryan>  
]>

## **Retrieve Records :**

**From car to ceo:**

**>>>mercedes.ceo**

<Ceo: Adam Smith>

**>>> mercedes.ceo.name**

Adam Smith

**From ceo to car:**

**>>> ceo = Ceo.objects.get(name=”Adam Smith”)  
>>>ceo**

<Ceo: Adam Smith>

**>>>ceo.car**

<Car: Mercedes>

**>>> ceo.car.name**

Mercedes

**>>>ceo.car.ceo**

<Ceo: Adam Smith>

**Filter cars by ceo:**

**>>>Car.objects.filter(ceo\_\_name\_\_startswith=”A”).all()**

<QuerySet [  
<Car: Mercedes>  
]>

**Filter cars by ceo:**

**>>> Ceo.objects.get(car\_\_name=”Mercedes”)**

<Ceo: Adam Smith>

## **Delete Records :**

When an object is deleted, Django will emulate the behavior of the SQL constraint specified by the on\_delete argument. There are seven possible actions to take when such event occurs:

***CASCADE****: When the referenced object is deleted, also delete the objects that have references to it .****PROTECT****: Forbid the deletion of the referenced object. To delete it you will have to delete all objects that reference it manually.****RESTRICT****: Similar behavior as PROTECT. Unlike PROTECT, deletion of the referenced object is allowed if it also references a different object that is being deleted in the same operation, but via a CASCADE relationship.****SET\_NULL****: Set the reference to NULL****SET\_DEFAULT****: Set the default value.****SET()****: Set a given value.****DO\_NOTHING****: Take no action.*

class Ceo(models.Model):

name = models.CharField(max\_length=255, blank=True)

car = models.OneToOneField(Car, on\_delete=models.CASCADE)

def \_\_str\_\_(self):

return self.name

When the car record is deleted, the ceo record is also deleted.

**>>>audi.delete()**

(2, {'modelrelations.Ceo': 1, 'modelrelations.Car': 1})

**>>>Car.objects.all()**

<QuerySet [  
<Car: Mercedes>,   
<Car: BMW>  
]>

**>>>Ceo.objects.all()**<QuerySet [  
<Ceo: Adam Smith>,   
<Ceo: Jack Ryan>  
]>

**SET\_NULL:**

class Ceo(models.Model):

name = models.CharField(max\_length=255, blank=True)

car = models.OneToOneField(Car, on\_delete=models.SET\_NULL, blank=True, null=True)

def \_\_str\_\_(self):

return self.name

When the car record is deleted, the car field in the ceo record is set to null.

**>>>bmw.delete()**

(1, {‘modelrelations.Car’: 1})

**>>>Car.objects.all()**

<QuerySet [  
<Car: Mercedes>  
]>

**>>>Ceo.objects.all()**

<QuerySet [  
<Ceo: Adam Smith>,   
<Ceo: Jack Ryan>  
]>

When we delete BMW, BMW’s ceo set as null.

**>>> bmw\_ceo.car.name**

AttributeError: ‘NoneType’ object has no attribute ‘name’

We can create a BMW record and reassign this CEO to that record.

**>>>bmw = Car.objects.create(name=”BMW”)  
>>>bmw.ceo= bmw\_ceo  
>>>bmw.ceo.save()  
>>>bmw.ceo**

<Ceo: Jack Ryan>

When we try to assign the same ceo to Mercedes, we get an IntegrityError error.

**$ mercedes.ceo = ceo  
$ mercedes.ceo.save()**

django.db.utils.IntegrityError: UNIQUE constraint failed: modelrelations\_ceo.car\_id

24-june-2024

**One-To-One relationship**

The one-to-one relationship defines a link between two tables, where each row in a table appears once in another table.

For example, each employee has a contact and each contact belongs to one employee. So the relationship between employees and contacts is a one-to-one relationship.

To create a one-to-one relationship, you use the OneToOneField class:

The following example uses a OneToOneField class to define a one-to-one relationship between Contact and Employee models in the models.py

from django.db import models

class Contact(models.Model):

phone = models.CharField(max\_length=50, unique=True)

address = models.CharField(max\_length=50)

def \_\_str\_\_(self):

return self.phone

class Employee(models.Model):

first\_name = models.CharField(max\_length=100)

last\_name = models.CharField(max\_length=100)

contact = models.OneToOneField(Contact, on\_delete=models.CASCADE)

def \_\_str\_\_(self):

return f'{self.first\_name} {self.last\_name}'

First, create a new Employee object and save it to the database:

>>>e = Employee(first\_name='John',last\_name='Doe')

>>>e.save() # Error

Second, create and save a new contact into the database:

>>>c = Contact(phone='40812345678', address='101 N 1st Street, San Jose, CA')

>>>c.save()

Third, associate a contact with an employee:

>>>e.contact = c

>>>e.save()

Getting data from a one-to-one relationship

First, get the employee with the name John Doe:

>>>e = Employee.objects.filter(first\_name='John',last\_name='Doe').first()

<Employee: John Doe>

When you access the contact attribute of the employee:

>>>e.contact

The following gets the contact with id 1:

>>>c = Contact.objects.get(id=1)

When associating a contact with an employee, you can access the employee from the contact object:

>>>c.employee

<Employee: John Doe>

Let’s create another contact that doesn’t associate with any employee:

>>>c = Contact(phone='4081111111',address='202 N 1st Street, San Jose, CA')

>>>c.save()

>>>e = Employee(first\_name='Jane',last\_name='Doe')

>>> e.contact=c

>>>e.save()

>>>Employee.objects.all()

Django returns two employees:

<QuerySet [<Employee: John Doe>, <Employee: Jane Doe>]>

if you have to display all employees as well as their contacts on the same page, then you have the N+1 query problem:

First, you need one query to get all employees (N employees).

Second, you need N queries to select the related contact of each employee.

To avoid this, you can query all employees and contacts using a single query by using the **select\_related()** method:

>>>Employee.objects.select\_related('contact').all()

QuerySet [<Employee: John Doe>, <Employee: Jane Doe>]>

**Django one-to-many relationships**

In a one-to-many relationship, a row in a table is associated with one or more rows in another table. For example, a department may have one or more employees and each employee belongs to one department.

The relationship between departments and employees is a one-to-many relationship. Conversely, the relationship between employees and departments is a many-to-one relationship.

To create a one-to-many relationship in Django, you use ForeignKey. For example, the following uses the ForeignKey to create a one-to-many relationshipbetween Department and Employee models

from django.db import models

class Contact(models.Model):

phone = models.CharField(max\_length=50, unique=True)

address = models.CharField(max\_length=50)

def \_\_str\_\_(self):

return self.phone

class Department(models.Model):

name = models.CharField(max\_length=255)

description = models.TextField(null=True, blank=True)

def \_\_str\_\_(self):

return self.name

class Employee(models.Model):

first\_name = models.CharField(max\_length=100)

last\_name = models.CharField(max\_length=100)

contact = models.OneToOneField(

Contact,

on\_delete=models.CASCADE,

null=True

)

department = models.ForeignKey(

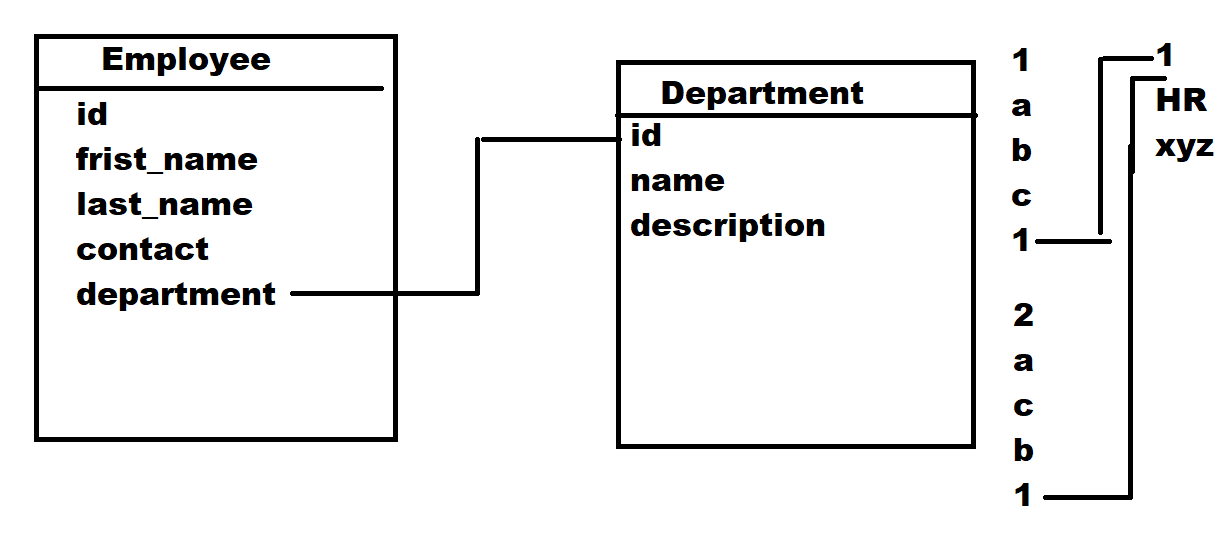
Department,

on\_delete=models.CASCADE

)

def \_\_str\_\_(self):

return f'{self.first\_name} {self.last\_name}'



>>>d = Department(name='IT',description='Information Technology')

>>>d.save()

>>>e = Employee(first\_name='John',last\_name='Doe',department=d)

>>>e.save()

>>>e = Employee(first\_name='Jane',last\_name='Doe',department=d)

>>>e.save()

>>>e.department

<Department: IT>

>>>e.department.description

'Information Technology'

>>>d.employee\_set.all()

<QuerySet [<Employee: John Doe>, <Employee: Jane Doe>]>

Note that we did not define the employee\_set property in the Department model. Internally, Django automatically added the employee\_set property to the Department model when we defined the one-to-many relationship using the ForeignKey.

The all() method of the employee\_set returns a QuerySet that contains all employees who belong to the department.

Using select\_related() to join employee with department

The following returns the first employee:

>>> e = Employee.objects.first()

To access the department of the first employee, you use the department attribute:

>>>e.department

In this case, Django executes two queries. The first query selects the first employee and the second query selects the department of the selected employee.

If you select N employees to display them on a web page, then you need to execute N + 1 query to get both employees and their departments. The first query (1) selects the N employees and the N queries select N departments for each employee. This issue is known as the N + 1 query problem.

To fix the N + 1 query problem, you can use the select\_related() method to select both employees and departments using a single query.

>>>Employee.objects.select\_related('department').all()

1. june-2024

**One-To-One relationship**

The one-to-one relationship defines a link between two tables, where each row in a table appears once in another table.

For example, each employee has a contact and each contact belongs to one employee. So the relationship between employees and contacts is a one-to-one relationship.

To create a one-to-one relationship, you use the OneToOneField class:

The following example uses a OneToOneField class to define a one-to-one relationship between Contact and Employee models in the models.py

from django.db import models

class Contact(models.Model):

phone = models.CharField(max\_length=50, unique=True)

address = models.CharField(max\_length=50)

def \_\_str\_\_(self):

return self.phone

class Employee(models.Model):

first\_name = models.CharField(max\_length=100)

last\_name = models.CharField(max\_length=100)

contact = models.OneToOneField(Contact, on\_delete=models.CASCADE)

def \_\_str\_\_(self):

return f'{self.first\_name} {self.last\_name}'

First, create a new Employee object and save it to the database:

>>>e = Employee(first\_name='John',last\_name='Doe')

>>>e.save() # Error

Second, create and save a new contact into the database:

>>>c = Contact(phone='40812345678', address='101 N 1st Street, San Jose, CA')

>>>c.save()

Third, associate a contact with an employee:

>>>e.contact = c

>>>e.save()

Getting data from a one-to-one relationship

First, get the employee with the name John Doe:

>>>e = Employee.objects.filter(first\_name='John',last\_name='Doe').first()

<Employee: John Doe>

When you access the contact attribute of the employee:

>>>e.contact

The following gets the contact with id 1:

>>>c = Contact.objects.get(id=1)

When associating a contact with an employee, you can access the employee from the contact object:

>>>c.employee

<Employee: John Doe>

Let’s create another contact that doesn’t associate with any employee:

>>>c = Contact(phone='4081111111',address='202 N 1st Street, San Jose, CA')

>>>c.save()

>>>e = Employee(first\_name='Jane',last\_name='Doe')

>>> e.contact=c

>>>e.save()

>>>Employee.objects.all()

Django returns two employees:

<QuerySet [<Employee: John Doe>, <Employee: Jane Doe>]>

if you have to display all employees as well as their contacts on the same page, then you have the N+1 query problem:

First, you need one query to get all employees (N employees).

Second, you need N queries to select the related contact of each employee.

To avoid this, you can query all employees and contacts using a single query by using the **select\_related()** method:

>>>Employee.objects.select\_related('contact').all()

QuerySet [<Employee: John Doe>, <Employee: Jane Doe>]>

**Django one-to-many relationships**

In a one-to-many relationship, a row in a table is associated with one or more rows in another table. For example, a department may have one or more employees and each employee belongs to one department.

The relationship between departments and employees is a one-to-many relationship. Conversely, the relationship between employees and departments is a many-to-one relationship.

To create a one-to-many relationship in Django, you use ForeignKey. For example, the following uses the ForeignKey to create a one-to-many relationship between Department and Employee models

from django.db import models

class Contact(models.Model):

phone = models.CharField(max\_length=50, unique=True)

address = models.CharField(max\_length=50)

def \_\_str\_\_(self):

return self.phone

class Department(models.Model):

name = models.CharField(max\_length=255)

description = models.TextField(null=True, blank=True)

def \_\_str\_\_(self):

return self.name

class Employee(models.Model):

first\_name = models.CharField(max\_length=100)

last\_name = models.CharField(max\_length=100)

contact = models.OneToOneField(

Contact,

on\_delete=models.CASCADE,

null=True

)

department = models.ForeignKey(

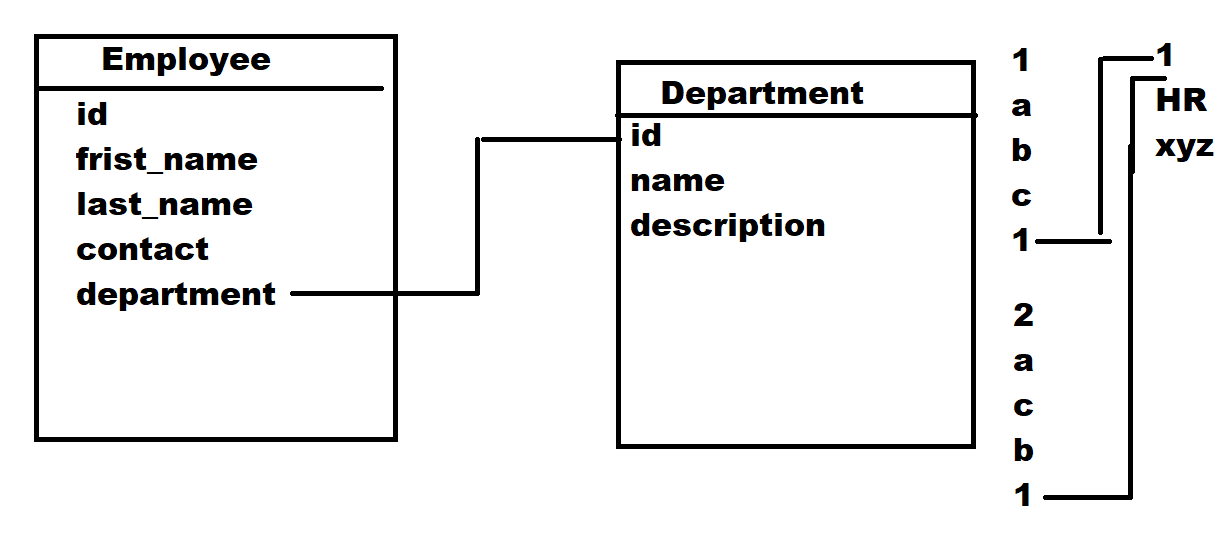
Department,

on\_delete=models.CASCADE

)

def \_\_str\_\_(self):

return f'{self.first\_name} {self.last\_name}'



>>>d = Department(name='IT',description='Information Technology')

>>>d.save()

>>>e = Employee(first\_name='John',last\_name='Doe',department=d)

>>>e.save()

>>>e = Employee(first\_name='Jane',last\_name='Doe',department=d)

>>>e.save()

>>>e.department

<Department: IT>

>>>e.department.description

'Information Technology'

>>>d.employee\_set.all()

<QuerySet [<Employee: John Doe>, <Employee: Jane Doe>]>

Note that we did not define the employee\_set property in the Department model. Internally, Django automatically added the employee\_set property to the Department model when we defined the one-to-many relationship using the ForeignKey.

The all() method of the employee\_set returns a QuerySet that contains all employees who belong to the department.

Using select\_related() to join employee with department

The following returns the first employee:

>>> e = Employee.objects.first()

To access the department of the first employee, you use the department attribute:

>>>e.department

In this case, Django executes two queries. The first query selects the first employee and the second query selects the department of the selected employee.

If you select N employees to display them on a web page, then you need to execute N + 1 query to get both employees and their departments. The first query (1) selects the N employees and the N queries select N departments for each employee. This issue is known as the N + 1 query problem.

To fix the N + 1 query problem, you can use the select\_related() method to select both employees and departments using a single query.

>>>Employee.objects.select\_related('department').all()

mysql> select app1\_employee.first\_name,app1\_department.name from app1\_employee,app1\_department where app1\_employee.department\_id=app1\_department.id;

+------------+------+

| first\_name | name |

+------------+------+

| John | IT |

| Jane | IT |

+------------+------+

2 rows in set (0.00 sec)

**Django Many-to-Many relationship**

In a many-to-many relationship, multiple rows in a table are associated with multiple rows in another table.

For example, an employee may have multiple compensation programs and a compensation program may belong to multiple employees.

Therefore, multiple rows in the employee table are associated with multiple rows in the compensation table. Hence, the relationship between employees and compensation programs is a many-to-many relationship.

Typically, relational databases do not implement a direct many-to-many relationship between two tables. Instead, it uses a third table, the join table, to establish two [one-to-many relationships](https://www.pythontutorial.net/django-tutorial/django-one-to-many/) between the two tables and the join table.

# ...

class Compensation(models.Model):

name = models.CharField(max\_length=255)

def \_\_str\_\_(self):

return self.name

class Employee(models.Model):

first\_name = models.CharField(max\_length=100)

last\_name = models.CharField(max\_length=100)

contact = models.OneToOneField(

Contact,

on\_delete=models.CASCADE,

null=True

)

department = models.ForeignKey(

Department,

on\_delete=models.CASCADE

)

compensations = models.ManyToManyField(Compensation)

def \_\_str\_\_(self):

return f'{self.first\_name} {self.last\_name}'

Creating data

>>> c1 = Compensation(name='Stock')

>>> c1.save()

>>> c2 = Compensation(name='Bonuses')

>>> c2.save()

>>> c3 = Compensation(name='Profit Sharing')

>>> c3.save()

>>>Compensation.objects.all()

<QuerySet [<Compensation: Stock>, <Compensation: Bonuses>, <Compensation: Profit Sharing>]>

Third, get the employee with the first name John and last name Doe:

>>> e = Employee.objects.filter(first\_name='John',last\_name='Doe').first()

>>> e

<Employee: John Doe>

Adding compensations to employees

First, enroll John Doe in the stock (c1) and bonuses (c2) compensation programs using the add() method of the compensations attribute and the save() method of the Employee object:

>>>e.compensations.add(c1)

>>>e.compensations.add(c2)

>>>e.save()

Second, access all compensations program of John Doe using the all() method of the compensations attribute:

>>>e.compensations.all()

<QuerySet [<Compensation: Stock>, <Compensation: Bonuses>]> )

As clearly shown in the output, John Doe has two compensation programs.

Third, enroll Jane Doe in three compensation programs including stock, bonuses, and profit sharing:

>>>e = Employee.objects.filter(first\_name='Jane',last\_name='Doe').first()

>>>e

<Employee: Jane Doe>

>>>e.compensations.add(c1)

>>>e.compensations.add(c2)

>>>e.compensations.add(c3)

>>>e.save()

>>>e.compensations.all()

<QuerySet [<Compensation: Stock>, <Compensation: Bonuses>, <Compensation: Profit Sharing>]>

Internally, Django inserted the ids of employees and compensations into the join table:

id | employee\_id | compensation\_id

----+-------------+-----------------

1 | 5 | 1

2 | 5 | 2

3 | 6 | 1

4 | 6 | 2

5 | 6 | 3

(5 rows)

Fourth, find all employees who were enrolled in the stock compensation plan using the employee\_set attribute of the Compensation object:

>>>c1

<Compensation: Stock>

>>>c1.employee\_set.all()

<QuerySet [<Employee: John Doe>, <Employee: Jane Doe>]>

It returned two employees as expected.

Fifth, you can use the employee\_set attribute to find all employees who have the profit-sharing compensation program:

>>>c3

<Compensation: Profit Sharing>

>>>c3.employee\_set.all()

<QuerySet [<Employee: Jane Doe>]>

It returned one employee.

Django allows you to query across the relationship. For example, you can find all employees who have the compensation with id 1:

>>>Employee.objects.filter(compensations\_\_id=1)

<QuerySet [<Employee: John Doe>, <Employee: Jane Doe>]>

Or with the name "Profit Sharing":

>>>Employee.objects.filter(compensations\_\_name="Profit Sharing")

<QuerySet [<Employee: Jane Doe>]>

**Removing compensations from employees**

To remove a compensation program from an employee, you use the remove() method of the compensations attribute of the Employee object. For example:

First, get the employee whose name is Jane Doe:

>>>e = Employee.objects.filter(first\_name='Jane',last\_name='Doe').first()

>>>e

<Employee: Jane Doe>Code language:Python(python)

Second, remove the profit sharing compensation (c3) from Jane Doe and save the changes to the database:

>>>e.compensations.remove(c3)

>>>e.save()Code language:Python(python)

Third, get all the compensation programs of Jane Doe:

>>>e.compensations.all()

<QuerySet [<Compensation: Stock>, <Compensation: Bonuses>]>

**Orderby() method of QuerySet object**

we are using **Django Query Set** to perform the Order By operation

**Syntax:** order\_by(‘column-name’) 🡪 sort in ascending order

**Syntax:** order\_by(‘-column-name’) 🡪 sort in descending order

>>> from app1.models import \*

>>> e1=Employee.objects.values()

>>> print(e1)

<QuerySet [{'id': 2, 'empno': 102, 'ename': 'suresh', 'job': 'ACC', 'salary': 10000.0}, {'id': 3, 'empno': 103, 'ename': 'ramesh', 'job': 'SALES', 'salary': 9500.0}, {'id': 6, 'empno': 101, 'ename': 'naresh', 'job': 'HR', 'salary': 6100.0}, {'id': 7, 'empno': 104, 'ename': 'kishore', 'job': 'HR', 'salary': 8500.0}, {'id': 8, 'empno': 105, 'ename': 'kiran', 'job': 'ACC', 'salary': 5100.0}]>

>>> for emp in e1:

... print(emp['empno'],emp['ename'],emp['job'],emp['salary'])

...

102 suresh ACC 10000.0

103 ramesh SALES 9500.0

101 naresh HR 6100.0

104 kishore HR 8500.0

105 kiran ACC 5100.0

>>> e2=Employee.objects.values().order\_by('salary')

>>> for emp in e2:

... print(emp['empno'],emp['ename'],emp['job'],emp['salary'])

...

105 kiran ACC 5100.0

101 naresh HR 6100.0

104 kishore HR 8500.0

103 ramesh SALES 9500.0

102 suresh ACC 10000.0

>>> e3=Employee.objects.values().order\_by('-salary')

>>> for emp in e3:

... print(emp['empno'],emp['ename'],emp['job'],emp['salary'])

...

102 suresh ACC 10000.0

103 ramesh SALES 9500.0

104 kishore HR 8500.0

101 naresh HR 6100.0

105 kiran ACC 5100.0

>>>