RAE411

Telecommunications Software

Name: Farjana Rahman

Student ID: 221AEM003

Session: Spring 2023

Practical Exercise No: 3

Practical Exercise Title: Practical Exercise 3 (Python cloud full stack development)

Course Instructor: Tianhua Chen

EXAMPLE 1/TASK 1

Django is a Python web framework that allows for the rapid building of safe and maintained websites. Django takes care of much of the hassle of web development, allowing you to focus on developing your app instead of reinventing the wheel. Django is a free and open-source, Python-based web framework that follows the model–template–views architectural pattern.

In the Example 1, we will design Django's Hello World program (on request, it will return a string-related web page that can be combined with CSS and JS to design your style).

First, I created a folder for this task in my Ubuntu

Using the terminal, I changed the location to the created folder

```
farjana@farjana:~/Practical_3$ cd
farjana@farjana:~$ cd Practical_3/
```

The execution of Django installation in the selected folder “Practical_3”

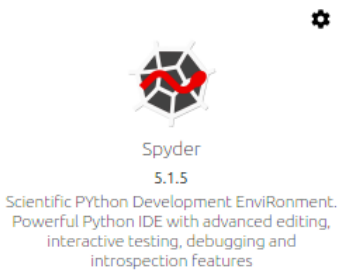
```
(base) farjana@farjana:~/Practical_3$ python -m pip install Django
Collecting Django
  Downloading Django-4.2.2-py3-none-any.whl (8.0 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 8.0/8.0 MB 5.5 MB/s eta 0:00:00
Collecting sqlparse>=0.3.1
  Downloading sqlparse-0.4.4-py3-none-any.whl (41 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 41.2/41.2 kB 2.0 MB/s eta 0:00:00
Collecting asgiref<4,>=3.6.0
  Downloading asgiref-3.7.2-py3-none-any.whl (24 kB)
Requirement already satisfied: typing-extensions>=4 in /home/farjana/anaconda3/l
es (from asgiref<4,>=3.6.0->Django) (4.5.0)
Installing collected packages: sqlparse, asgiref, Django
Successfully installed Django-4.2.2 asgiref-3.7.2 sqlparse-0.4.4
```

Confirming the installation of Django and creating a new Django project called “TASK1”

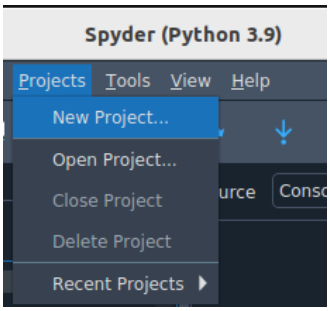
```
(base) farjana@farjana:~/Practical_3$ python -m django --version
4.2.2
(base) farjana@farjana:~/Practical_3$ django-admin startproject TASK1
```

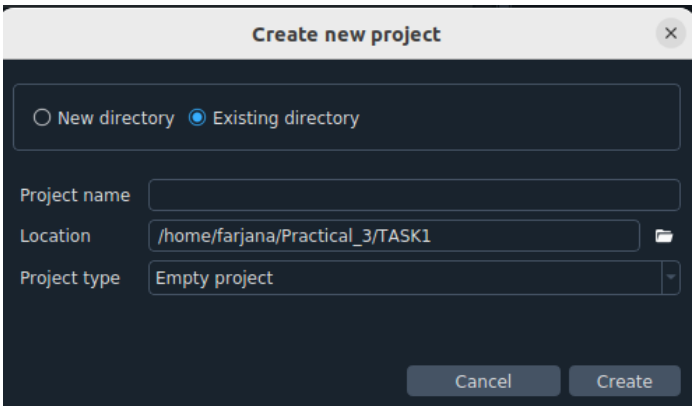
Opening Anaconda-Navigator to work in SPIDER(Python)

```
farjana@farjana:~/Practical_3$ anaconda-navigator
5-06 23:31:24,573 - WARNING linux_scaling.get_scaling_factor_using_dbus:44
detect system scaling factor settings for primary monitor.
```

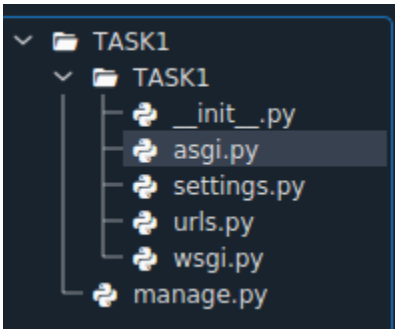


Creating a new Project while selecting the existing directory “TASK1” previously created by Django

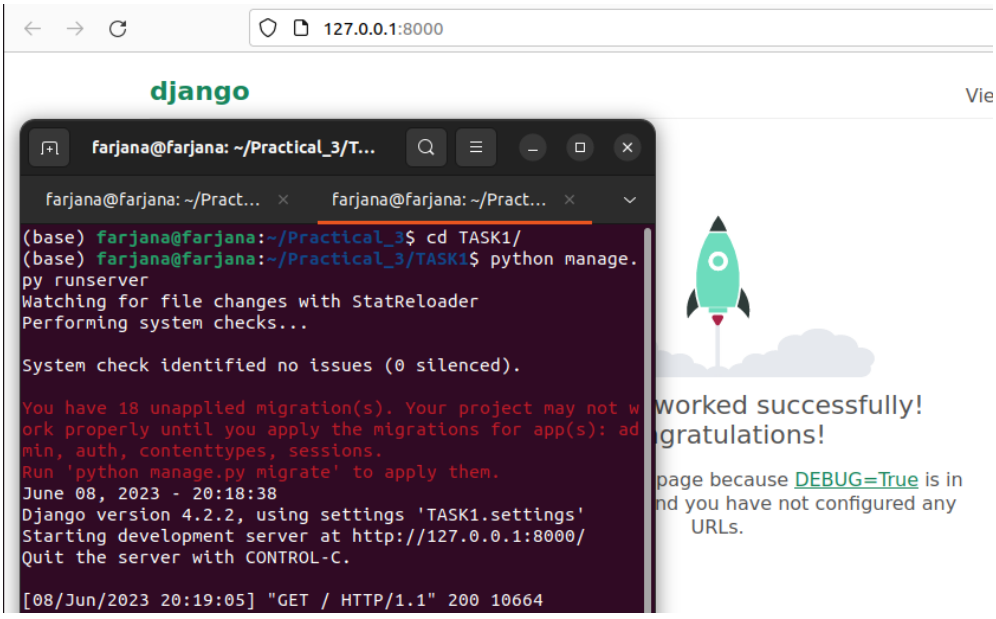




The new project structure which is created by Django environment



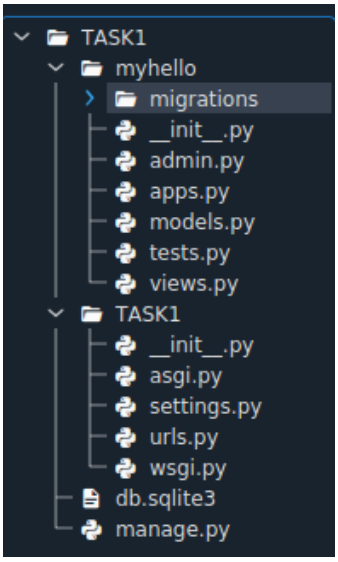
Running the server and opening the link/address.



Modifying project and creating specific application “myhello”, that will be accessible later on through 127.0.0:8000/index/

```
farjana@farjana: ~/Pr...   farjana@farjana: ~/Pr...   farjana@farjana: ~/Pr...
(base) farjana@farjana:~/Practical_3/TASK1$ python manage.py startapp myhello
(base) farjana@farjana:~/Practical_3/TASK1$
```

The aftermath in the structure after creating a new application



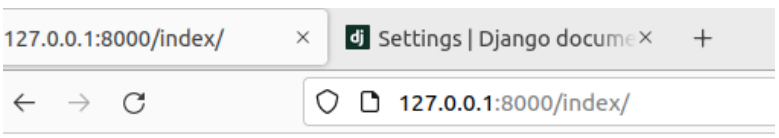
For ”myhello” to work, have to edit “urls.py” and add that in case of request of /index, the function “hello” from “views.py” will be executed.

```
16  """
17  from django.contrib import admin
18  from django.urls import path, include
19  from myhello import views
20
21  urlpatterns = [
22      path("admin/", admin.site.urls),
23      path("index/", views.hello),
24  ]
```

Setting up a function “hello” in views.py, to respond with “Hi, this is practical work 3”

```
temp.py   views.py   settings.py   urls.py
1  from django.shortcuts import HttpResponse
2  def hello(request):
3      return HttpResponse("Hi, this is practical work 3")
4  # Create your views here.
5
```

And the final result of accessing 127.0.0:8000/index/



Hi, this is practical work 3

The summary of task 1 – we created a response to a HTTP request using Django environment.

EXAMPLE 2/TASK 2

The task is to improve the example 1 and return a HTTP page instead of a string.

Django's architecture is MVT (Model-View-Template). MVT is a software design pattern used in the creation of a web application. MVT Structure is made up of three parts:

Model: The model will serve as your data's interface. It is responsible for data management. A database represents the logical data structure that supports the entire application.

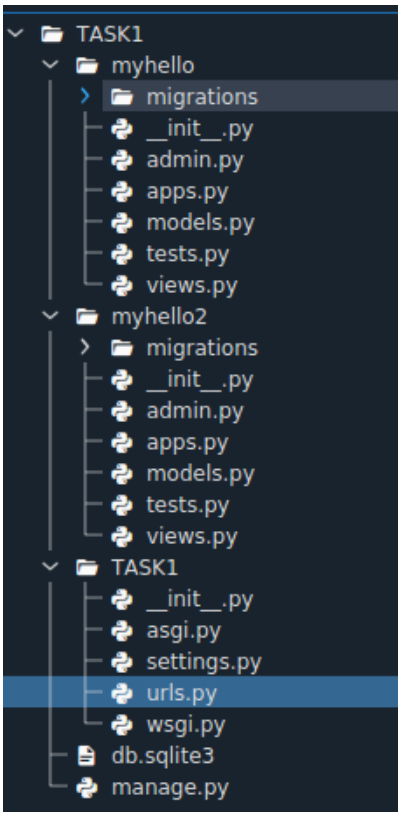
The user interface — what you see in your browser when you render a website — is referred to as the View. HTML/CSS/Javascript and Jinja files are used to represent it.

A template is made up of static sections of the desired HTML output as well as unique syntax that describes how dynamic content will be introduced.

Modifying project and creating specific application “myhello2”, that will be accessible later on through 127.0.0:8000/index2/

```
(base) farjana@farjana:~/Practical_3/TASK1$ python manage.py startapp myhello2
(base) farjana@farjana:~/Practical_3/TASK1$
```

The aftermath of the structure after creating a new application



Editing “urls.py” to include “index2”

```
urlpatterns = [
    path("admin/", admin.site.urls),
    path("index/", views.hello),
    path("index2/", include('myhello2.urls')),
]
```

Creating urls.py for index2 to open a html page I will save later

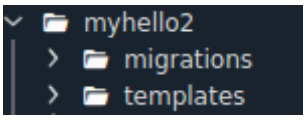
```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Jun 10 21:24:55 2023

@author: farjana
"""

from django.urls import path
from . import views
urlpatterns = [
    path('', views.hello),
]
```

Editing views.py to upon request of /index2/ to return saved/created html page.

views.py - myhello2* | views.py - myhello | settings.py

```python
from django.shortcuts import render
def hello(request):
    return render(request, 'savedhtmlpage.html')

# Create your views here.
```

Creating a **template** folder



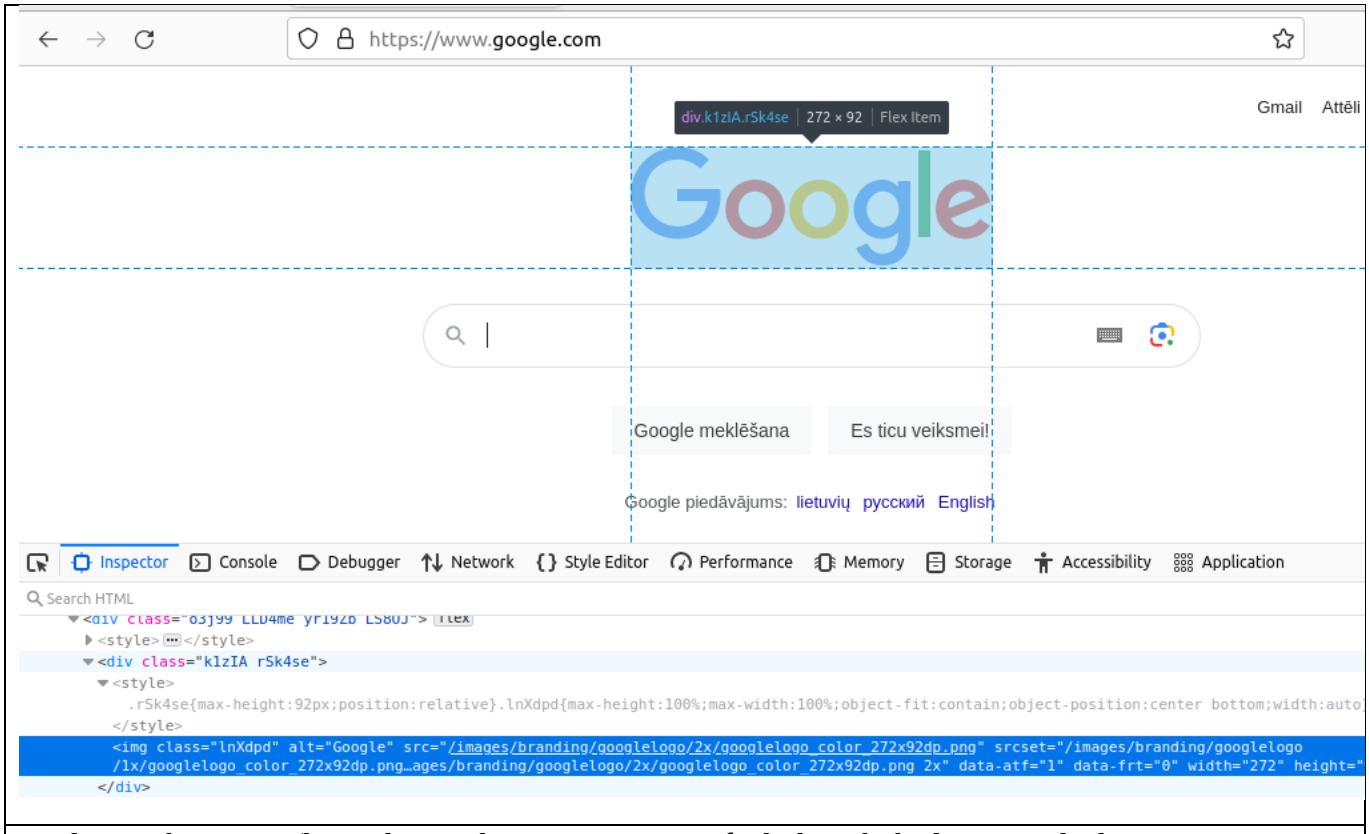
Preparing **setup.py** file by adding directory to route everything properly

/home/farjana/Practical_3/TASK1/TASK1/settings.py

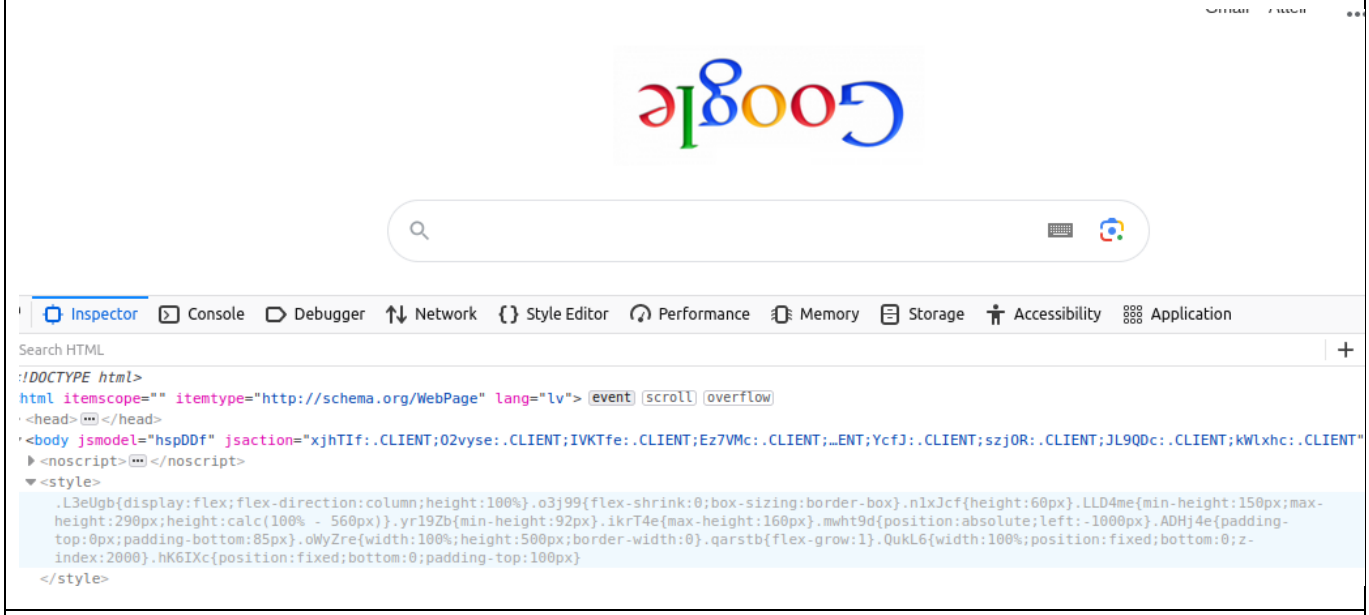
temp.py | views.py - myhello2 | views.py - myhello | settings.py

```python
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplate
        "DIRS": [os.path.join(BASE_DIR, 'myhello2/templates')],
        "APP_DIRS": True,
        "OPTIONS": {
            "context_processors": [
                "django.template.context_processors.debug",
```

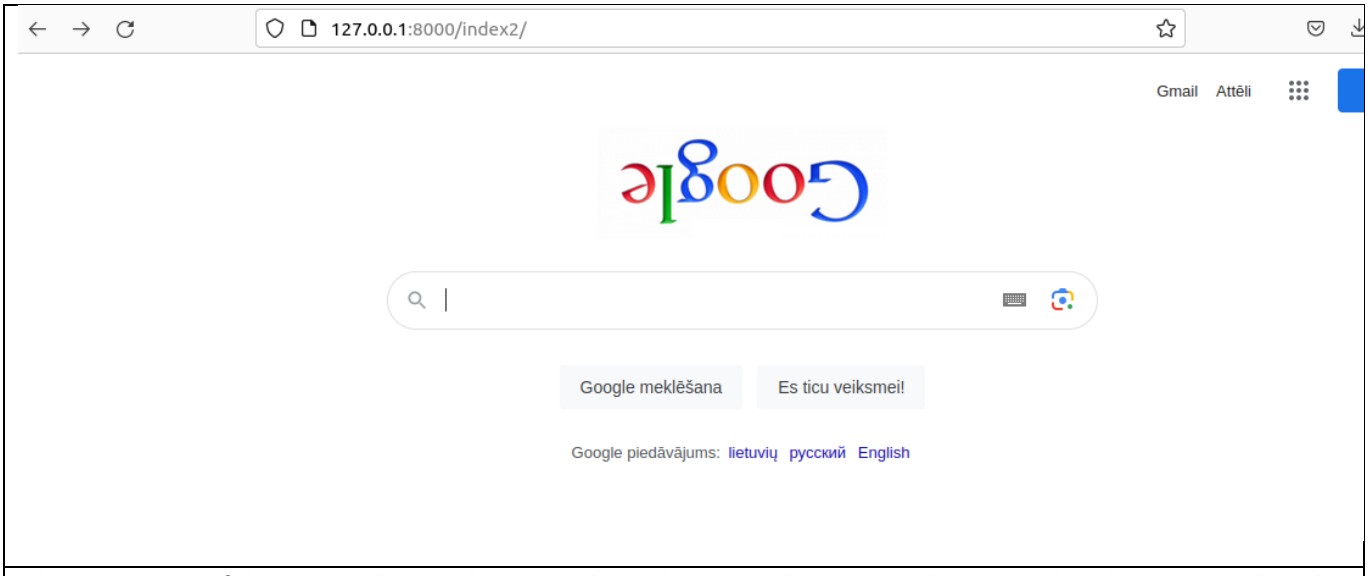
As everything in Django is now prepared, I chose google.com as my saved html page. Using the addon for Firefox browser, I saved the google.com page. To make this more individual, I inspected the elements of page and found the lines that are responsible for the logo. I decided to replace original google logo with another picture. I replaced it with upside down logo.



Replacing the picture(by replacing the picture source of a link with the logo upside down



Opening /index2



The summary of this example 2/task 2 is to show, how a web page can be created and maintained with Django environment. We studied the structure and made necessary changes to python codes to make the request of **/index2/** respond with created html page instead of a string.

EXAMPLE 3/TASK 3

Cloud message board. Basic function definition: 1, Submit message function: Users can set their own name as A, specify any name B and leave a message to B, record it as msg, and the message will be saved in the cloud. 2, Get message function: input name A, and the cloud will return the 20 latest message records.

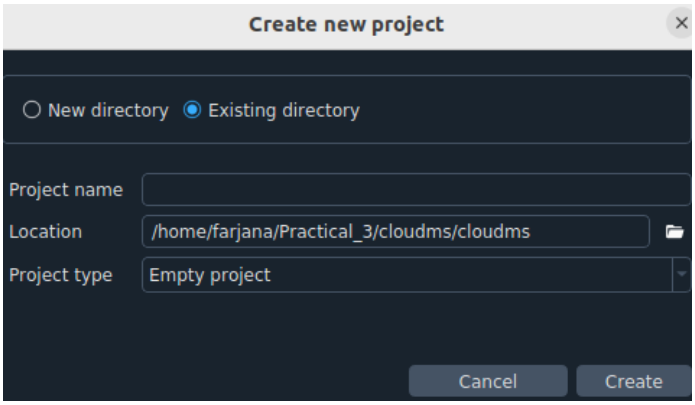
Creating a new folder “cloudms”

```
(base) farjana@farjana:~$ cd Practical_3/
(base) farjana@farjana:~/Practical_3$ ls
TASK1
(base) farjana@farjana:~/Practical_3$ mkdir cloudms
(base) farjana@farjana:~/Practical_3$ cd cloudms/
(base) farjana@farjana:~/Practical_3/cloudms$ ls
(base) farjana@farjana:~/Practical_3/cloudms$
```

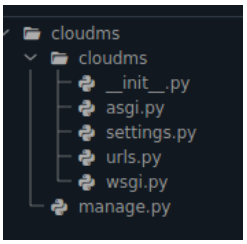
creating a new Django project called “cloudms”

```
(base) farjana@farjana:~/Practical_3/cloudms$ django-admin startproject cloudms
(base) farjana@farjana:~/Practical_3/cloudms$ ls
cloudms
```

Creating a new application in spyder



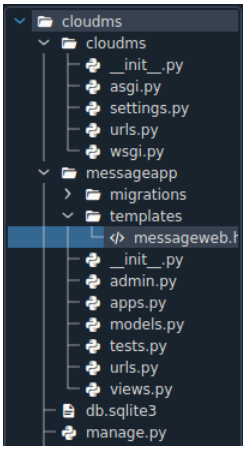
The new project structure which is created by Django environment



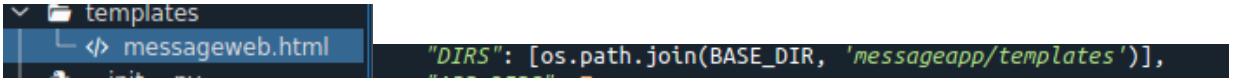
Modifying project and creating specific application “messageapp”

```
(base) farjana@farjana:~/Practical_3/cloudms/cloudms$ python manage.py startapp message
```

The aftermath of the structure after creating a new application



Creating a template folder and creating a html file which will be used as a template. In the settings section we added the directory of template.

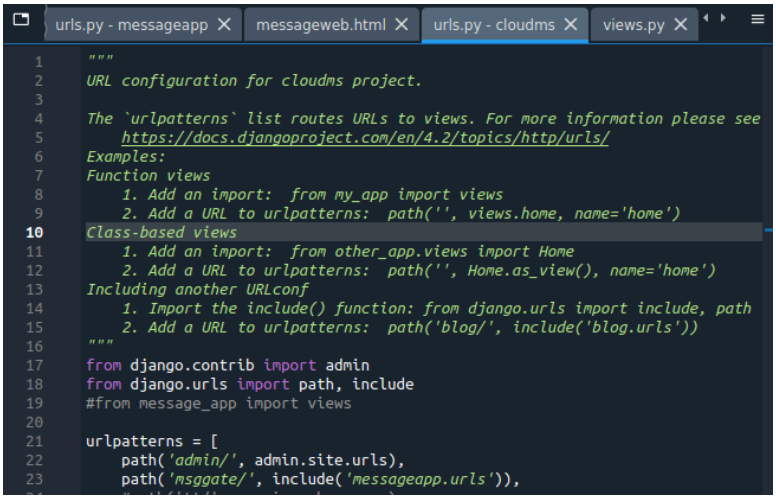


Template with custom text color and font, which will be responsible for the visual look of the page

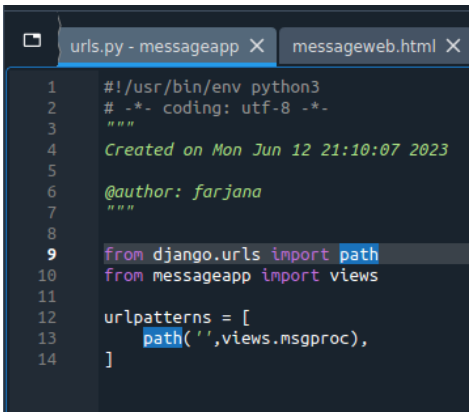
```
<!DOCTYPE html>
<html lang ='en'>
<head>
    <meta charset = 'UTF-8'>
    <title>  Cloud Message Board</title>
    </head>
<body>
    <h1><p style="font-family:'Courier New'"><font color="blue"> Leave your Message </h></p>
    <form action='/msggate/' method ='post'>
    {% csrf_token %}
    Sender <input type='text' name='userA' /> <br>
    Receiver <input type='text' name='userB' /> <br>
    Message <input type='text' name='msg' /> <br>
    <input type='submit' value='Message submission' />
    </form></font>

    <h1><p style="font-family:'Courier New'"><font color="green"> Get your Message </h1></p>
    <form action='/msggate/' method ='get'>
    Receiver <input type='text' name='userC'/> <br>
    <input type='submit' value='Get a message' />
    </form>
    <table border='1'>
    <thead>
            <th> Message Time</th>
            <th> Message Source</th>
            <th> Message content</th>
        </thead>
        <br>
        <tbody>
        {% for line in data %}
        <tr>
        <td> {{line.time}}</td>
        <td align='center'>{{ line.userA}}</td>
        <td> {{ line.msg}}</td>
        </tr>
        {% endfor%}
        </tbody>
```

```
</table>
</body></font>
</html>
```

Doing URL routing. After asscessing our defined /msggate/ the code will look into messageapp urls



Doing URL routing and creating necessary codes under the messageapp section. Letting the code know to use msgproc function from views of messageapp

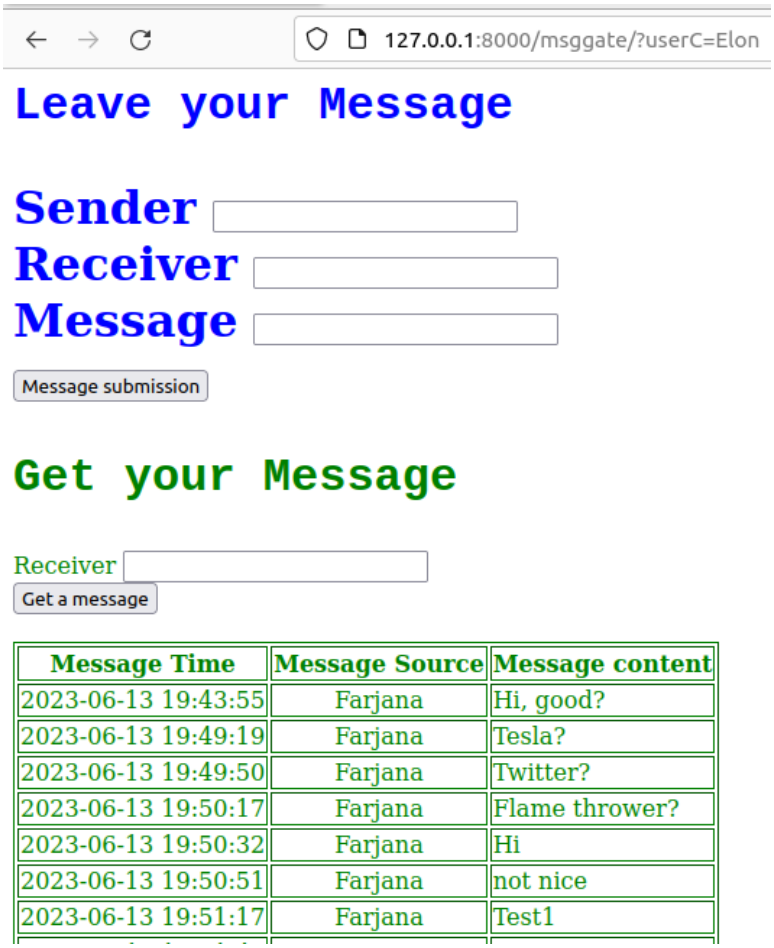


Creating views.py, which will be responsible for this messaging service, by requesting senders, messages, receivers and giving out the received messages showing the time and date for last 20 messages.

```
from django.shortcuts import render
from datetime import datetime
#from django.http import HttpResponse, JsonResponse, FileResponse
#from django.template import Template, Context
#import os

def msgproc(request):
    datalist =[]
    if request.method == 'POST':
       userA = request.POST.get( 'userA', None)
       userB = request.POST.get( 'userB', None)
       msg = request.POST.get('msg', None)
       time = datetime.now()
       with open('msgdata.txt', 'a+') as f:
           f.write('{}--{}--{}--{}--\n'.format(userB, userA, msg, time.str
    if request.method == 'GET':
        userC = request.GET.get('userC', None)
        if userC != None:
            with open('msgdata.txt','r') as f:
                cnt = 0
                for line in f:
                    linedata = line.split('--')
                    if linedata[0] == userC:
                        cnt = cnt+1
                        d = {'userA':linedata[1], 'msg':linedata[2], 'time
                        datalist.append(d)
                    if cnt >= 20:
                        break
    return render(request, 'messageweb.html', {'data':datalist})
```

The final result of accessing 127.0.0:8000/msggate/

In the template section I replaced the color of the sender part to blue and receiver part to green, I replaced the font of the text for titles.



Summary - In this EXAMPLE 3, we created a simple messaging service. We can see, that after entering the required information we can send and receive messages. And this code even includes a text database, where the 20 messages will be stored.

EXAMPLE 4/TASK 4

In this example, we will test Django's different response types, including HttpResponse class and subclasses (10 in total), JsonResponse class, StreamingHttpResponse, and FileResponse class.

First – HttpResponse

Edditing urls by adding /tt/ link for this test. Defining /tt/ to open function homeproc from views file.

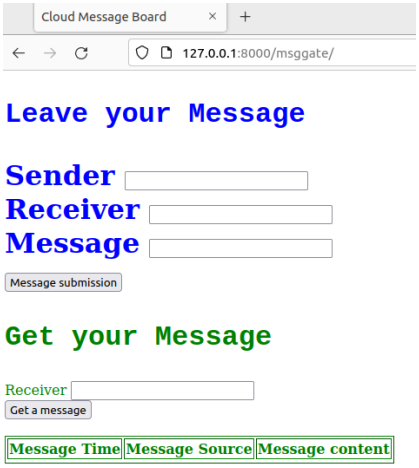
```
from django.contrib import admin
from django.urls import path, include
from messageapp import views as msgviews

urlpatterns = [
    path('admin/', admin.site.urls),
    path('msggate/', include('messageapp.urls')),
    path('tt/', msgviews.homeproc),
```

Adding homeproc function to views.py. It will give my defined response and a link to open the message app, which we created previously.

```
def homeproc(request):
    response = HttpResponse()
    response.write("<h1> This is a page for test of HttpResponse.|
                   To see the message app, visit |
                   <a href='http://127.0.0.1:8000/msggate/'> here</a></h1>")
    return response
```

The final result of accessing 127.0.0:8000/tt/

And after pressing “here”, it will lead back to /msggate/



Second – HttpResponseRedirect

In this example I made a new function in views.py *homeproc1* and routing urls.py */tt1/*. This function immediately redirects to assigned page after accessing 127.0.0:8000/tt1/ it redirects to 127.0.0:8000/msggate/. Can’t show the result with screenshot, it will just be 127.0.0:8000/msggate/ screenshot which has been showed previously.

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('msggate/', include('messageapp.urls')),
    path('tt/', msgviews.homeproc),
    path('tt1/', msgviews.homeproc1),
    #path('', msgviews.pgproc),
]
```

```
def homeproc1(request):
    response = HttpResponseRedirect('http://127.0.0.1:8000/msggate/')
    return response
```

Third – HttpResponsePermanentRedirect

This example will do the same thing as HttpResponseRedirect, but the only difference can be seen in the terminal, that this request will have a code 301, while HttpResponseRedirect 302

```
[14/Jun/2023 19:40:31] "GET /tt2/ HTTP/1.1" 301 0
[14/Jun/2023 19:40:31] "GET /msggate/ HTTP/1.1" 200 1155
[14/Jun/2023 19:40:41] "GET /tt1/ HTTP/1.1" 302 0
[14/Jun/2023 19:40:41] "GET /msggate/ HTTP/1.1" 200 1155
```

The code from views.py

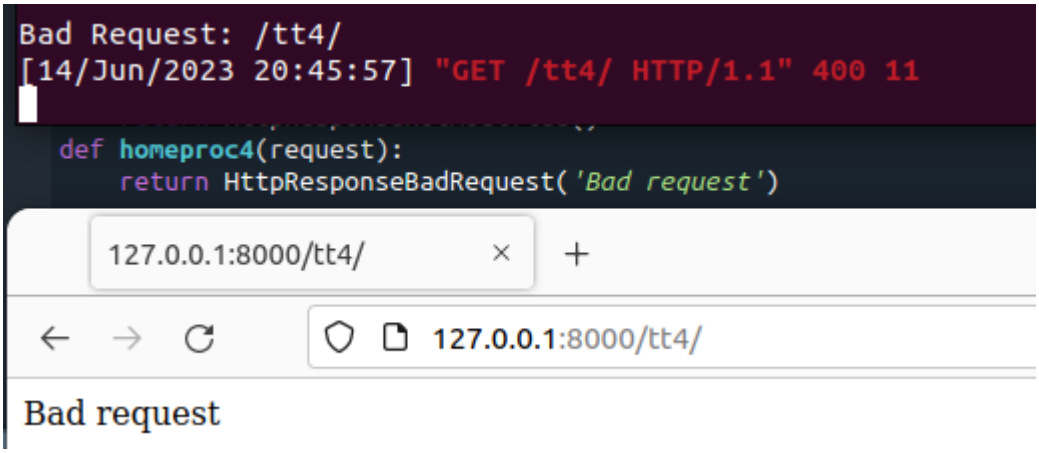
```
def homeproc2(request):
    response = HttpResponsePermanentRedirect('http://127.0.0.1:8000/msggat
    return response
```

Fourth- HttpResponseNotModified

The constructor doesn't take any arguments and no content should be added to this response. Use this to designate that a page hasn't been modified since the user's last request (status code 304).

Fifth – HttpResponseBadRequest

This code will give the HTTP status code 400.



Pictures of the server HTTP code(400), then the code from views.py and what we get when access /tt4/ and function from views.py

The same structure with other HTTP response subclasses- we will just get a different status code in the terminal from server

HttpResponseForbidden (Forbidden, HTTP status code is 403)

HttpResponseNotAllowed (not allowed, HTTP status code 405)

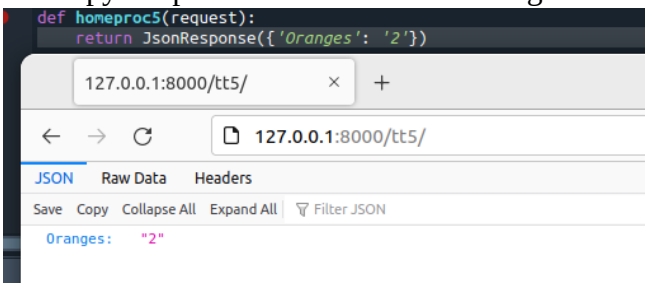
HttpResponseGone (HTTP status code is 410)

HttpResponseServerError (server error with HTTP status code 500)

HttpResponseNotFound (404 error, HTTP status code is 404)

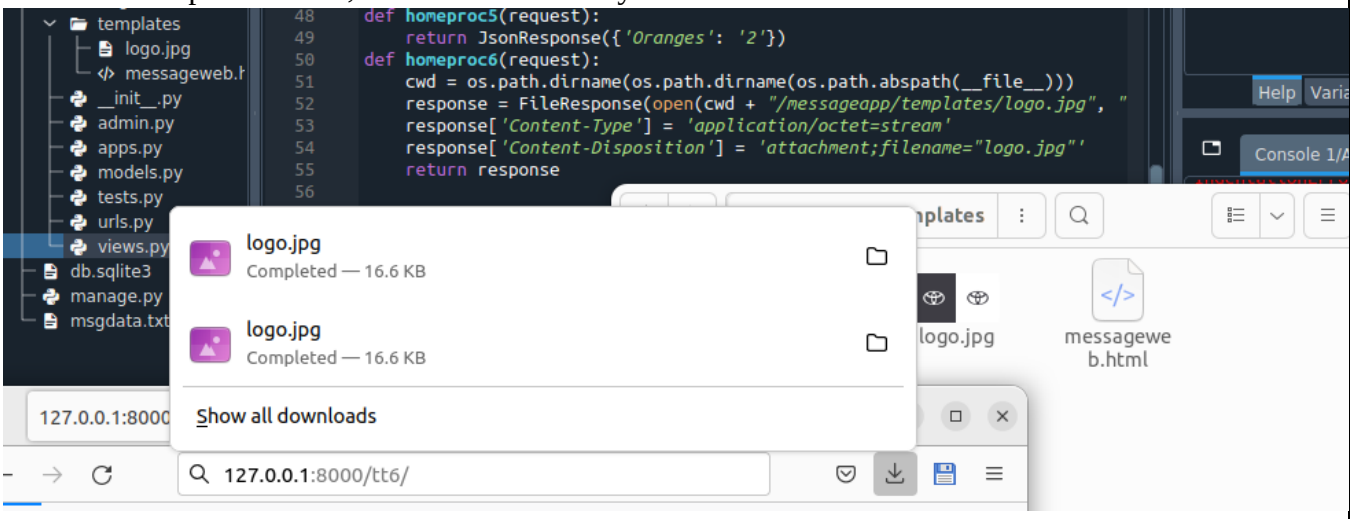
JsonResponse

JavaScript Object Notation (JSON) is a standard text-based format for representing structured data based on JavaScript object syntax. It is commonly used for transmitting data in web applications (e.g., sending some data from the server to the client, so it can be displayed on a web page, or vice versa).
views.py script and the result of accessing it.

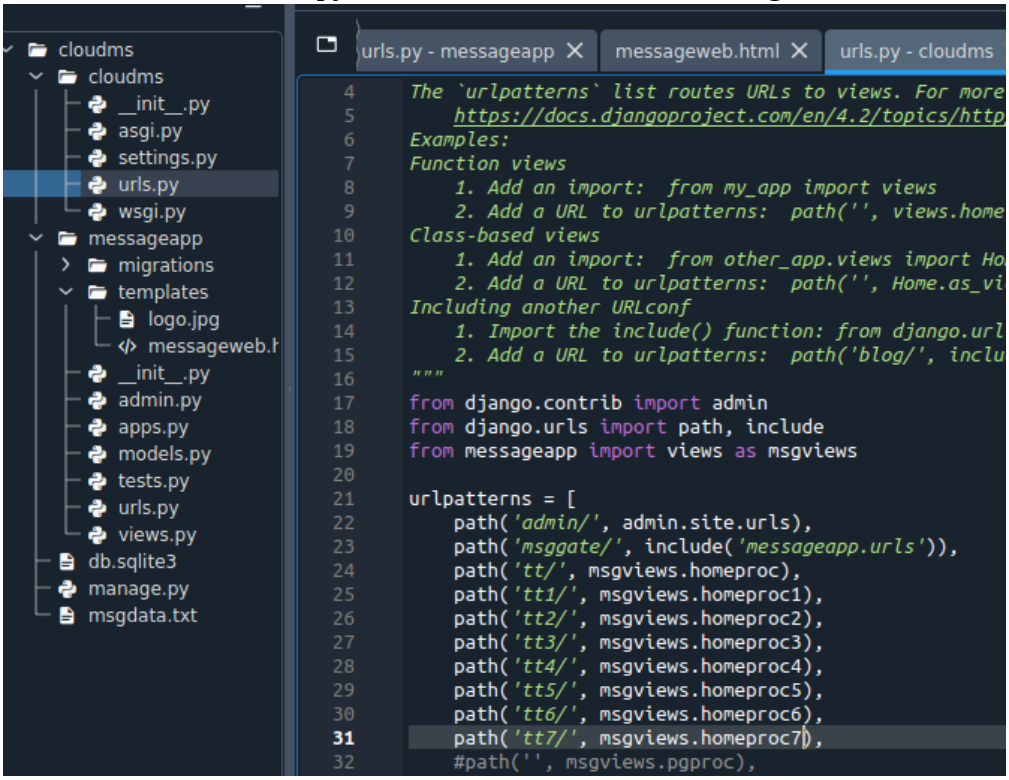


In this case, it's basically used for data representation.

FileResponse- in this scenario – after opening 127.0.0:8000/tt6/ a file that's previously downloaded and moved to /templates folder, will be automatically downloaded from the 127.0.0:8000/tt6/



And this is how the urls.py looked like after all of the coding



All of the examples can be still accessed separately as I defined different functions for each of the links

Summary of Example 4. In this example, we tested Django's different response types, including HttpResponse class and some of the subclasses (there are 10 in total), JsonResponse class, and FileResponse class. I showed the required python codes and all the outputs after requesting sites.