Python with Django
(UNIT- IV & V)

by
Dr. Partha Roy,
Associate Professor,
Bhilai Institute of Technology, Durg

UNIT- IV

Django-I

Introduction to Django Web Framework: Web development, Introduction to Django Web Framework, Features of Django, Installing Django, MVC model, HTTP concepts, Views, URL Mapping.

UNIT- V

Django-II

Django Template Language, Utilities of Templates, Creating Template Objects, Tags, Variables and Filters, Rendering Templates, Template Inheritance, Form Handling, Form validation and Error Messages, Form Display.

Installing Django

- Python 3 or above should be pre-installed.
- To install Django we use the following command in command prompt:
  - **pip install django**
- To view all the installed modules (packages) in the system we use the following command in command prompt:
  - **pip freeze**

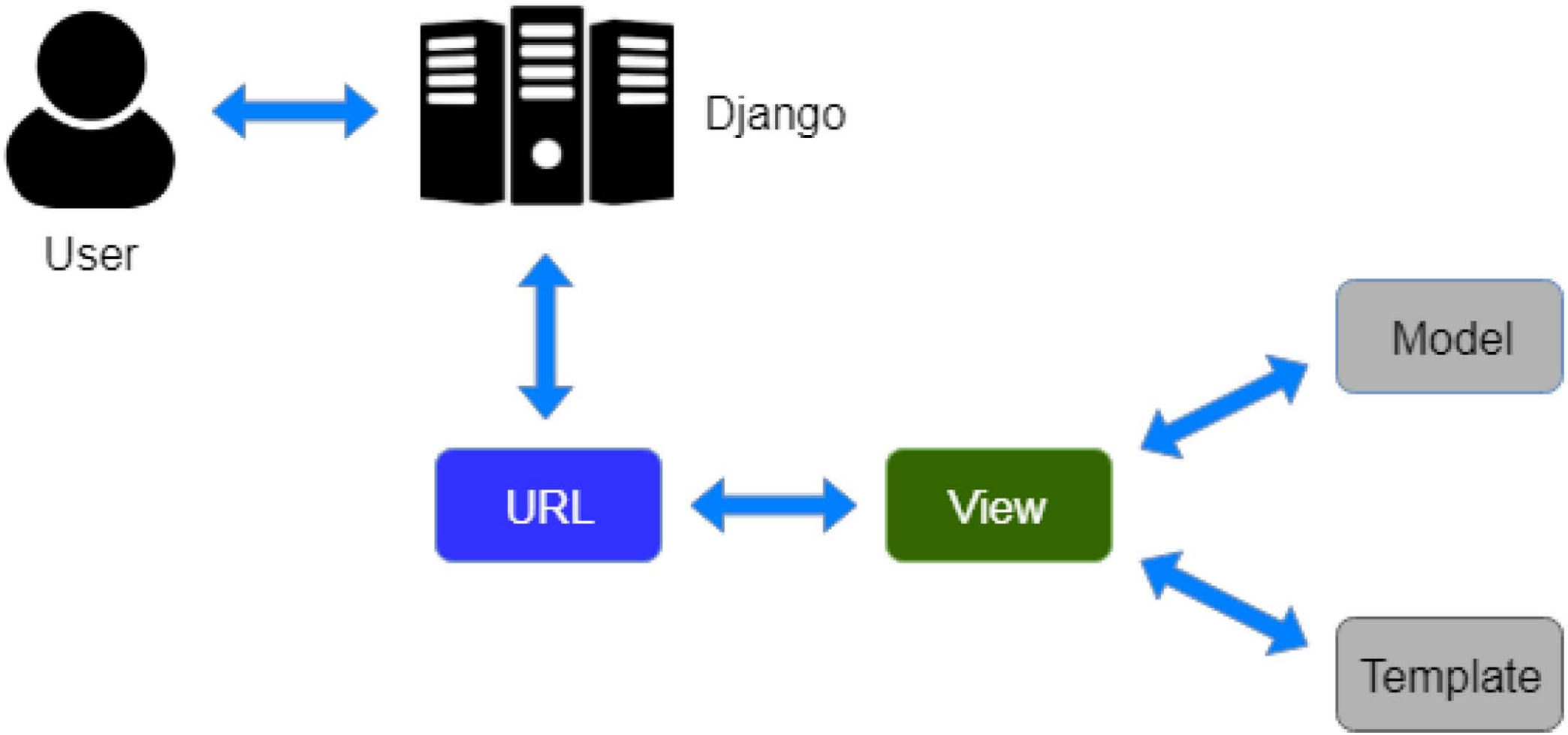
About Django

- Django is a Python framework that makes it easier to create web sites using Python.
- Django emphasizes reusability of components, also referred to as DRY (Don't Repeat Yourself), and comes with ready-to-use features like login system, database connection and **CRUD** operations (**C**reate **R**ead **U**pdate **D**elete).
- Django is especially helpful for database driven websites.

MVT (Model View Template)

- Django follows the MVT design pattern (**M**odel **V**iew **T**emplate).
- **Model** - The data we want to present, usually data from a database.
- **View** - A request handler that returns the relevant template and content - based on the request from the user.
- **Template** - A text file (like an HTML file) containing the layout of the web page, with logic on how to display the data.

MVT (Model View Template)



MVT (Model View Template)

- A user requests for a resource to the Django, **Django** works as a **controller** and check to the available resource in URL.
- If URL maps, a **view** is called that interact with **model** and **template**, it renders a **template**.
- **Django** responds back to the user and sends a **template as a response**.

Model

- *The models are usually located in a file called* ***models.py***
- The model provides data from the database.
- In Django, the data is delivered as an **O**bject **R**elational **M**apping (**ORM**), which is a technique designed to make it easier to work with databases.
  - The most common way to extract data from a database is SQL. One problem with SQL is that we have to have a pretty good understanding of the database structure to be able to work with it.
- Django, with **ORM**, makes it easier to communicate with the database, *without having to write complex SQL statements*.

View

- *The views are usually located in a file called* ***views.py***
- A view is a function or method that
  - takes **http requests** as arguments,
  - imports the relevant model(s), and finds out what data to send to the template,
  - and returns the final result.

Template

- *The templates of an application are located in a **folder** named **templates.***
- Here we describe how the result should be represented.
- Templates are often **.html** files, with HTML code describing the layout of a web page, but it can also be in other file formats to present other results, but we will concentrate on **.html** files.
- Django uses standard HTML to describe the layout, but uses Django tags to add logic

URLs

- Django also provides a way to navigate around the different pages in a website.
- When a user requests a URL, Django decides which view it will send it to.
- This is done in a file called **urls.py**

The working algorithm of Django

- When we provide a URL to Django then it performs the following operations:
  - Django receives the URL, checks the **urls.py** file, and calls the view that matches the URL.
  - The view, located in **views.py**, checks for relevant models.
  - The models are imported from the **models.py** file.
  - The view then sends the data to a specified template in the **template folder**.
  - The template contains HTML and Django tags, and with the data it returns finished HTML content back to the browser.

Creating a Django project

- First we need to create a folder in which we want the Django project to exist.
- Then we open the command prompt from the same folder.
- In the command prompt we type the following command:
  - **django-admin startproject MyProject1**
  - *Here the name of the project will become "MyProject1", but we can give any name we want.*
- It will create a folder named *MyProject1* and create some files and folders in it.
- It also creates a very important file named **manage.py** in the project folder.

Running the Django project

- After Django has created the folder named *MyProject1* and a file named **manage.py**, we do the following activities:
  - Go inside the project folder *MyProject1*
  - Then we open the command prompt from the project folder and write the following command:
    - **python manage.py runserver**

Running the Django project

- We would get the following o/p:

```
PS D:\PYTHON PRACTICE\DJango Projects\roy2\myproject1> python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes
, sessions.
Run 'python manage.py migrate' to apply them.
September 10, 2022 - 22:01:59
Django version 4.1.1, using settings 'MyProject1.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

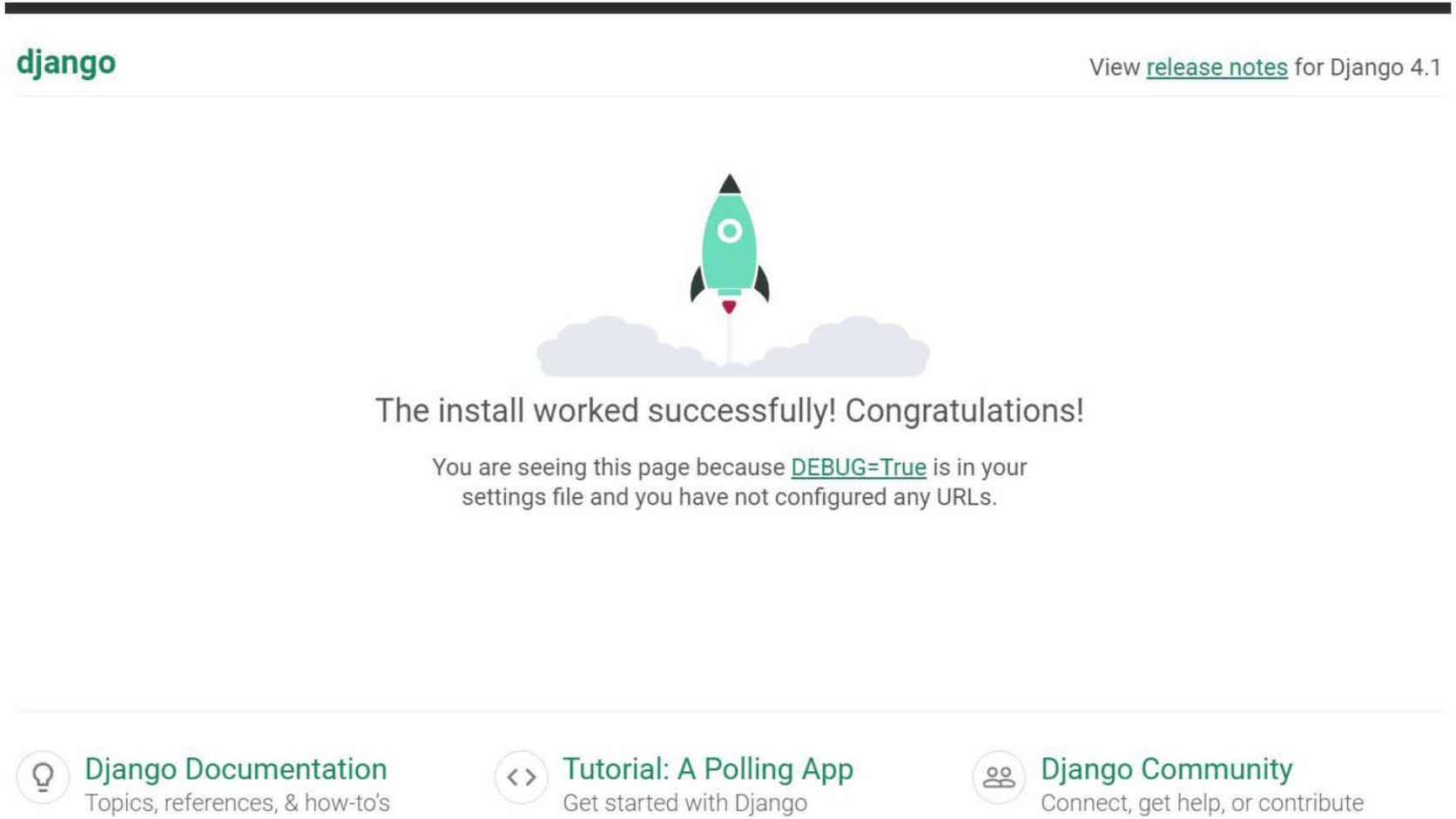
Running the Django project

```
PS D:\PYTHON PRACTICE\DJango Projects\roy2\myproject1> python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes
, sessions.
Run 'python manage.py migrate' to apply them.
September 10, 2022 - 22:01:59
Django version 4.1.1, using settings 'MyProject1.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

- Then we type in the web-browser the project URL mentioned in the output: **http://127.0.0.1:8000/**

Running the Django project

- The following output proves that our project is successfully running:



The App folder

- The word “App” is the short form of Web Application.
- Inside project folder “MyProject1” we can see another folder with the same name “MyProject1”, this folder is the App created as default by Django.
- Inside the app folder there are various prebuilt files out of which **settings.py** is the most important and we need it for setting various properties of the web application.

Creating important folders

- In the project folder "MyProject1" we need to create the following three folders:
  - **templates** : It is used to put all the HTML files related to the web app.
  - **static** : It is used to put all the static files related to the web app i.e. javascript, CSS, images font files, etc..
  - **media** : It will be used to store all the dynamically uploaded contents by the user, like, image, pdfs, doc files etc while using the web app.

Migration

- In order to perform administrative operations on the web application we need to perform the process of migration.
- The migration process causes Django to build the basic database schema and database for the web app.
- The default database engine used by Django is **SQLite**
- A file names **db.sqlite3** is created by default in the project directory when we create the project.

Migration

- For performing the migration process we need to open command prompt from the project folder "MyProject1" and run the following command:
  - **python manage.py migrate**
- We will get the following output:

```
PS D:\PYTHON PRACTICE\DJango Projects\ROY2\MyProject1> python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
```

SQLite database

- After running the migrate command the SQLite database **db.sqlite3** gets populated with necessary tables that are required for the web app.
- After migration process we also get access to the login credentials for the **admin panel**, from where we can perform user administration for the web app.
- To view the contents of **db.sqlite3** file we need to install **DB Browser** software in our system.
- We can download it for free from **https://download.sqlitebrowser.org/DB.Browser.for.SQLite-3.12.2-win64.msi**

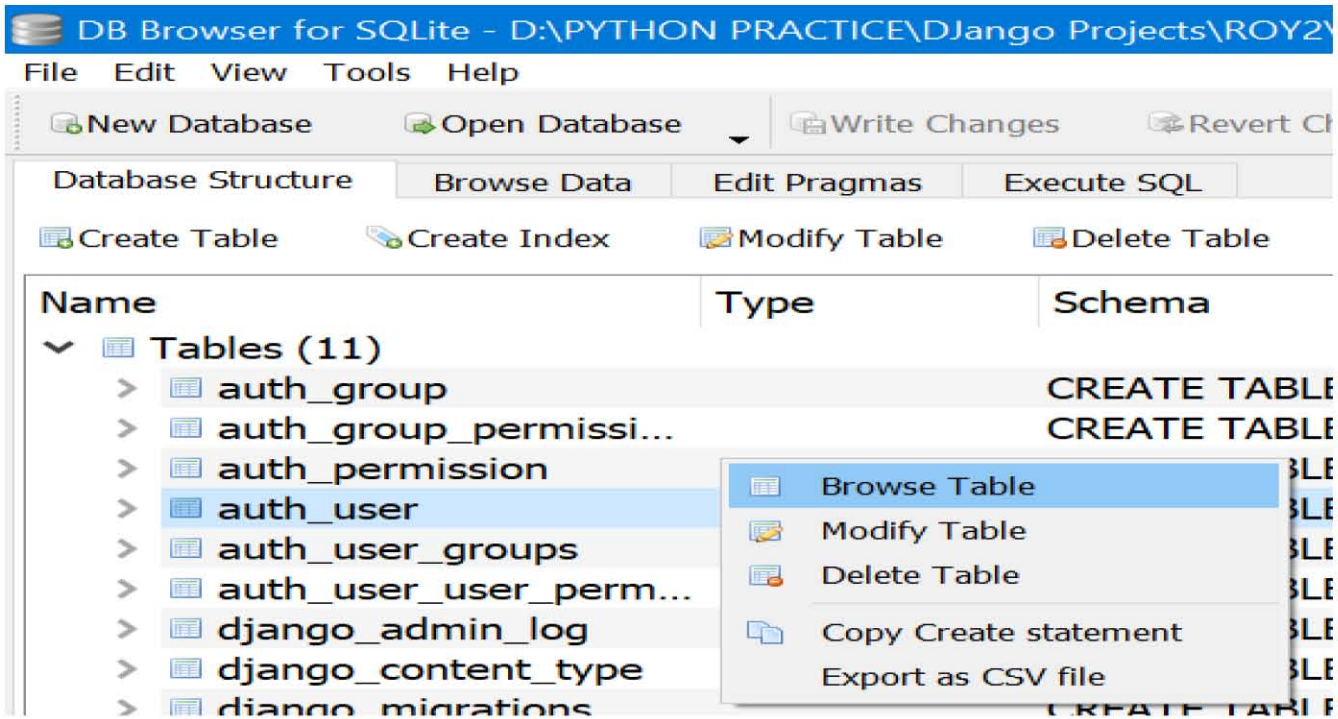
Creating Super-user

- From the project folder we need to open the command prompt and type the following command to create the super user:
  - **python manage.py createsuperuser**
- This command will ask us to input the following information:

```
PS D:\PYTHON PRACTICE\DJango Projects\ROY2\MyProject1> python manage.py createsuperuser
Username (leave blank to use 'admin'):
Email address: admin@admin.com
Password:
Password (again):
Error: Your passwords didn't match.
Password:
Password (again):
The password is too similar to the username.
This password is too short. It must contain at least 8 characters.
This password is too common.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
PS D:\PYTHON PRACTICE\DJango Projects\ROY2\MyProject1>
```

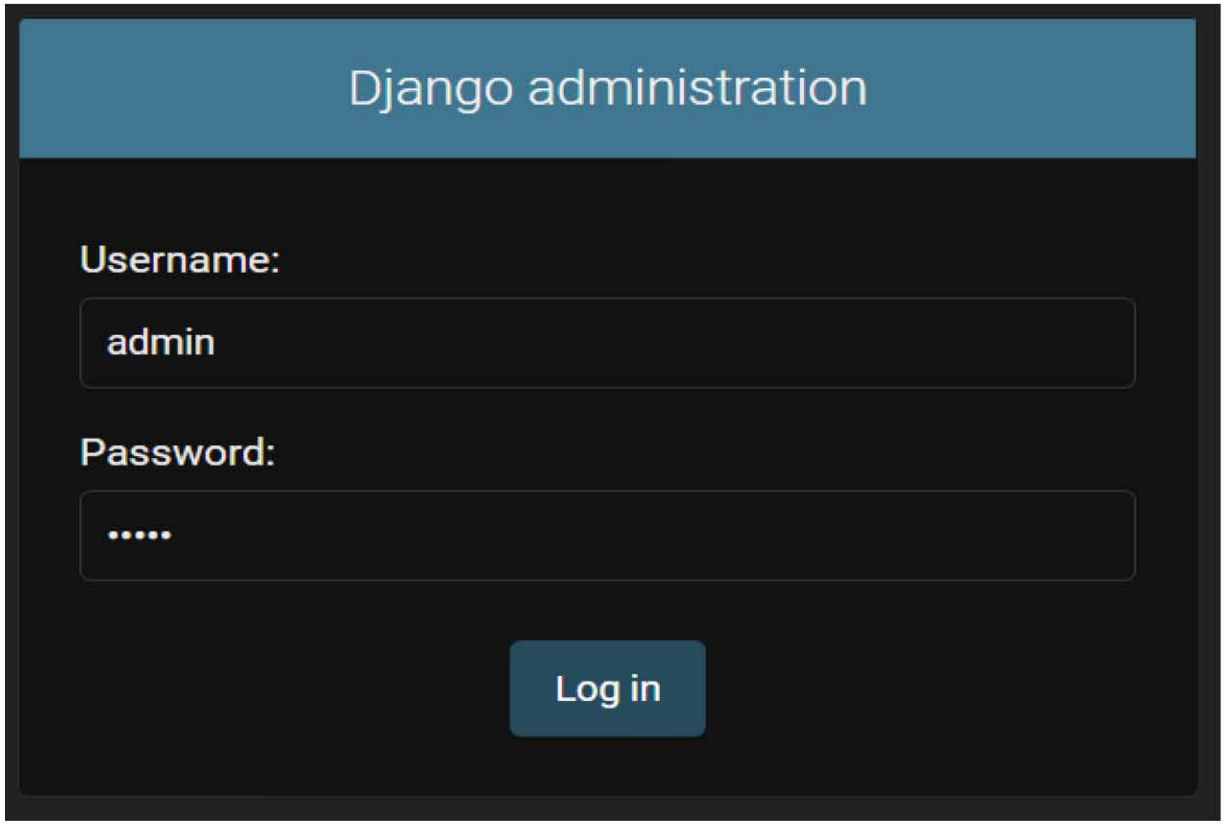
Creating Super-user

- Following are the login credentials for our project:
  - Username: admin
  - Password: admin
- The table in which the user credentials are stored is **auth_user**



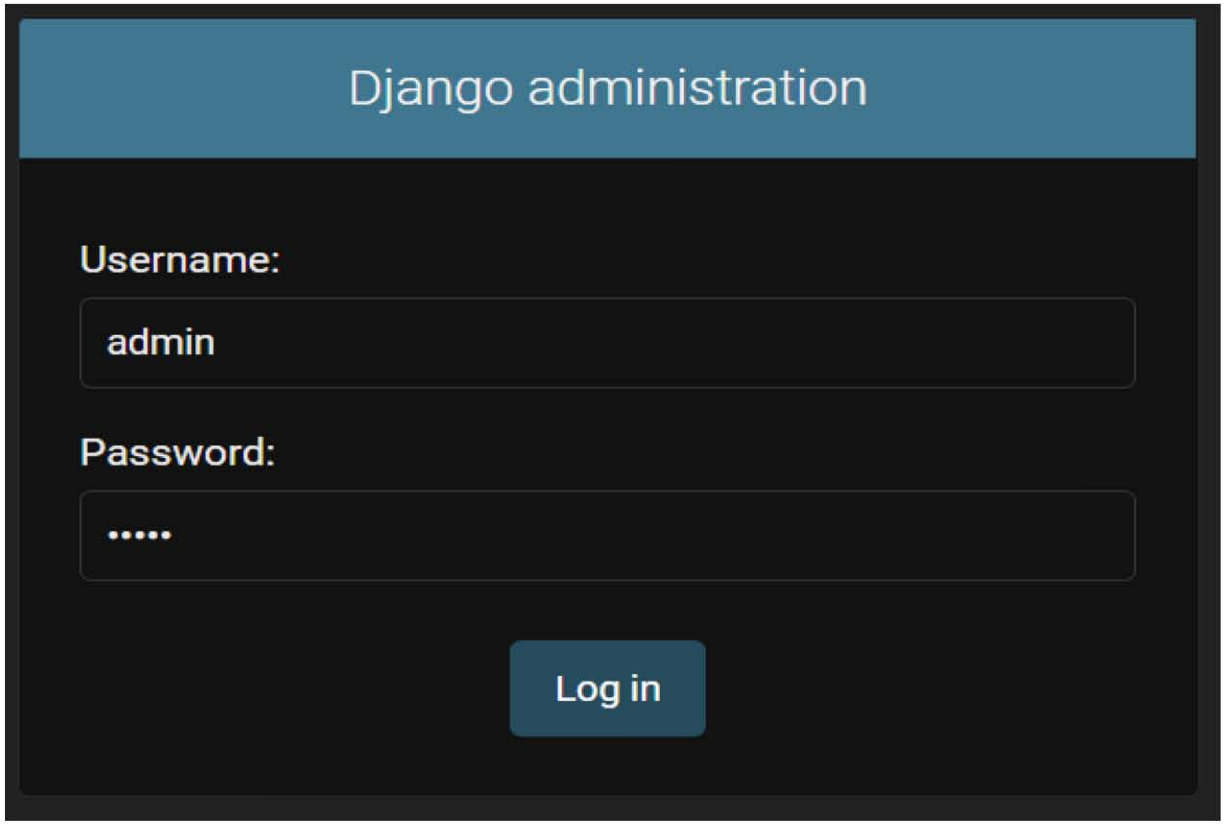
Entering the Admin Panel

- After running the project we open the following link for opening the admin panel:
  - **http://127.0.0.1:8000/admin**

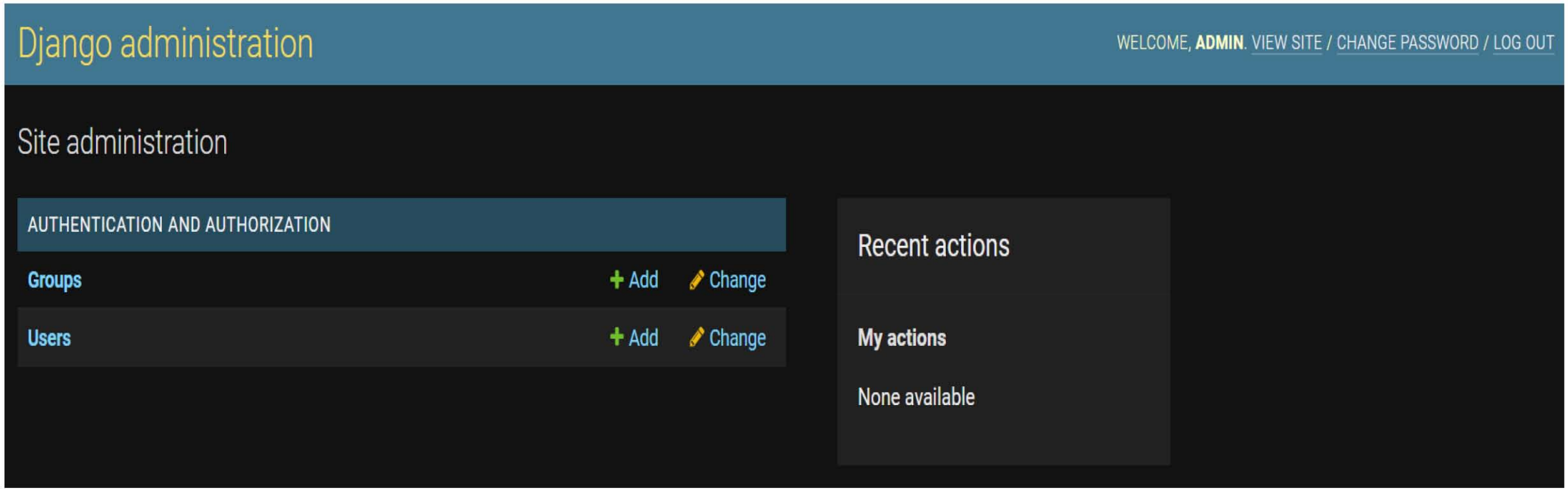


Entering the Admin Panel

- After running the project we open the following link for opening the admin panel:
  - **http://127.0.0.1:8000/admin**



Admin Panel



URLs and Views

- URL is **U**niform **R**esource **L**ocator and they are used to navigate through pages present in the web application.
- They are like routes through which we can navigate through the web app.
- It is visible in the browser's address bar.
- View help us to create contents dynamically when a particular route is selected through the URL.
- Views are functions defined in **views.py** file.
- Every function in **views.py** is associated with a URL.
- The mapping of URL and its corresponding response is configured in **urls.py** file.

URLs and Views

- When a web request is made to a Django application, it is the job of the **urls.py** file to determine what to do with that request.
- This is what is known as **routing** in a web application.
- So we might have URLs set up in our project like **/contact, /stats, /about-us** and so on.
- Each of those routes will trigger a different function in the **views.py** file.
- Each different function will perform a different operation depending on the route the user has chosen via the web browser.

Creating URLs and Views

- We need to get inside the web app folder "MyProject1" which is inside the project folder "MyProject1".
- Right click in the web app folder "MyProject1" and create a new file and name it as **views.py**

views.py

- Inside **views.py** we need to write the following code:

```
from django.http import HttpResponse
```

- The HttpResponse will help us to write text on the client's browser screen.
- Then we create the following function that will be used as a response to a URL request:

```
def f1(request):
    msg="<h1>Welcome to DJango</h1>"
    return HttpResponse(msg)
```

- Next we need to open the **urls.py** file and perform the mapping of url with response function.

views.py

- The **views.py** would look like this:

```
1   from django.http import HttpResponse
2
3
4   def f1(request):
5       msg="<h1>Welcome to DJango</h1>"
6       return HttpResponse(msg)
```

urls.py

- Inside **urls.py** we need to add the following code at the top:

```
from MyProject1 import views
```

- Then we need to add the following code to **urlpatterns** property:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('test/',views.f1)
]
```

- The first element in the list **path('admin/', admin.site.urls)** is available by default, it means that if we write the URL **http://127.0.0.1:8000/admin** then from admin.site.urls the admin panel will open.
- We need to add the second element i.e. **path('test/',views.f1).** It means that if we write the URL **http://127.0.0.1:8000/test** then a new page will open with the message returned from **f1()** in **views.py** file.

urls.py

- The **urls.py** would look like this:

```
16    from django.contrib import admin
17    from django.urls import path
18    from MyProject1 import views
19    urlpatterns = [
20        path('admin/', admin.site.urls),
21        path('test/',views.f1),
```

urls.py (sec-A)

- We will get the following output in the web browser:

Dynamic Routes or Dynamic URLs

- Dynamic routing is done to keep the page as it is but changing its contents dynamically according to the dynamic URL
- There are four types of data that we can add in a URL to make dynamic URLs or routes:
  - **Int** : integer type data
  - **Str** : string type data
  - **Slug** : a string with hyphen (-) in between, e.g. “about-us”
  - **no type** : if we do not specify any type then python will assume its type according to the value used
- On the basis of any of the three types of data dynamic pages can be displayed on the client’s web browser.
- The value that we provide can be used in the mapped function code to generate a customized message back to the browser.

Dynamic Routing

- Sample: **path('test/<int:var_name>',views.f2)** or
  path('test/<str: var_name>',views.f2) or
  path('test/<int: var_name>',views.f2) or
  path('test/<var_name>',views.f2)
- Here, **var_name** will be passed to **f2()** along with the request parameter, so now we will have two parameters in **f2** function: **f2(request,var_name)**
- Inside **views.py** we can write the following code for **f2()**

```
def f2(request,var_name):
    msg="<h1>Value = {} </h1>".format(var_name)
    return HttpResponse(msg)
```

- The variable **msg** would contain the dynamic response string that will be sent to the client's browser.

Dynamic Routing (views.py)

- The "**views.py**" would look as follows:

```
1    from django.http import HttpResponse
2
3
4    def f1(request):
5        msg="<h1>Welcome to DJango</h1>"
6        return HttpResponse(msg)
7    def f2(request,n):
8        msg="<h1>Value = {} </h1>".format(n)
9        return HttpResponse(msg)
```

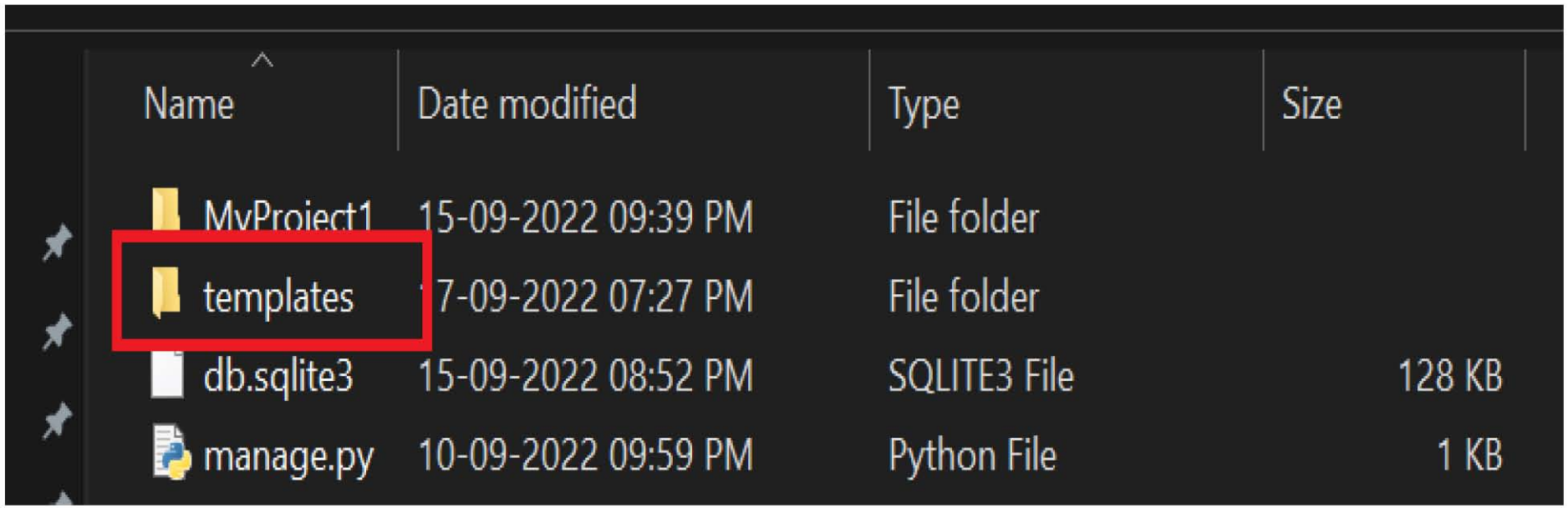
Dynamic Routing (urls.py)

- The "**urls.py**" would look as follows:

```
16  from django.contrib import admin
17  from django.urls import path
18  from MyProject1 import views
19
20  urlpatterns = [
21      path('admin/', admin.site.urls),
22      path('test/',views.f1),
23      path('test/<int:n>',views.f2),
24      path('test/<str:n>',views.f2),
25      path('test/<slug:n>',views.f2),
26      path('test/<n>',views.f2),
27
28  ]
```

Rendering HTML template as response

- To keep the HTML pages we need to create a folder named **templates** in the project folder "MyProject1" where there are two files already present with the names "manage.py" and "db.sqlite3"



Rendering HTML template as response

- Inside the "**templates**" folder we need to create the HTML pages, for example we create an html page with the name "**Test1.html**" and it has the following contents:

```
<html>
   <body>
      <h1>This is an HTML page named Test.html</h1>
   </body>
</html>
```

Rendering HTML template as response

- Now inside the "**settings.py**" file there is a list declared with the name "**TEMPLATES**", inside it is a dictionary and inside that dictionary there is a key named as "**DIRS:[]**", in the value part we put the name of the folder/directory which will hold the templates i.e. HTML pages of our web app.
- So in "**DIRS:[]**" the value part will be **[BASE_DIR, "templates"]**
- Finally it will look like follows:
  - **DIRS: [BASE_DIR, "templates"]**
- This only makes sure that Django will search for the customized html pages in the "templates" directory, we can name it anything else also.

Rendering HTML template as response (settings.py)

- The "**settings.py**" file content will look like this:

```
53
54  TEMPLATES = [
55      {
56          'BACKEND': 'django.template.backends.django.DjangoTempla
57          'DIRS': [BASE_DIR,"templates"],
58          'APP_DIRS': True,
59          'OPTIONS': {
60              'context_processors': [
61                  'django.template.context_processors.debug',
62                  'django.template.context_processors.request',
63                  'django.contrib.auth.context_processors.auth',
64                  'django.contrib.messages.context_processors.mess
65              ],
66          },
67      },
68  ]
```

Rendering HTML template as response

- Now inside "**views.py**" we need to put the following code:

```
from django.shortcuts import render
def f3(request):
    return render(request,"Test1.html")
```

- Here, "f3" is the name of the function that will be invoked and it will return (request object and "Test1.html") due to which "Test1.html" will be rendered in the client's machine.

Rendering HTML template as response (views.py)

- Inside "**views.py**" the code will look as follows:

```
1   from django.http import HttpResponse
2   from django.shortcuts import render
3
4   def f1(request):
5       msg="<h1>Welcome to DJango</h1>"
6       return HttpResponse(msg)
7   def f2(request,n):
8       msg="<h1>Value = {} </h1>".format(n)
9       return HttpResponse(msg)
10  def f3(request):
11      return render(request,"Test1.html")
12
```

Rendering HTML template as response

- Now inside "**urls.py**" we need to put the following code:

  path('',views.f3)

- Putting ***empty quotes*** represents ***home page***, so when any user types the following url for our web app: **http://127.0.0.1:8000/** then the page mentioned in **f3(request)** inside **views.py** will be rendered.

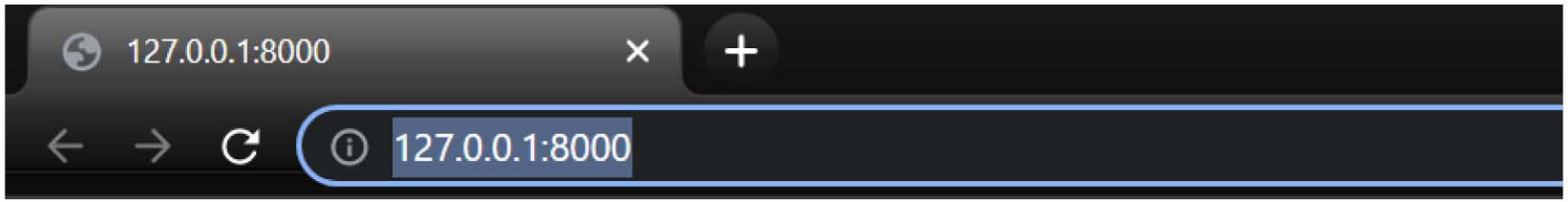
Rendering HTML template as response (urls.py)

- The "**urls.py**" would look as follows:

```
16  from django.contrib import admin
17  from django.urls import path
18  from MyProject1 import views
19
20  urlpatterns = [
21      path('admin/', admin.site.urls),
22      path('test/',views.f1),
23      path('test/<int:n>',views.f2),
24      path('test/<str:n>',views.f2),
25      path('test/<slug:n>',views.f2),
26      path('test/<n>',views.f2),
27      path('',views.f3)
28  ]
```

Rendering HTML template as response

- When any use types the url **http://127.0.0.1:8000/** then we will get the following output:



This is an HTML page named Test.html

Django Template Variables:
Passing data from view to a template

➢In "**views.py**" we need to write the following code:

```
def f3(request):
    mymessage={
        "m1":"Welcome to Django",
        "m2":"This is a message from views.py"
    }
    return render(request, "Test1.html", mymessage)
```

Here, "**mymessage**" is a dictionary that contains variables as keys whose values will be displayed in the html file "**Test1.html**" when we use **{{m1}} or {{m2}}** in the html file.

Django Template Variables: Passing data from view to a template

➢The "**views.py**" will look like this:

```
1   from django.http import HttpResponse
2   from django.shortcuts import render
3
4   def f1(request):
5       msg="<h1>Welcome to DJango</h1>"
6       return HttpResponse(msg)
7   def f2(request,n):
8       msg="<h1>Value = {} </h1>".format(n)
9       return HttpResponse(msg)
10  def f3(request):
11      mymessage={
12          "m1":"Welcome to Django",
13          "m2":"This is a message from views.py"
14      }
15      return render(request,"Test1.html",mymessage)
16
```

Django Template Variables: Passing data from view to a template

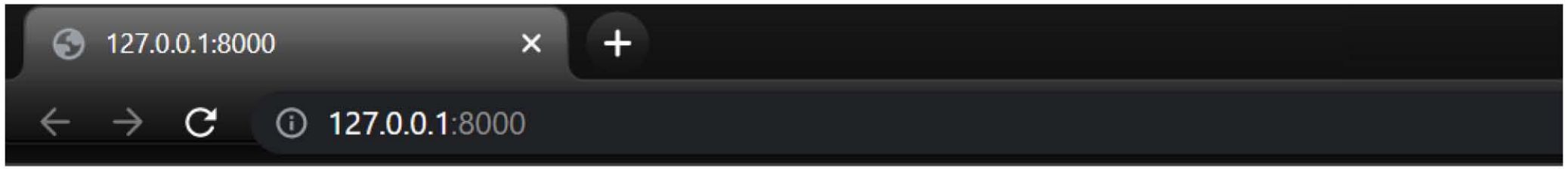
- The "**Test1.html**" will look like this:

```
<html>
    <body>
        <h1>This is an HTML page named Test.html
            <br/>Message1: {{m1}}
            <br/>Message2: {{m2}}
        </h1>
    </body>
</html>
```

- The keys "**m1**" and "**m2**" will be replaced by their respective values when "**Test1.html**" is rendered in the client window.

Django Template Variables:
Passing data from view to a template

- The rendered "**Test1.html**" will look like this:



This is an HTML page named Test.html
Message1: Welcome to Django
Message2: This is a message from views.py

Django Template tag: For Loop (reading list data)

- In "**views.py**" we create a new key value pair in which the value is a list object and we will iterate through the list when rendering it in the html file "**Test1.html**".

```
def f3(request):
    mymessage={
        "m1":"Welcome to Django",
        "m2":"This is a message from views.py",
        "m3":["Python","Java","DJango","C++"]
    }
    return render(request,"Test1.html",mymessage)
```

Django Template tag: For Loop (views.py)

- The "**views.py**" would look like this:

```
1   from django.http import HttpResponse
2   from django.shortcuts import render
3
4   def f1(request):
5       msg="<h1>Welcome to DJango</h1>"
6       return HttpResponse(msg)
7   def f2(request,n):
8       msg="<h1>Value = {} </h1>".format(n)
9       return HttpResponse(msg)
10  def f3(request):
11      mymessage={
12          "m1":"Welcome to Django",
13          "m2":"This is a message from views.py",
14          "m3":["Python","Java","DJango","C++"]
15      }
16      return render(request,"Test1.html",mymessage)
```

Django Template tag: For Loop (reading list data)

- In "**Test1.html**" we write the following code to run a for loop:

```
<ul>
  {% for n in m3%}
    <li> {{n}}
  {% endfor %}
</ul>
```

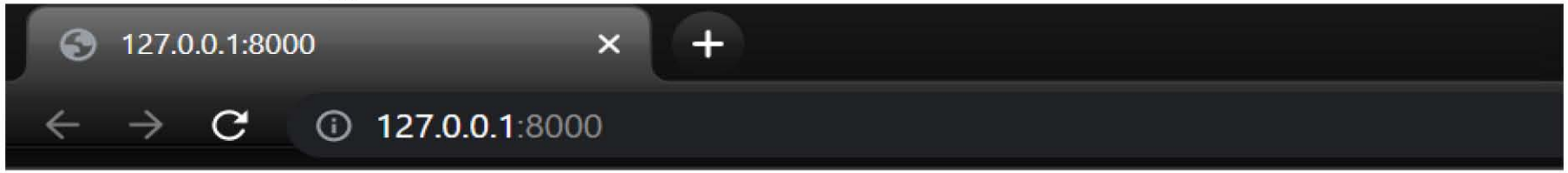
Django Template tag: For Loop (reading list data)

➢The "**Test1.html**" will look like this:

```
<html>
    <body>
        <h1>This is an HTML page named Test.html
            <br/>Message1: {{m1}}
            <br/>Message2: {{m2}}
            <br/>
            <ul>
            {% for n in m3%}
                <li> {{n}}
            {% endfor %}
            </ul>
        </h1>
    </body>
</html>
```

Django Template tag: For Loop (reading list data)

- After rendering "**Test1.html**" will look like this:



This is an HTML page named Test.html
Message1: Welcome to Django
Message2: This is a message from views.py

- **Python**
- **Java**
- **DJango**
- **C++**

Django Template tag:
For Loop (reading dictionary data)

- In "**views.py**" we create a new key value pair in which the value is a list object, inside which is a dictionary object and we will iterate through the keys when rendering it in the html file "**Test1.html**".

```
def f3(request):
    mymessage={
        "m1":"Welcome to Django",
        "m2":"This is a message from views.py",
        "m3":["Python","Java","DJango","C++"],
        "m4":[
            {"Name":"Amit","RollNo":111},
            {"Name":"Sumit","RollNo":222},
            {"Name":"Suresh","RollNo":333}
        ]
    }   return render(request,"Test1.html",mymessage)
```

Django Template tag: For Loop (views.py)

- The "**views.py**" would look like this:

```
def f3(request):
    mymessage={
        "m1":"Welcome to Django",
        "m2":"This is a message from views.py",
        "m3":["Python","Java","DJango","C++"],
        "m4":[
            {"Name":"Amit","RollNo":111},
            {"Name":"Sumit","RollNo":222},
            {"Name":"Suresh","RollNo":333}
        ]
    }
    return render(request,"Test1.html",mymessage)
```

Django Template tag:
For Loop (reading dictionary data)

- In "**Test1.html**" we write the following code to run a for loop:

```
<table border=2 cellpadding=10>
    <th>Name</th>
    <th>Roll No.</th>
    {% for n in m4%}
    <tr>
        <td>{{n.Name}}</td>
        <td>{{n.RollNo}}</td>
    </tr>
    {% endfor%}
</table>
```

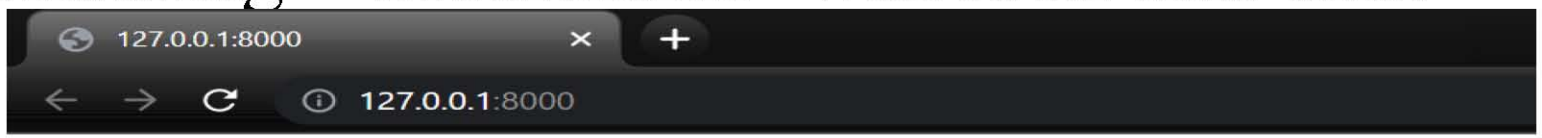
Django Template tag:
For Loop (reading dictionary data)

➢The "**Test1.html**" will look like this:

```
<table border=2 cellpadding=10>
    <th>Name</th>
    <th>Roll No.</th>
    {% for n in m4%}
    <tr>
        <td>{{n.Name}}</td>
        <td>{{n.RollNo}}</td>
    </tr>
    {% endfor%}
</table>
```

Django Template tag:
For Loop (reading dictionary data)

- After rendering "**Test1.html**" will look like this:



This is an HTML page named Test.html
Message1: Welcome to Django
Message2: This is a message from views.py

- **Python**
- **Java**
- **DJango**
- **C++**

Name	Roll No.
Amit	111
Sumit	222
Suresh	333

Django Template tag: If-elif-else

- The code in "**views.py**" will look like this:

```
def f3(request):
    mymessage={
        "m1":"Welcome to Django",
        "m2":"This is a message from views.py",
        "m3":["Python","Java","DJango","C++"],
        "m4":[
            {"Name":"Amit","RollNo":111},
            {"Name":"Sumit","RollNo":222},
            {"Name":"Suresh","RollNo":333}
        ],
        "m5":[10,20,30,40,50,60,70]
    }
    return render(request,"Test1.html",mymessage)
```

Django Template tag: If-elif-else

➢The "**views.py**" will look like this:

```
10    def f3(request):
11        mymessage={
12            "m1":"Welcome to Django",
13            "m2":"This is a message from views.py",
14            "m3":["Python","Java","DJango","C++"],
15            "m4":[
16                {"Name":"Amit","RollNo":111},
17                {"Name":"Sumit","RollNo":222},
18                {"Name":"Suresh","RollNo":333}
19            ],
20            "m5":[10,20,30,40,50,60,70]
21        }
22        return render(request,"Test1.html",mymessage)
```

Django Template tag: If-elif-else

- The code in "**Test1.html**" will look like this:

```
<br/> Print "Welcome" when values <=30,
       Print "To" when value=40 and
       Print "Python" when values>40
       {% for n in m5 %}
       <br/>
          {% if n <= 30 %}
             Welcome
          {% elif n == 40 %}
             To
          {% else %}
             Python
          {% endif %}
       {% endfor %}
```

Django Template tag: If-elif-else

➢The "**Test1.html**" will look like this:

```
22          <br/> Print "Welcome" when values <=30,
23          Print "To" when value=40 and
24          Print "Python" when values>40
25          {% for n in m5 %}
26          <br/>
27              {% if n <= 30 %}
28                  Welcome
29              {% elif n == 40 %}
30                  To
31              {% else %}
32                  Python
33              {% endif %}
34          {% endfor %}
```

Django Template tag: If-elif-else

- The "**Test1.html**" after rendering will look like this:

Print "Welcome" when values <=30, Print "To" when value=40 and Print "Python" when values>40
Welcome
Welcome
Welcome
To
Python
Python
Python

Managing static files

- Static files are present in "**static**" folder.
- The files types that should be kept in this folder are: images, java script, CSS, etc.
- Steps to be followed:
  - **Step-1**: create an HTML theme template (it usually contains an index.html and other html pages along with directories for css, javascript etc.)
  - **Step-2**: copy only the HTML pages in the **templates** folder
  - **Step-3**: copy all the folders (css, fonts, icon, images and js) of the HTML theme template into the **static** folder
  - **Step-4**: add the following line of code in **settings.py** file:
    - **STATICFILES_DIRS=[BASE_DIR,"static"]**
  - **Step-5**: append **"/static/"** in all the places where there are references to images or other files in the HTML pages
    - Example: If there is a hyperlink like About then replace it with About

Django Template tag: Include

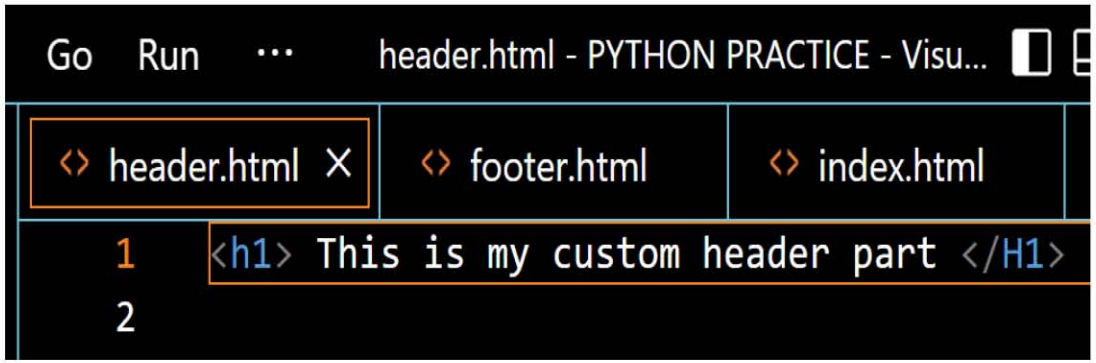
- The **include** tag allows us to include a template (html) inside the current template (html).
- This is useful when we have a block of content that are the same for many pages.
- We use the following code in those places where we want the repeated contents to appear:

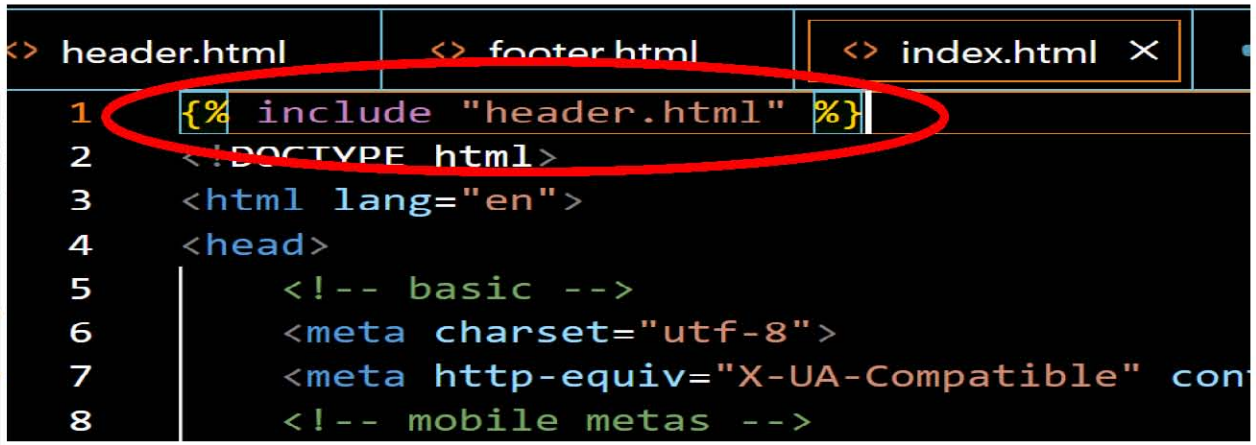
  {% include "Repeated contents.html" %}

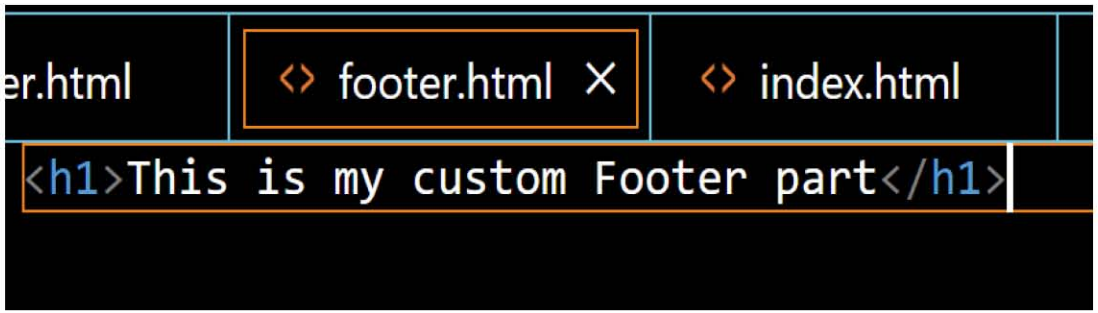
- We can use the **include** tag any number of times.

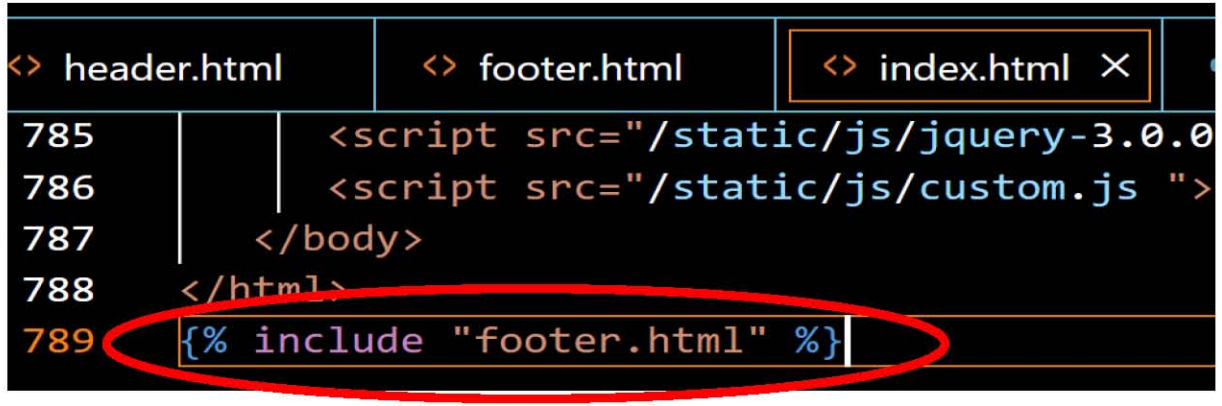
Django Template tag: Include example

- We create two common content html files: **header.html** and **footer.html**
- Our intention is that the contents of **header.html** should be visible at the top of **index.html** and the contents of **footer.html** should be visible at the bottom of **index.html**



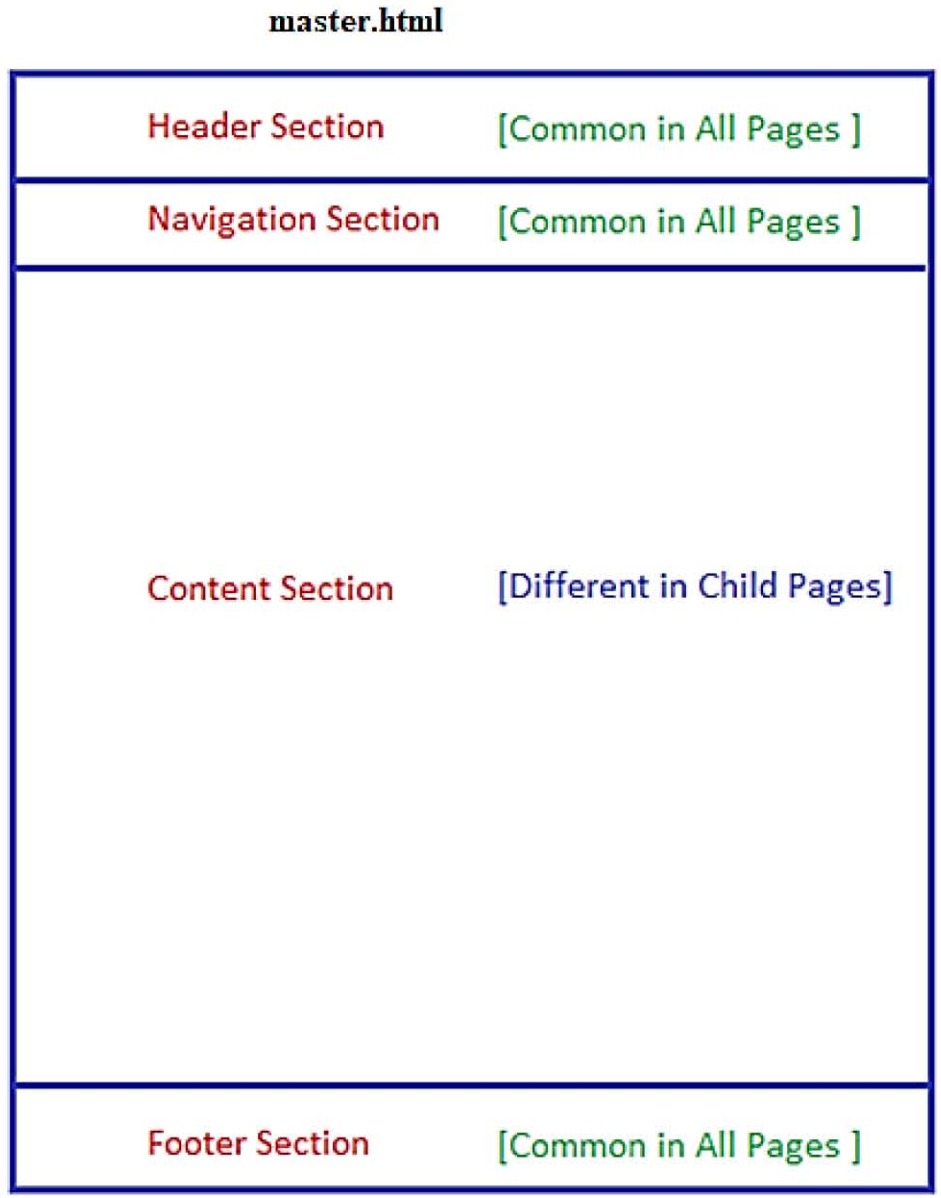






Django Template tag: Extends

- Creating a master design to be implemented in all the child templates in the project.
- Following depicts a sample master template with name **master.html**
- The proposed design will be applicable to all the child templates in the project.



Django Template tag: Extends

- The **extends** tag allows us to create a master template (html) and this master template is inherited by other templates (html) in the project.
- The design that we specify in master template is followed by all the other templates who inherit the master template.
- In master template we specify the placeholders where the child contents will be visible.
- The design in master template is created using the following tags:

```
{% block  block_name %}
{% endblock  block_name %}
```

Django Template tag: Extends

- In the master template we specify the place where the child template contents will be situated using the **{%block%}** tag, so it is like a design guidelines that we provide that will be followed by the child templates.
- When a child template extends a master template then its content will appear exactly in place where the **{%block%}** is specified in the master template.
- The child template should include the following code in them: **{% extends 'master.html' %}** and also the **{%block%}** tags to specify its contents.

Django Template tag: Extends

- Let us see the following example:
- Master template (**master.html**):

master.html | index2.html | views.py

```
1  <h1> Master template custom header </h1>
2  {% block childcontentarea %}
3
4  {% endblock childcontentarea %}
5  <h1> Master template custom footer </h1>
```

Django Template tag: Extends

➢Child template (**index2.html**):

master.html | <> index2.html × | view

```
1   {% extends 'master.html' %}
2   {% block childcontentarea %}
3   <h1> Child page content </h1>
4   {% endblock childcontentarea %}
```

Django Template tag: Extends

- When we run the project and view **index2.html** we see the following output:

Master template custom header

Child page content

Master template custom footer

Form handling (HTML form)

- **Step-1:** We create an HTML page **"form1.html"** in **templates** folder.
- **Step-2:** We write the HTML form code:

```
<form>
   <br/>Enter your Name:<input type="text" name="t1" />
   <br/><input type="submit"/>
   {{message}}
</form>
```

- The variable **{{message}}** will be used to display a welcome message along with name entered in textbox **"t1"**

Form handling (HTML form)

- **Step-3:** In **urls.py** we perform the mapping as follows:

```
urlpatterns = [
        path('form1/', views.form1),
]
```

- **Step-4:** In **views.py** we create the function **form1()** as follows:

```
def form1(request):
    msg=""
    try:
        nm=request.GET["t1"]
        msg="WELCOME:{}".format(nm)
    except:
        pass
    return render(request,"form1.html",{"message":msg})
```

End of UNIT-IV & UNIT-V