Django is a High level Python web-framework developed for rapid development of web-applications.This article is an introduction to Django.

Setup*:-

Virtual Environment:-

Since web-frameworks get updated rapidly and they are mostly not backward-compatible , we would wish to have a separate environment for each project which can override system functionality and not affect other environments and system.This is possible with virtual-environments. Below we describe how to setup a virtual environment (for MS-windows).

First we would install a package that would allow us to create Virtual Environment.For doing that open your CMD and write the following command

```
 ...\>pip install virtualenvwrapper-win
```

After installation of the package now we are ready to create our virtual-environment.To create a virtual environment named X type the following command

```
...\>mkvirtualenv X
```

With this we have created a virtual environment named X , our command prompt will be looking as shown below

```
(X) ...\>
```

This means that we are currently working in Virtual-Environment X.

To deactivate (i.e. return back to system config.) Type the following command

```
(X) ...\>deactivate
```

To continue working on environment X , Type the following command

```
...\>workon X
```

Throughout this entire article we will be working on environment X.

Django:-

To install Django type the following command

* - Reader is assumed to have setup python and pip

```
(X) ...\>pip install Django
```

This would install the latest version of Django for us.

PostgreSQL:-

We will need a database to work on. for this article we would use PostgreSQL.
To install it go to https://www.postgresql.org/ and download the appropriate file.follow the instructions and note the details during setup.

Also install pgAdmin from https://www.pgadmin.org/ which is an administration and development platform for PostgreSQL

Also we need a python database adapter for PostgreSQL.for this we install Psycopg2 by running the following command

```
(X) ...\>pip install psycopg2
```

Understanding Django:-

Django follows Model Template View (MTV) Architectural Pattern . We will understand each of these below

Views:-

Views are python functions that take a HttpRequest object , process it and return a HttpResponse object.

Templates:-

Templates are special Html files which can be configured with dynamic variables and dynamic processes during run-time

Models:-

Models are python classes that provide an interface for tables in the database. Each member of the class is a column in the database.

Whenever a url is sent to the project , it gets mapped to a view. The view may read or write to the database , may process a template and return the requested page.

Initial Steps:-

So, first we will create a project with name projectX by entering the following command in CMD (in the directory we wish to create)

```
(X) ...\>django-admin startproject projectX
```

So Now in our directory we will have

```
projectX/
    manage.py
    projectX/
        __init__.py
        settings.py
        urls.py
        asgi.py
        wsgi.py
```

manage.py - it is a command line utility which lets us interact with our project in various ways.
projectX/settings.py - this module is used to configure settings of our project.
projectX/urls.py - this module is used to map urls to their appropriate view

From now on we will be inside the upper projectX directory , i.e. where our project lives

Lets try and launch our project.Write the following command in cmd

```
(X) ...\projectX>py manage.py runserver
```

Now go to your localhost at port 8000 (http://127.0.0.1:8000/ by default) and we can now see a default welcome page and nothing else because we have not made anything yet.
If you want to stop running your project press ctrl+C. To run again, write the same command shown above.

Now we will create an app.
Apps are modular units of a Django project.They are python packages that provide some set of features.Also apps are plug and play i.e. the same app can be used in multiple projects.
To create a app with name appX type the following command

```
(X) ...\projectX>py manage.py startapp appX
```

Now our project directory would be looking like this

```
projectX/
    manage.py
    projectX/
        __init__.py
        settings.py
        urls.py
        asgi.py
        Wsgi.py
    appX/
        __init__.py
        admin.py
        apps.py
        models.py
        tests.py
        views.py
        migrations/
            __init__.py
```

We will learn about each of these in detail later.

Now for our project to know we have a app , go to projectX/settings.py and update the INSTALLED_APPS field with the name of class residing in AppX/apps.py

projectX/settings.py

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'appX.apps.AppxConfig'
]
```

In this project we will be making a Q&A site.

Views:-

Let's write our first view.Go to AppX/views.py and include the following code

AppX/views.py

```
from django.http import HttpResponse

def sample(request):
    return HttpResponse("This is a sample view")
```

Here we have instantiated the HttpResponse object with the string and returned it.

Now we must be mapping some url to this view. To do that we first create a file with the name urls.py in appX.
And then we update both urls.py as follows

projectX/urls.py

```
from django.urls import include

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

appX/urls.py

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

from . import views

urlpatterns = [
    path('sample/',views.sample,name="sample")
]
```

So here we have used two new function , lets see their uses
django.urls.path(route,view,kwargs=None,name=None)

This function returns an object to be included in urlpatterns list so that if the url pattern when compared from start matches,then it will send the request object to view.it has optional arguments for passing key datas to view and to name the path.

django.urls.include(module,namespace=None)

This function can be passed as view argument in django.urls.path . It tells the path that, strip off the route part of url when matched and check for remaining in module.urlpatterns.It can optionally instantiate a namespace for the URL entries being included.

What happens here is as follows, when I enter the address http://127.0.0.1:8000/sample/ in the browser,a Request object is passed to projectX. Django scans through projectX.urls.urlpatterns , the first match is empty string “” , and its view argument is include(appX.urls) ,so it strips “” from “sample/” retaining “sample/” and then looks for a match in appX.urls.urlpatterns.It gets matched here to appX.views.sample view , so now it goes and executes appX.views.sample function with HttpRequest object passed to it.

Check it by going to http://127.0.0.1:8000/sample/ on your browser.

It is infeasible to include all content in a string , so we'll use our templates for this job.

Templates:-

First we will create a templateX folder in our project and then add BASE_DIR/”templateX” in projectX/settings.py ‘s TEMPLATE[0][‘DIRS’]

projectX/settings.py

```
TEMPLATES = [
    {
        'BACKEND':
'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR/"templates"],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [

'django.template.context_processors.debug',

'django.template.context_processors.request',
```

```
'django.contrib.auth.context_processors.auth',

'django.contrib.messages.context_processors.messages'
,
                ],
            },
        },
]
```

This tells django to look for templates in this directory also , by default it searches for templates in templates folder under apps.

Now let's write a template to show the given items in a list

templates/listing.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>listing</title>
</head>
<body>
    <ul>
        {% for item in list%}
            <li>{{item}}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

{{ variableX }} - displays value of variableX
{% logicX %} - provides functionality of logicX

Complete features can be looked in Templates

Now let's write view and urlpattern to view this template

appX/views.py

```
def listing(request):
    listt = ["apple","banana","carrot","dates","elp"]
    context ={"list": listt}
    return render(request,'listing.html',context)
```

Here we have a new function **django.shortcuts.render(request,template_name,context=None,content_type=None,status=None,using=None) -** it renders the template_name using context as values of placeholder in template(a dictionary) and request as HttpRequest object.

appX/urls.py

```
urlpatterns = [
    path('sample/',views.sample,name="sample"),
    path('listing/',views.listing,name="listing")
]
```

After saving this try going to http://127.0.0.1:8000/listing/ and look at our work

Here we have given all values by ourselves , let's create models to get it from database

Models:-

First we will update settings.py to let it know that we are using PostgreSQL database

projectX/settings.py

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'databaseX',
#create database using pgAdmin first
        'USER': 'mydatabaseuser',
#fill it with what was set while configuration
        'PASSWORD': 'mypassword',
```

```
#fill it with what was set while configuration
        'HOST': '127.0.0.1',
        'PORT': '5432',
#fill it with what was set while configuration
    }
}
```

Let's make two class representing two tables, one for question , other for answer
Open appX/models.py and add the following

appX/models.py

```
from django.db import models

# Create your models here.

class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField()

    def __str__(self):
        return self.question_text

class Answer(models.Model):
    question = models.ForeignKey(Question ,
on_delete=models.CASCADE)
    answer_text = models.TextField()
    pub_date = models.DateTimeField()

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

All model classes must inherit from models.Model
We have defined the columns in model classes.
Also a id field is added by default for all models

Refer Model field reference for more details on fields and models

Now to create these tables in database we have to write 2 commands(quit the server by ctrl+C)

```
(X) ...\projectX>py manage.py makemigrations
```

This will convert the model classes changes into a new python file in appX/migrations that will show changes which are going to happen.
Don't delete files under migration folders . it is like a version control system

```
(X) ...\projectX>py manage.py migrate
```

This will convert those migrate files into SQL commands and hence make updates in the database.

You need to do these two steps , every time you change your models.

Now let us create views and template for our Q&A site

templates/home.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>listing</title>
</head>
<body>
    <ul>
        {% for item in list%}
            <li>{{item}}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

templates/question.html

```
<!DOCTYPE html>
```

```
<html>
<head>
    <meta charset="utf-8">
    <title>question-{{question.id}}</title>
</head>
<body>
    <h1>{{question.question_text}}</h1>
    <hr>
    {% if answers %}
        <ul>
            {%for answer in answers%}
                <li> {{answer.answer_text}}</li>
            {% endfor %}
        </ul>
    {% else %}
        <p> no answers yet </p>
    {% endif %}
</body>
</html>
```

appX/views.py

```
def home(request):
    latest_question_list =
models.Question.objects.order_by('-pub_date')[:5]
    context = {"latest_questions" :
latest_question_list}
    return render(request,"home.html",context)

def question(request,question_id):
    question =
models.Question.objects.get(pk=question_id)
    answers = question.answer_set.all()
    context = {"question":question , "answers" :
answers}
    return render(request,"question.html",context)
```

appX/urls.py

```
urlpatterns = [
    path('sample/',views.sample,name="sample"),
    path('listing/',views.listing,name="listing"),
    path("",views.home,name="home"),
path("<int:question_id>",views.question,name="questio
n")
]
```

After saving all these you can go and look at http://127.0.0.1:8000/ to see the new home page.

But how will we create questions and answers? Let's see

Superuser:-

Django on its own provides a admin interface,go and look at http://127.0.0.1:8000/admin/

Now how to create an admin id?
We do this by the following command

```
(X) ...\projectX>py manage.py createsuperuser
```

Enter the details as per your choice and then you can run server and login

Now to allow admin to add question and answer rows in database, we need to register the models by editing appX/models.py

appX/models.py

```
from . import models

admin.site.register(models.Question)
admin.site.register(models.Answer)
```

Now feel free to login and add question and answers and view our website again

Users:-

We will first create a new app with name accounts for account related processes.
Type the following command in CMD

```
(X) ...\projectX>py manage.py startapp accounts
```

And we update the projectX/urls.py as follows

projectX/urls.py

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

Register:-

Let's first see how to register a user.
We already have user table in database which can be accessed through
django.contrib.auth.models.User

So create these three files as shown

templates/register.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Register</title>
</head>
<body>
    <form action="/accounts/register/" method="POST">
        {% csrf_token %}

        <input type="text" name="username"
placeholder="username"><br>
        <input type="email" name="email"
placeholder="email"><br>
        <input type="password" name="password"
placeholder="password"><br>
```

```
        <input type="submit">
    </form>
</body>
</html>
```

accounts/views.py

```
from django.shortcuts import render,redirect
from django.contrib.auth.models import User

# Create your views here.

def register(request):
    if request.method == "POST" :
        username = request.POST['username']
        email = request.POST['email']
        password = request.POST['password']

        user =
User.objects.create_user(username=username,email=emai
l,password=password)
        user.save()

        return redirect("/")

    else:
        return render(request,'register.html')
```

accounts/urls.py

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

from . import views

urlpatterns=[
    path("register/",views.register,name="register")
]
```

Login:-

django.contrib.auth.models.auth gives us functionality to login and logout a user

For login create/update the following files

templates/login.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>login</title>
</head>
<body>
    <form action="/accounts/login/" method="POST">
        {% csrf_token %}
        <input type="text" name="username"
placeholder="username"><br>
        <input type="password" name="password"
placeholder="password"><br>
        <input type="submit" name="login">
    </form>
</body>
</html>
```

accounts/views.py

```
from django.contrib.auth.models import auth

def login(request):
    if request.method == "POST":
        username = request.POST['username']
        password = request.POST['password']
        user =
auth.authenticate(username=username,password=password
)
        if (auth):
            auth.login(request,user)
        return redirect("/")
```

```
    else:
        return render(request,"login.html")
```

accounts/urls.py

```
urlpatterns=[
    path("register/",views.register,name="register"),
    path("login/",views.login,name="login")
]
```

Logout:-

Logout can also be quickly implemented using django.contrib.auth.models.auth
Quickly update the following files

accounts/views.py

```
def logout(request):
    auth.logout(request)
    return redirect("/")
```

accounts/urls.py

```
urlpatterns=[
    path("register/",views.register,name="register"),
    path("login/",views.login,name="login"),
    path("logout/",views.logout,name="logout")
]
```

We have written the processes for all three but how does the user know it is successful , Lets update our home page

templates/home.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
```

```
    <title>Q and A</title>
</head>
<body>
    {% if user.is_authenticated %}
        Hello,{{user.username}}
        <a
href="/accounts/logout"><button>logout</button></a>
    {% else %}
        <a
href="/accounts/register"><button>register</button></
a>
        <a
href="/accounts/login"><button>login</button></a>
    {% endif %}
    {%if latest_questions%}
        <ol>
            {% for question in latest_questions%}
                <li><a href="/{{question.id}}">
{{question.question_text}}</a></li>
            {% endfor %}
        </ol>
    {% else %}
        <p> no Questions are available <p>
    {% endif %}
</body>
</html>
```

Messages:-

While login what if authentication fails?
Quite often we would need to print messages caused due to current request in next request, for this we use django.contrib.messages

Let's update our login to show message if authentication fails

accounts/views.py

```
from django.contrib import messages
```

```
def login(request):
    if request.method == "POST":
        username = request.POST['username']
        password = request.POST['password']
        user =
auth.authenticate(username=username,password=password
)
        if (user):
            auth.login(request,user)
            return redirect("/")
        else:
            messages.error(request,"invalid
credentials")
            return redirect("/accounts/login/")
    else:
        return render(request,"login.html")
```

templates/login.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>login</title>
</head>
<body>
    <form action="/accounts/login/" method="POST">
        {% csrf_token %}
        <input type="text" name="username"
placeholder="username"><br>
        <input type="password" name="password"
placeholder="password"><br>
        <input type="submit" name="login">
    </form>

    {% for message in messages %}
        <b> {{message}} </b>
```

```
    {% endfor %}
</body>
</html>
```

Now try to enter wrong login credentials in the login page.