



Management of Django Web Development in Python

Ashish Chandiramani¹, Pawan Singh²

Department of Computer Science and Engineering, Amity University Uttar Pradesh, Lucknow Campus, India^{1,2}
ashishchandiramani24@gmail.com¹, pawansingh51279@gmail.com²

How to cite this paper: A. Chandiramani and P. Singh (2021) Management of Django Web Development in Python. *Journal of Management and Service Science*, **1(2)**, 5, pp. 1-17.

<https://doi.org/10.54060/JMSS/001.02.005>

Received: 28/02/2021

Accepted: 12/03/2021

Published: 25/07/2021

Copyright © 2021 The Author(s).

This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).

<http://creativecommons.org/licenses/by/4.0/>



Open Access

Abstract

Application Development is not only limited to the business or promotional field, but it also helps to provide the necessary services to the common people across the world by creating an interaction between the companies and the customers through the website. It also creates a platform for the common people who have the ability to attract the audience using their talent. We can take the example of various blog writers, online councilors and fashion influencers who have a large audience because of their quality content and knowledge in their field. We have generally seen how the electronic companies have marked their presence on the online platform via the website which enables the users to contact the company easily. Online retail companies interact with the users via website and provide the orders at the doorsteps of the people. There are various news agencies which had created their website which keeps the users updated with the latest national and international articles. The website development has helped to create a link between the people and the companies, but behind all these facilities there is a powerful language and commands that holds these computational activities and data transfers which helps to create a good interaction between the users and the website owners. Web developers are classified into Front-end and Back-end web developers, Front-end developers help to create an interactive interface that can be easily accessed by the users, whereas Back-end development deals with the main work that is done behind the screens. We can say that it helps to power the website by managing data transfer and other computational activities. In this project, the importance of Django and back-end web development is explained. Using the tools and commands of these languages we can make an interactive and user-friendly interface that can be easily accessed by people. Back-end was earlier known for limited purposes but now when it is explored thoroughly proved to be one of the best languages for website development, gaming, Artificial Intelligence, and highly advanced mobile applications.

Keywords

Web Development, Django, Python, Backend



1. Introduction

The basics of web development using Django to build blog applications that have the (CRUD) Create, Read, Update, Delete functionality. Django is a widely used free, open-source, and high-level web development framework.[5] It provides a lot of features to the developers "out of the box," so development can be rapid. However, websites built from it are secured, scalable, and maintainable at the same time.

2. Objective

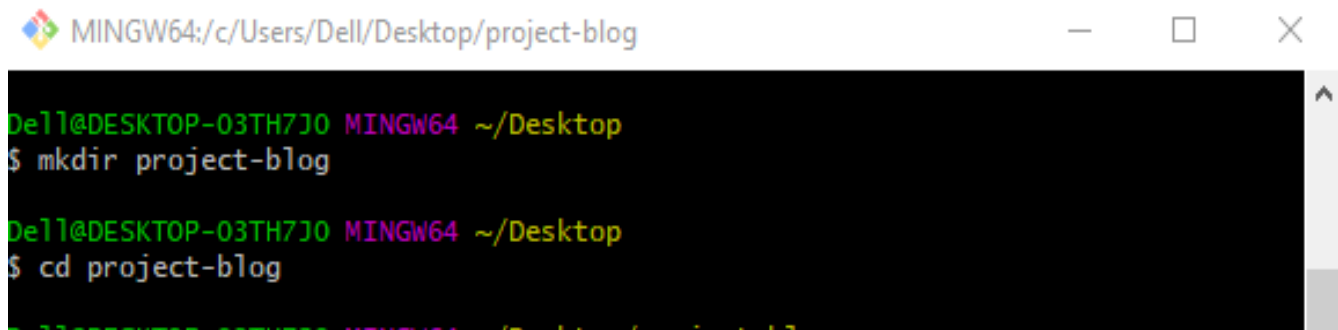
The goal of this project is to build a blog application and a Web App (Weather app) which are updated through an administration panel.

3. Virtual Environment

Virtual Environment act as dependencies to Python-related project. It works as a self-contained container or an isolated environment where all Python-related package [5] and the required versions related to specific project are installed. Since new versions of Python, Django, or packages, etc. will roll out,[4] through the help of a Virtual Environment, you can work with older versions that are specific to any project.

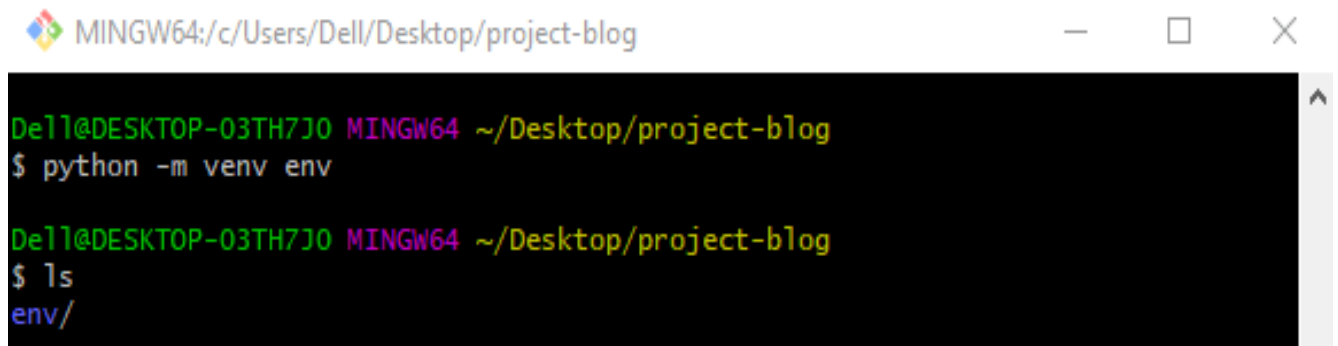
Steps to create a Virtual Environment:

- 1.To create the new directory named 'project-blog' by using 'mkdir' command in your Desktop.
- 2.Change the directory to 'project-blog' by using 'cd' command.



```
MINGW64:/c/Users/Dell/Desktop/project-blog
Dell@DESKTOP-03TH7J0 MINGW64 ~/Desktop
$ mkdir project-blog
Dell@DESKTOP-03TH7J0 MINGW64 ~/Desktop
$ cd project-blog
```

- 3.The virtual environment is created by using 'python -m venv env', where env is our virtual environment shown by 'ls' command.



```
MINGW64:/c/Users/Dell/Desktop/project-blog
Dell@DESKTOP-03TH7J0 MINGW64 ~/Desktop/project-blog
$ python -m venv env
Dell@DESKTOP-03TH7J0 MINGW64 ~/Desktop/project-blog
$ ls
env/
```

- 4.For Activating your Virtual Environment: The Virtual Environment can be activated by using the 'source' command where the 'Scripts' folder needs to be enabled or activated.

```

MINGW64:/c/Users/Dell/Desktop/project-blog
Dell@DESKTOP-03TH7J0 MINGW64 ~/Desktop/project-blog
$ source env/Scripts/activate
(env)
Dell@DESKTOP-03TH7J0 MINGW64 ~/Desktop/project-blog
$

```

The 'env' will be shown in the parenthesis when successfully activated the Virtual Environment.

5. Installing the required package: We can use 'pip install django' to install Django in your specific Virtual Environment.

```

(env)
Dell@DESKTOP-03TH7J0 MINGW64 ~/Desktop/project-blog
$ pip install django
Collecting django
  Using cached https://files.pythonhosted.org/packages/55/d1/8ade70e65fa157e1903fe4078305ca53b6819ab212d9fbb5755afc8ea2e/Django-3.0.2-py3-none-any.whl

```

4. Creating a Django Project

1. The first step is creating project by using the 'django-admin startproject project_name' command, where 'project_name' is 'django_blog' in our case

```

MINGW64:/c/Users/Dell/Desktop/project-blog
(env)
Dell@DESKTOP-03TH7J0 MINGW64 ~/Desktop/project-blog
$ django-admin startproject django_blog
(env)

```

2. Change the directory to the newly created project using 'cd' command and to view the created file using 'ls' command.

```

MINGW64:/c/Users/Dell/Desktop/project-blog/django_blog
(env)
Dell@DESKTOP-03TH7J0 MINGW64 ~/Desktop/project-blog
$ cd django_blog/
(env)
Dell@DESKTOP-03TH7J0 MINGW64 ~/Desktop/project-blog/django_blog
$ ls
django_blog/  manage.py*
(env)
Dell@DESKTOP-03TH7J0 MINGW64 ~/Desktop/project-blog/django_blog
$

```

3. we can run our project by using 'python manage.py runserver'.



```
MINGW64:/c:/Users/Dell/Desktop/project-blog/django_blog
(env)
Dell@DESKTOP-03TH7J0 MINGW64 ~/Desktop/project-blog
$ cd django_blog/
(env)
Dell@DESKTOP-03TH7J0 MINGW64 ~/Desktop/project-blog/django_blog
$ ls
django_blog/  manage.py*
(env)
Dell@DESKTOP-03TH7J0 MINGW64 ~/Desktop/project-blog/django_blog
$ python manage.py runserver
```

4. The project can be viewed in our favorite browser (Google Chrome, Mozilla Firefox, etc.). We can come into our browser and type 'localhost:8000' or '127.0.0.1:8000' in the URL, as shown below in figure 1.

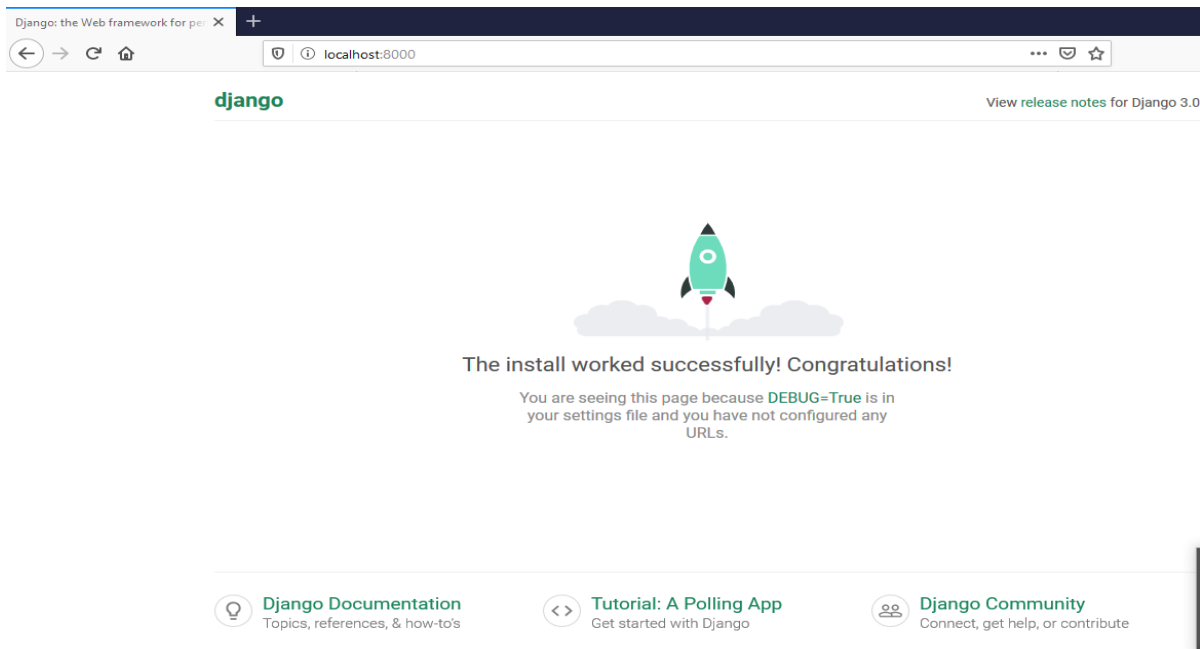


Figure 1. Django local host installation screen

4.1. Starting the new Project

For creating a new Django project, it is a 2-step process, which can be seen below.

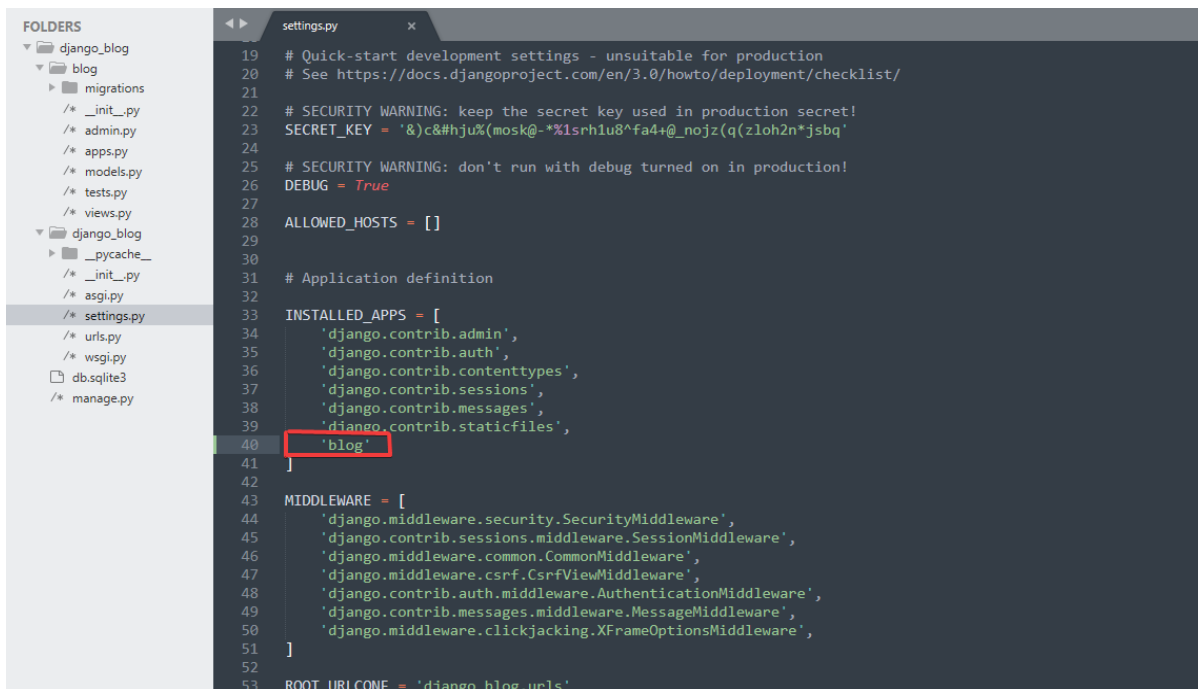
1. To create an app by using 'python manage.py startapp app_name' command, where app_name is 'blog' in our case. In Django, there are many apps to the single project where each app serves as single and specific functionality to the project.

```

MINGW64:/c:/Users/Dell/Desktop/project-blog/django_blog/blog
(env)
Dell@DESKTOP-03TH7J0 MINGW64 ~/Desktop/project-blog/django_blog
$ python manage.py startapp blog
(env)
Dell@DESKTOP-03TH7J0 MINGW64 ~/Desktop/project-blog/django_blog
$ ls
blog/ db.sqlite3 django_blog/ manage.py*

```

2.To make our project let know about our newly created app by making changes to the 'django_blog/settings.py' INSTALLED_APP section.



```

19 # Quick-start development settings - unsuitable for production
20 # See https://docs.djangoproject.com/en/3.0/howto/deployment/checklist/
21
22 # SECURITY WARNING: keep the secret key used in production secret!
23 SECRET_KEY = '&c#hju%(mosk@-%1srh1u8^fa4+@nojz(q(zloh2n*jsbq'
24
25 # SECURITY WARNING: don't run with debug turned on in production!
26 DEBUG = True
27
28 ALLOWED_HOSTS = []
29
30
31 # Application definition
32
33 INSTALLED_APPS = [
34     'django.contrib.admin',
35     'django.contrib.auth',
36     'django.contrib.contenttypes',
37     'django.contrib.sessions',
38     'django.contrib.messages',
39     'django.contrib.staticfiles',
40     'blog'
41 ]
42
43 MIDDLEWARE = [
44     'django.middleware.security.SecurityMiddleware',
45     'django.contrib.sessions.middleware.SessionMiddleware',
46     'django.middleware.common.CommonMiddleware',
47     'django.middleware.csrf.CsrfViewMiddleware',
48     'django.contrib.auth.middleware.AuthenticationMiddleware',
49     'django.contrib.messages.middleware.MessageMiddleware',
50     'django.middleware.clickjacking.XFrameOptionsMiddleware',
51 ]
52
53 ROOT_URLCONF = 'django_blog.urls'

```

4.2. Changing in our Models

Django uses 'SQLite' as a default database, which is light can only be used for small projects, that is fine for this project. 'Object Relational Mapper (ORM)' is used which makes it uncomplicated to work with the database. The true database code is not written, since the database tables are created through the help of 'class' keyword in 'models.py'.

Inside 'blog/models.py', we are creating a new model named 'Post'. This is a class will become a database table after that which currently inherits from models. As in a standard blog, a certain 'Post' contains a title, which will be a field called Char Field. It is a text-based column and accepts mandatory argument as 'max_length', which happens to be 50 in our case. Also, there is another field named 'content', which is the Text Field, which contains the detail text of the 'Post' as in a standard blog. The double underscore('str') method is defined, which overrides the field 'title' and returns the name of actual 'title' instead of some objects.

```

C:\Users\Dell\Desktop\project-blog\django_blog\blog\models.py (django_blog) - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help

FOLDERS
  django_blog
    blog
      __pycache__
      migrations
        __pycache__
        /* 0001_initial.py
        /* __init__.py
        /* __init__.py
        /* admin.py
        /* apps.py
        /* models.py
        /* tests.py
        /* views.py
    django_blog
      __pycache__
      /* __init__.py
      /* asgi.py
      /* settings.py
      /* urls.py
      /* wsgi.py
      db.sqlite3
      /* manage.py

admin.py x models.py x
1 from django.db import models
2
3 # Create your models here.
4 class Post(models.Model):
5     title = models.CharField(max_length = 50)
6     content = models.TextField()
7
8     def __str__(self):
9         return self.title
10

```

4.3. Making Migrations

'python manage.py make migrations' is a first step process which reads the 'models.py' after its creation.

```

MINGW64:/c/Users/Dell/Desktop/project-blog/django_blog
(env)
Dell@DESKTOP-03TH7J0 MINGW64 ~/Desktop/project-blog/django_blog
$ python manage.py makemigrations
Migrations for 'blog':
  blog\migrations\0001_initial.py
  - Create model Post

```

Migrating to the database: This is the 2nd step where 'python manage.py migrate' read the newly created folder 'migrations' thus creating the database.

```

(env)
Dell@DESKTOP-03TH7J0 MINGW64 ~/Desktop/project-blog/django_blog
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, blog, contenttypes, sessions
Running migrations:
  No migrations to apply.

```

Registering to the admin: Let's go to 'blog/admin.py' and do a import of a models called 'Post' by make use of 'from models import Post'. For registering models to the admin, command as follows 'admin.site.register(Post)'.

```

File Edit Selection Find View Goto Tools Project Preferences Help
FOLDERS
  django_blog
    blog
      _pycache_
      migrations
        _pycache_
        /* 0001_initial.py
        /* __init__.py
        /* __init__.py
        /* admin.py
        /* apps.py
        /* models.py
        /* tests.py
admin.py
1 from django.contrib import admin
2 from .models import Post
3 # Register your models here.
4
5 admin.site.register(Post)

```

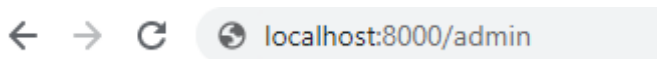
Creating SuperUser and Viewing in the Administration panel: we need to create a SuperUser before accessing the 'admin' panel. To do so, use 'winpty python manage.py createsuperuser'.

```

MINGW64:/c:/Users/Dell/Desktop/project-blog/django_blog
(env)
Dell@DESKTOP-03TH7J0 MINGW64 ~/Desktop/project-blog/django_blog
$ winpty python manage.py createsuperuser
Username (leave blank to use 'dell'): admin
Email address:
Password:
Password (again):
Superuser created successfully.

```

Run server in the background in bash by command `python manage.py runserver`.



Fill out details afterward, i.e., the username and password that you've created earlier as shown in figure 2:

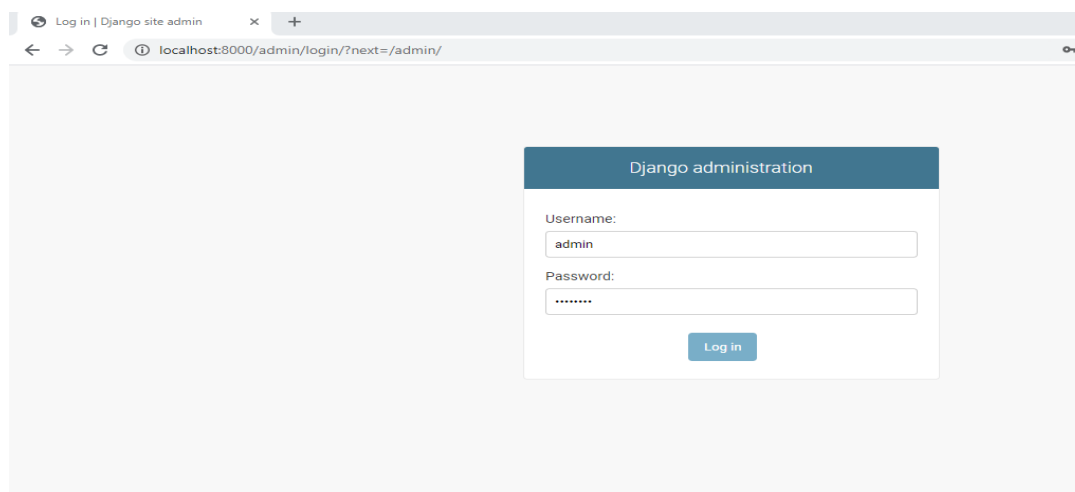


Figure 2(a). Django Administration window

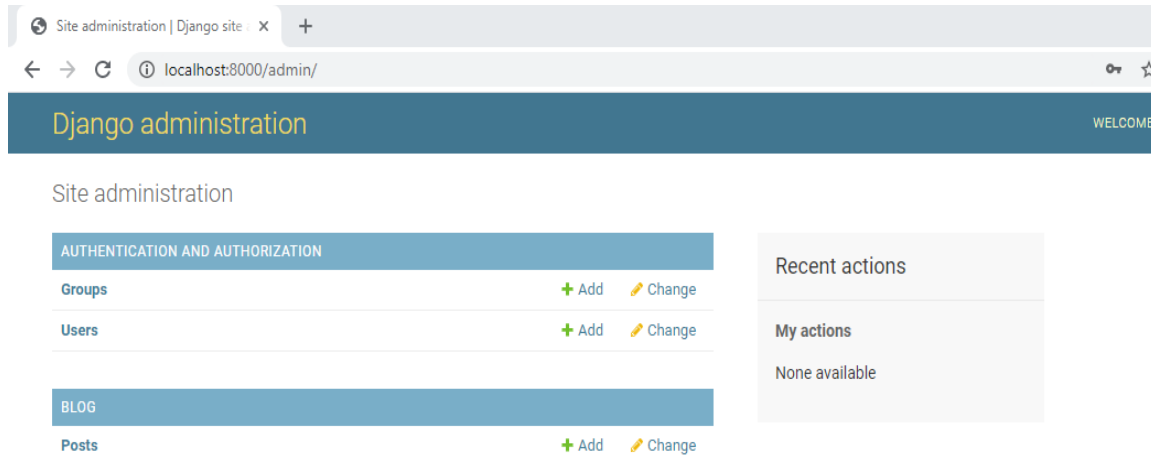


Figure 2(b). Django Administration window

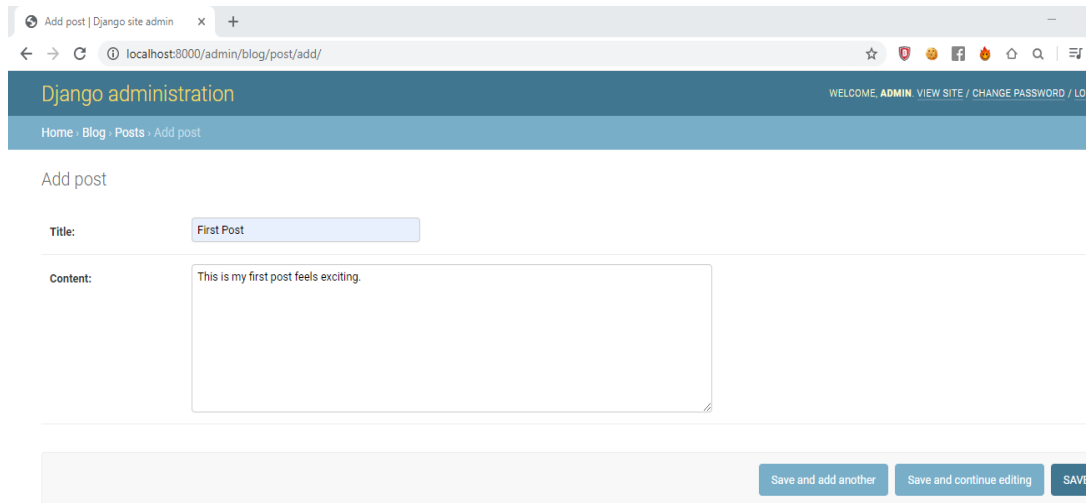


Figure 2(c). Django Administration window

5. Create a Web App (Weather App)

Let us build a simple Django application that displays the weather of various cities. To get current weather data, let us use the Open Weather Map API. We will work with the database and create a form, therefore the concepts used here can be applicable for more complicated projects.

The Admin Dashboard: Next we will look at admin dashboard given by Django. To do that, 1st we have to migrate the database, which means Django will generate the pre-defined tables that is needed for the default apps. To do this, run the migrate command.[8]

- python manage.py migrate

By running this command, Django has generated an SQLite database for us, default database in our settings, and it has added several tables to our database. We will know if our database was created if we see a new `db.sqlite3` file in our project directory. One of the tables Django gives you is a user table, which can be used to store users in our app. The app we are building does not need to have any users having the admin user will allow you to access the admin dashboard. To create the admin user, we will run the create superuser command.

- `python manage.py createsuperuser`

Give a username, email address, and a password for your admin user. Once you have done it, you will need to start your server again and navigate to admin dashboard.

- `python manage.py runserver`

Then go to `server/admin`. We can go to that page because admin is set up in your `urls.py`. If we log in with the username, password we just created, we should see the Django Admin Dashboard. Groups & users represent two models Django give us access to. Models are just code representations of tables in the database. Even though Django created additional tables, there is no need to access the rest of them directly, therefore no models were created. Now leave the admin dashboard for now & go to the code. We are required to create an app into our project for the weather app.

5.1. Creating the App

We are required to create a new application to handle everything regarding the weather. To create that app:

- `python manage.py startapp weather`

By `startapp`, Django has added a new directory & more files to our project. The newly files generated, let's create another file called `urls.py` in the app directory.

```
urls.py
from django.urls import path

urlpatterns = [
]
```

This file is same as to the `urls.py` in our weather directory. The distinction is that this `urls.py` file contains all the URL that are relevant to this app itself. We are not specifying a URL yet, rather we can set up our project to recognize the app and route any URL specific to the app `urls.py` file. First, check the `INSTALLED_APPS` list and then add this app to the list.

```
the_weather/the_weather/settings.py
...

INSTALLED_APPS = [
    'django.contrib.admin',
```



```

'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
'weather',
]
...

```

This lets Django know that we want to use our weather app in the project. By doing that, Django will know where the migrations and the URL are. Afterward, we need to modify the main `urls.py` to point to our weather app `urls.py` file. To do that, we add on a line under the existing path for admin dashboard. We are also required to import `include` so we can point to our app `urls.py` file.

```

the_weather/the_weather/urls.py
from django.contrib import admin
from django.urls import path, include

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

```

The null string means that we won't utilize an endpoint for the entry point to our app. Instead we'll let our app handle any specific endpoint. We could have put up something like `path('weather/', ...)`, which would have meant we would have to type `server/weather/` to get anything with our weather app.

5.2. Adding the Template and View

Next, we need to add template to our project. A template in Django is an HTML file that allow for extra syntax that make the template dynamic. We will be able to do things like add variables, if statement, and loops, among all other things.[9] We have an HTML files, but this will be enough for you to start. We are going to create a template directory to put that file in.

- `cd weather`
- `mkdir templates` && `cd templates`
- `mkdir weather`

We created another directory with the similar name as our application. We did this because Django combine all the template directories by the various apps we have. To prevent filename being duplicated, we can take the name of our app to prevent



creating the duplicates. Inside of our weather directory, creating a new file called index.html. This will be the main template. We have our template created, let us create a view and URL combined so we can actually have this in our app. Views in Django are either functions or classes. since we are creating a simple view, we will create a function. Add this function to your views.py:

```
the_weather/weather/views.py
from django.shortcuts import render

def index(request):
    return render(request, 'weather/index.html') #returns the index.html template
```

We are naming our view index because this will be at index of the app, which is the root URL. having the template render, we would return request, which is must for the render function, & the name of our template file we want to render, like in this case weather/index.html. Let's add this URL that will send this request to this view. The urls.py for this app, update the urlpatterns list.

```
the_weather/weather/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path("", views.index), #the path for our index view
]
```

This allows us to refer the view we created. Django goes to match any URL without the endpoint and route it to view function created by us. Go back to your project root, start the server again.

- `python manage.py runserver`

What we can see now is just the result of HTML you have in index.html file. You will see an input to add cities and the weather for Lucknow. However, the form does not work, and the weather is nothing more than a placeholder, Now we'll be creating those for this app.

5.3. Using the Weather API

What we need to do now is have to sign up for the "Open Weather Map API". This will allow us get real-time weather for any city that we add in our app [1][2][3]. Now we go to the site, create an account, and then go to the API keys on their dashboard. Enter a name & generate a new API key. This key allows us to use the API for obtaining the weather. The 1 endpoint we will use is shown below, so you can see following data that gets returned from modifying the following URL with our API key and navigating to the URLs in your browser [1][2][3]. It takes a few minutes for our API key to become active http://api.openweathermap.org/data/2.5/weather?q=lucknow&units=imperial&appid=YOUR_APP_KEY



With this, let us add in a request to obtain the data into our app. First, we will need to install request so we can call our API from inside our app.

- pipenv **install** requests

Let us update our index view so to send a request to our URL we have.

```
the_weather/weather/views.py
from django.shortcuts import render
import requests

def index(request):
    url = 'http://api.openweathermap.org/data/2.5/weather?q={ }&units=imperial&appid=YOUR_APP_KEY'
    city = 'Lucknow'
    city_weather = requests.get(url.format(city)).json()

    return render(request, 'weather/index.html') #returns the index.html template
```

With those 3 lines, we are adding the URL that we would send a request to. We will make the part for the city just a placeholder for when we allow user to add their own city. For now, we will set the city to be Lucknow, but later this will be set to the cities from the database. Now run the server again and we can see the details of the city regarding the weather

5.4. Displaying the Data in the Template

Next, we need to pass the data to the template so it can be displayed to the user. Let's create a dictionary to hold all of the data we need. Of the data returned, we need temperature, description, and icon.

```
the_weather/weather/views.py
def index(request):
    ...
    weather = {
        'city': city,
        'temperature': city_weather['main']['temp'],
        'description': city_weather['weather'][0]['description'],
        'icon': city_weather['weather'][0]['icon']
    }

    return render(request, 'weather/index.html') #returns the index.html template
```



Now that we have the information we need, we can pass that to our template. To pass it to our template, we'll create a variable named context. This will be a dictionary which will allow us to use its values inside our template.

```
the_weather/weather/views.py
def index(request):
    ...
    context = {'weather' : weather}
    return render(request, 'weather/index.html', context) #returns the index.html template
```

And then in render, we will add the context as the 3rd argument. With the weather data add inside of context, let us go to the template to add our data. Inside of the template, all we want to do is modify our HTML to use variables rather than values I typed in. Variables will be using {{}} tags [9], and they will refer to anything inside of our context dictionary. Note: Django will give you the city name. We do not use weather['city'] like we would in Python. With all the variables replaced, we should now see the current weather for our city. We will create a table in the database to hold the city that we want to know the weather for. We will create a model for this. Let us go to the models.py file in your weather app, and add the following:

```
the_weather/weather/models.py
from django.db import models

class City(models.Model):
    name = models.CharField(max_length=25)

    def __str__(self):
        return self.name

    class Meta:
        verbose_name_plural = 'cities'
```

This will create a table in the database that will have a column named name, which is the cities name. This city will be a CharField, which is a string. To get the changes in our database, we have should run make migrations to generate the code that updates the database and migrate to apply for those changes. `python manage.py makemigration`. `python manage.py migrate`: We need to make it so we can see this model from our admin dashboard. To achieve that, we need to register this in our `admin.py` file.

```
the_weather/weather/admin.py
from django.contrib import admin from .models import City admin.site.register(City)
```

The city is added as an option on the admin dashboard. We can then add some cities into the admin dashboard. I'll start with three: Ludhiana, Tamil Nadu, and Lucknow[3]. With the entries in the database, we need to query these entries in our view. Start by import the City model and then query that model for all objects.

```

the_weather/weather/views.py
from django.shortcuts import render
import requests
from .models import City
the_weather/weather/views.py
...
def index(request):
    url = 'http://api.openweathermap.org/data/2.5/weather?q={ }&units=imperial&appid=YOUR_APP_KEY'
    cities = City.objects.all()
    ...

```

Since we have the cities, we want to loop it over them & get the weather for each 1 and add it to the list that will eventually be passed to the template. This will just be a variation which is of what we did in the first case. Other difference is we want to looping and appending each dictionary to that list. We'll remove the original city variable in place of a city variable in the loop:

```

the_weather/weather/views.py
def index(request):
    ...
    weather_data = []

    for city in cities:

        city_weather = requests.get(url.format(city)).json()

        weather = {
            'city': city,
            'temperature': city_weather['main']['temp'],
            'description': city_weather['weather'][0]['description'],
            'icon': city_weather['weather'][0]['icon']
        }

        weather_data.append(weather)
    context = {'weather_data': weather_data}
    ...

```

Now let's update the context for to pass this list instead of the single dictionary.

```

the_weather/weather/views.py
...
context = {'weather_data': weather_data}
...

```

Next, inside of the template, we want to loop over this list & generate the HTML for every city in the list. For this, we can put up a for loop around the HTML that will generate a single box for that city. After updating the HTML file[9] we can see the data for all the cities we have in the database.

5.5. Creating the Form

The last thing we need to do is allow our user to add a city straight in the form. For that, we need to generate a form. We could generate the form directly, but since our form will have exact same field as our model, we can use our ModelForm. Create a new file called forms.py.

```

the_weather/weather/forms.py
from django.forms import ModelForm, TextInput
from .models import City

class CityForm(ModelForm):
    class Meta:
        model = City
        fields = ['name']
        widgets = {
            'name': TextInput(attrs={'class': 'input', 'placeholder': 'City Name'}),
        }

```

To view this form, we need to create it in the view and pass it to our template. To do that, let us update the index view that will create the form. We will replace the old city variable the same time because we no longer need it. We can also need to update context so the form gets passed to the template.

```

the_weather/weather/views.py
def index(request):
    ...
    form = CityForm()

    weather_data = []
    ...
    context = {'weather_data': weather_data, 'form': form}

```



Now in the template, let us update the form section so that we can use the form from our view & a `csrf_token`, which is needed for POST requests in Django. With the form in HTML working, we now want to handle the form data as this comes in[9]. For that, we'll write an if block checking for the POST request. We want to add the check for this type of request before we start grabbing our weather data, so we immediately obtain the data for the city we add.

```
the_weather/weather/views.py
def index(request):
    cities = City.objects.all()

    url = 'http://api.openweathermap.org/data/2.5/weather?q={ }&units=imperial&appid=YOUR_APP_KEY'

    if request.method == 'POST':
        form = CityForm(request.POST)
        form.save()

    form = CityForm()
    ...
```

By passing `request.POST`, we will be able to validate our form data. Now you should be able to type in the name of a city, click add, and see it show up. I will add Mumbai as the next city. When we drop out of the if block, the form will be recreated so we can add another city if we choose.

Conclusion

We now have a way to keep track of the weather for multiple cities in our app. We had to work with various parts of Django to get this working: views, models, forms, and templates. We also had to use the Python library `requests` to get the actual weather data[1][2][3].

Acknowledgements

When I look at the precious knowledge, I have gained till date in my B. Tech journey, I realize how much time and effort spent toward the completion of this work; not only by me but also by key individuals whom I feel very indebted to. I gratefully acknowledge the support of my supervisor Prof. Dr. Pawan Singh. I would not imagine completing this work without all his advices. I acknowledge the time he spent in our weekly meetings, even with his busy schedule and other commitments. I am deeply indebted to him for his patience. He assisted in all aspects of this work from discussing new ideas to writing and completing this project. I would like to thank Amity University, Lucknow Campus for providing me a wonderful platform for this work.

References

- [1]. S. Klamt and A. von Kamp, "An application programming interface for CellNetAnalyzer," *Biosystems.*, vol. 105, no. 2, pp. 162–168, 2011.
- [2]. S. P. Ong, S. Cholia, A. Jain, et al, "The Materials Application Programming Interface (API): A simple, flexible and efficient API for materials data based on representational State Transfer (REST) principles," *Comput. Mater. Sci.*, vol. 97, pp. 209–215, Feb 2015.



- [3]. S. E. Peters and M. Mc Clennen, "The Paleobiology Database application programming interface," *Paleobiology*, vol. 42, no. 1, pp. 1–7, 2016.
- [4]. A. Gazis and E. Katsiri, "Web frameworks metrics and benchmarks for data handling and visualization," in *Algorithmic Aspects of Cloud Computing*, Cham: Springer International Publishing, vol 11409, pp. 137–151, 2019.
- [5]. C. Burch, "A web framework using Python: tutorial presentation," *Journal of Computing Sciences in Colleges archive*, vol. 25 no.5 pp. 154-155, 2010.
- [6]. L. Titchkosky, M. Arlitt, and C. Williamson, "A performance comparison of dynamic Web technologies," *Perform. Eval. Rev.*, vol. 31, no. 3, pp. 2–11, 2003.
- [7]. S. Shenker, "Fundamental design issues for the future Internet," *IEEE j. sel. areas commun.*, vol. 13, no. 7, pp. 1176–1188, 1995.
- [8]. M. Jailia, A. Kumar, M. Agarwal, et al, "Behavior of MVC (Model View Controller) based Web Application developed in PHP and .NET framework," in *International Conference on ICT in Business Industry & Government* ,2016.
- [9]. Armin Ronacher, "Jinja is Beautiful", <http://jinja.pocoo.org/> ,March 21st, 2015.

