



**LINCOLN COLLEGE OF SCIENCE MANAGEMENT AND TECHNOLOGY  
FACULTY OF INFORMATION TECHNOLOGY AND ENGINEERING**

**CSE 111  
INTRODUCTION TO PROGRAMMING  
FIRST SEMESTER (2021)**

**GETTING STARTED WITH DJANGO**

**KABIRU ABDULHAMID ISA  
[kabiru@lincoln.edu.ng](mailto:kabiru@lincoln.edu.ng)**

## **What is Django?**

Django is a free and open source web application framework, written in Python. A web framework is a set of components that helps you to develop websites faster and easier.

When you're building a website, you always need a similar set of components: a way to handle user authentication (signing up, signing in, signing out), a management panel for your website, forms, a way to upload files, etc.

Luckily for you, other people long ago noticed that web developers face similar problems when building a new site, so they teamed up and created frameworks (Django being one of them) that give you ready-made components to use.

Frameworks exist to save you from having to reinvent the wheel and to help alleviate some of the overhead when you're building a new site.

## **Why Do You Need A Framework?**

To understand what Django is actually for, we need to take a closer look at the servers. The first thing is that the server needs to know that you want it to serve you a web page.

Imagine a mailbox (port) which is monitored for incoming letters (requests). This is done by a web server. The web server reads the letter and then sends a response with a web page. But when you want to send something, you need to have some content. And Django is something that helps you create the content.

## **What Happens When Someone Requests A Website From Your Server?**

When a request comes to a web server, it's passed to Django which tries to figure out what is actually requested. It takes a web page address first and tries to figure out what to do. This part is done by Django's **urlresolver** (note that a website address is called a URL – Uniform Resource Locator – so the name *urlresolver* makes sense). It is not very smart – it takes a list of patterns and tries to match the URL. Django checks patterns from top to bottom and if something is matched, then Django passes the request to the associated function (which is called *view*).

Imagine a mail carrier with a letter. She is walking down the street and checks each house number against the one on the letter. If it matches, she puts the letter there. This is how the `urlresolver` works!

In the *view* function, all the interesting things are done: we can look at a database to look for some information. Maybe the user asked to change something in the data? Like a letter saying, "Please change the description of my job." The *view* can check if you are allowed to do that, then update the job description for you and send back a message: "Done!" Then the *view* generates a response and Django can send it to the user's web browser.

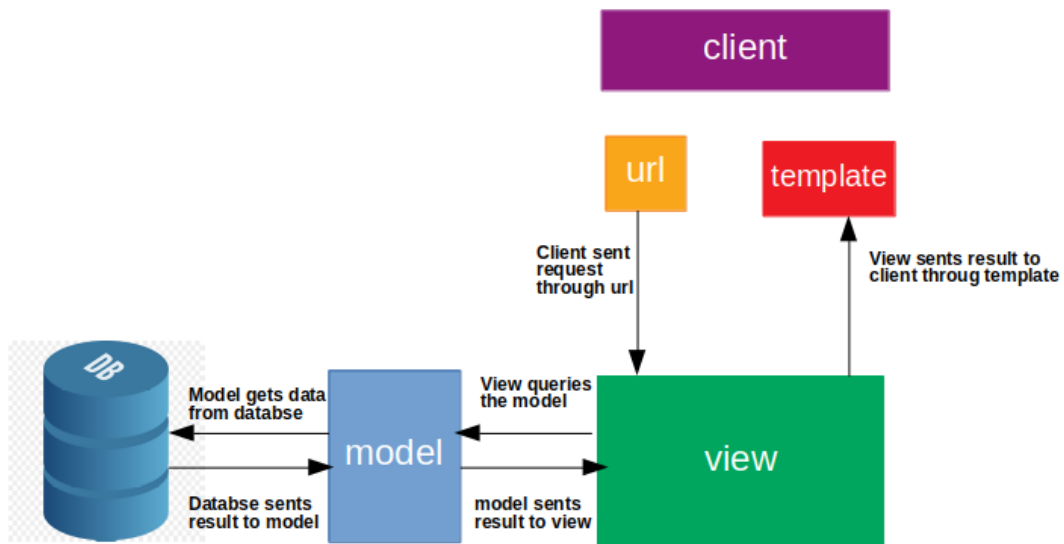
The description above is a little bit simplified, but you don't need to know all the technical things yet. Having a general idea is enough.

So instead of diving too much into details, we will start creating something with Django and we will learn all the important parts along the way!

## **Django Architecture**

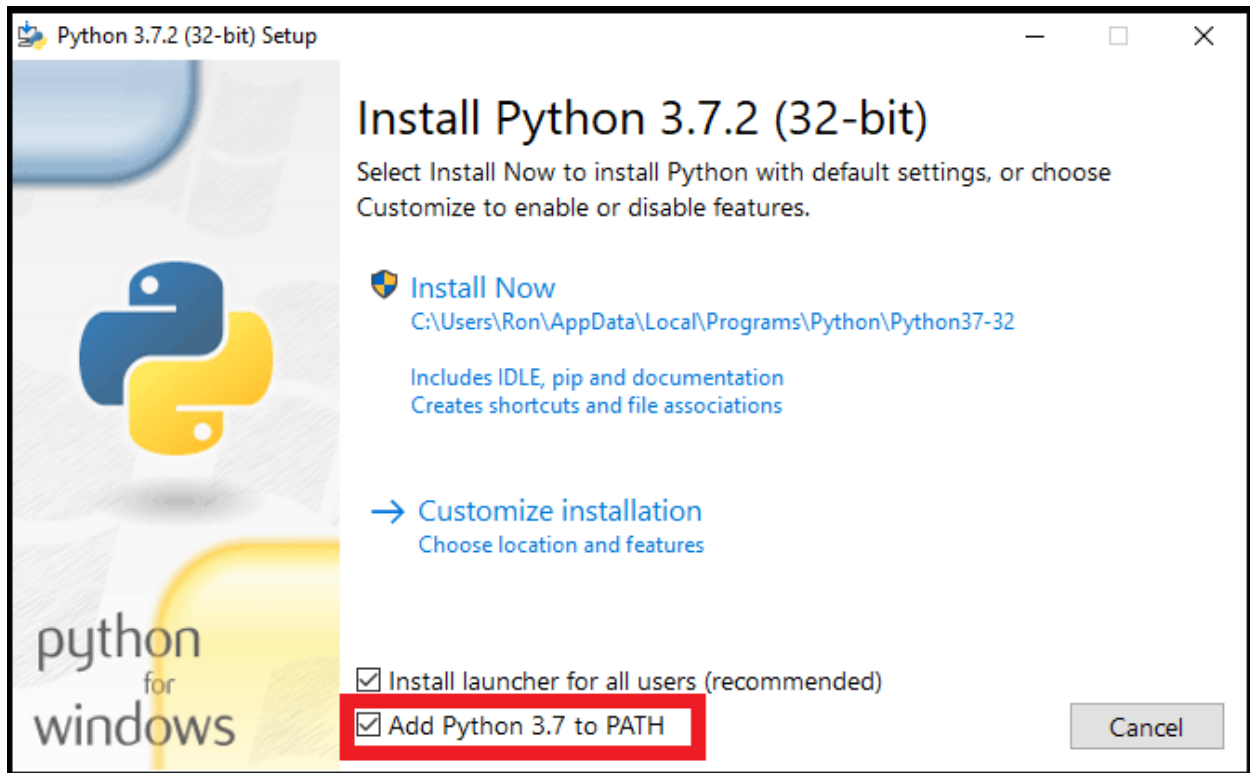
Django follows a Model View Template Architecture (MVT)

- **Model:** The Model is the part of the web-app which acts as a mediator between the website interface and the database. In technical terms, it is the object which implements the logic for the application's data domain.
- **View:** A **view** function, or "**view**" for short, is simply a Python function that takes a web request and returns a web response. The request is sent through url and the view mostly responds through a template(html file)
- **Template:** are the html files that the client interacts with



### **Django installation**

Django depends on python, so you must first install python on your machine. While installing python don't forget to add the python path to the system's environment variable. Do this by checking the option in the installation wizard.



To install Django, open your terminal or command prompt run the following command while connected to an internet

**pip install django**

### Django commands

Django needs some commands to perform operations. These commands are of two categories

1. **django-admin:** these commands are used for generating the skeleton of a django project as well as a django app
  - a. **django-admin startproject *project\_name*:** for creating the skeleton of the project
  - b. **django-admin startapp *app\_name*:** for generating the skeleton of an app
2. **python manage.py:** these commands are used after a **django project** is generated. They are used for interacting with the django project.
  - a. **python manage.py runserver:** used for starting the django local server. The server by default runs on port 8000 of the localhost **127.0.0.1:8000** or

**localhost:8000.** This is the address you should use in your browser to view your django project. You can stop the server using **CTRL + C.** and whenever to stop the server to perform other operations, you need to start it again to continue again

- b. python manage.py makemigrations:** Generates the SQL commands that will be used to create tables and schemas in the database. You should run this whenever you create or modify a model.
- c. python manage.py migrate:** after django generates the SQL commands from the **makemigrations**, this command finally applies the SQL to your database, Thereby creating the database tables. You should always run this command whenever you generate a new django project. Because django used to come with some prebuilt apps(e.g the user app) , so they already come with their own migrations. Therefore you need to run migrate to create the database tables. Similarly whenever you run **makemigrations**, you need to run **migrate** to effect the changes to the database.
- d. python manage.py createsuperuser:** creates a user with all privileges. Whenever you start a new project, you need to run this command inorder to create a new user.

### **Your first Django project!**

We're going to create a small blog!

The first step is to start a new Django project. Basically, this means that we'll run some scripts provided by Django that will create the skeleton of a Django project for us. This is just a bunch of directories and files that we will use later.

The names of some files and directories are very important for Django. You should not rename the files that we are about to create. Moving them to a different place is also not a good idea. Django needs to maintain a certain structure to be able to find important things.

### **django-admin startproject blog project**

This command will generate a new django project containing a folder with **the same name(blog\_project)** and **manage.py** file. The manage.py file is the program responsible for performing all operations with your django project.

```

blog_project
|
|
|_ blog_project
|       |__ asgi.py
|       |__ __init__.py
|       |__ settings.py
|       |__ urls.py
|       |__ wsgi.py
|
|_ manage.py

```

- **asgi.py:** contains implementation to run django in asynchronous mode
- **\_\_init\_\_.py:** use to indicate a python package
- **Settings.py:** contains all the settings for your projects. You need to register all your apps here before django knows their existence.
- **urls.py:** contains all the routes for your project. Each app urls needs to be also included here. It by default comes with django admin app urls. Each django app normally have it separate urls, so you need to register them in project urls
- **wsgi.py:** contains implementation to run django in synchronous mode. It forwards requests to your application from the clients

### Create your database tables

Django needs a database for storing and retrieving data.

run **python manage.py migrate.**

When you run this command for the first time, it does two things

- It creates a database, by default it creates an Sqlite database. You will see it in your project root as db.sqlite3
- Creates tables in the database for the prebuilt apps that django comes with. These apps are Users, Groups, Permissions, contenttypes e.t.c

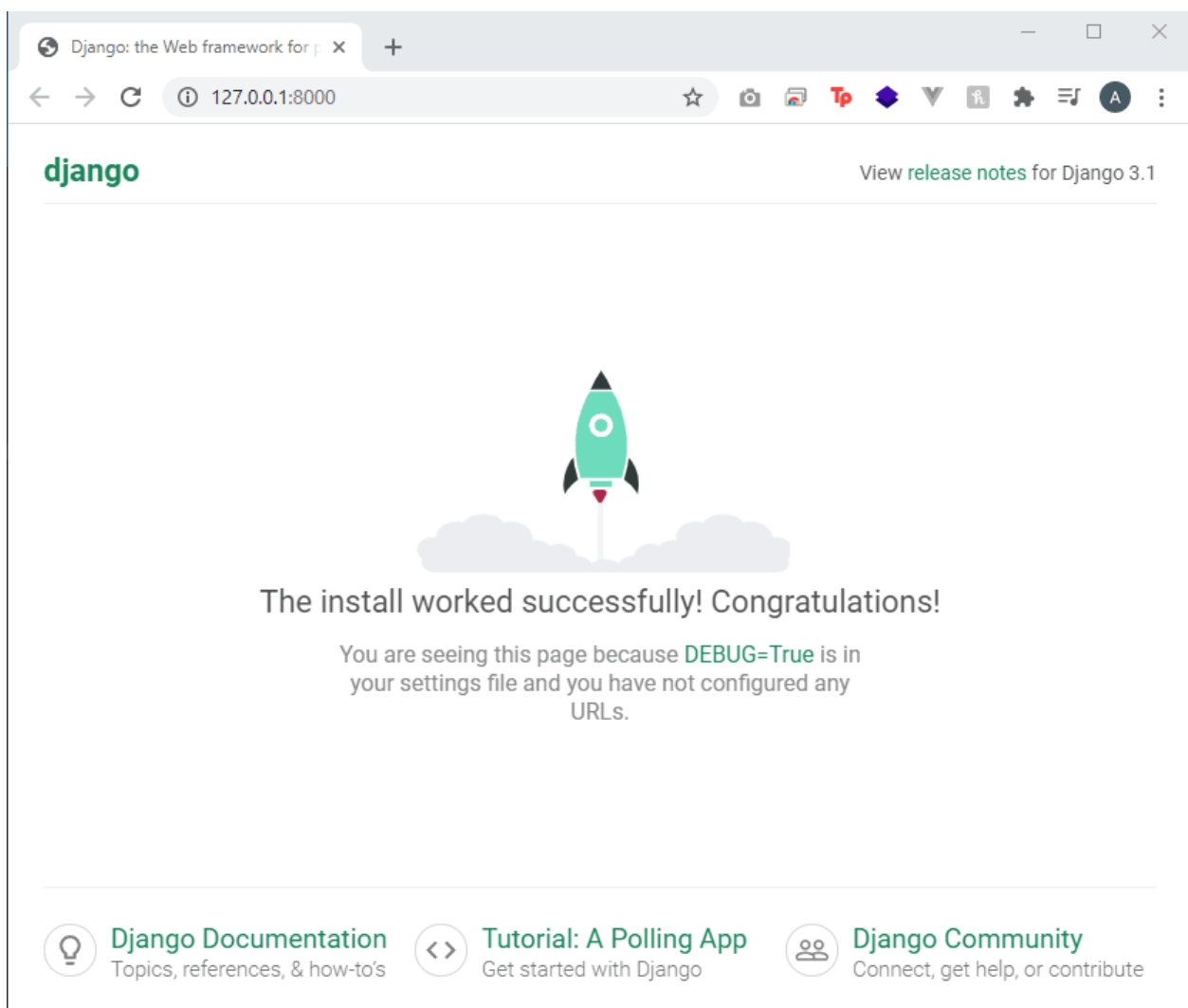
### Create a user to log in to your admin

Run **python manage.py createsuperuser** to create an admin user

### Run server

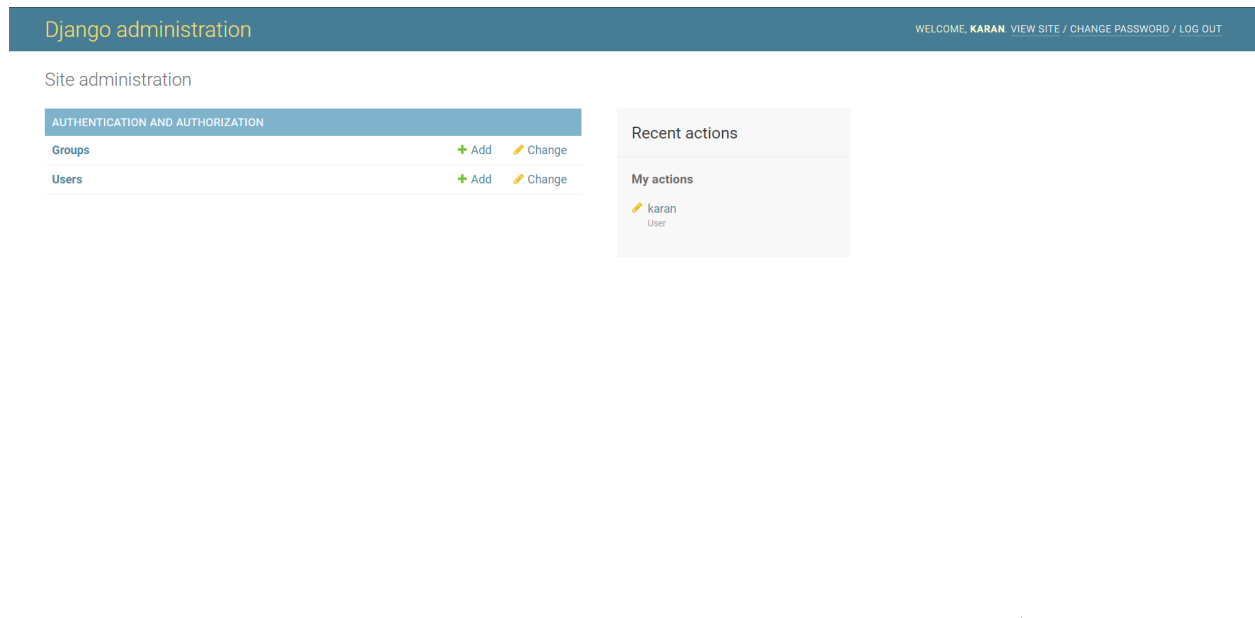
Run **python manage.py runserver**

The server will start at **127.0.0.1:8000**. Open it in your browser, and you will see the django welcome page, congratulating you that your installation is successful



Navigate to **127.0.0.1:8000/admin** and log in to the django admin site.





You will by default see an app named **authentication and permission** with two apps **groups** and **users**. Django saves your time and already did that for you

### Changing The Default **Django Administration** Header

you can change the default header and title by adding the followings lines in the project's `urls.py` file

- `admin.site.site_header = 'My Blog'`
- `admin.site.site_title = 'My Blog'`



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

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

admin.site.site_title = 'My Blog'
admin.site.site_header = 'My Blog'
```

### **Your First Django app**

An app is the unit of modularity of features in a django project. You need to have at least one app in your django project. You can have as many apps as possible when working on a large project. Apps gives you flexibility in breaking your project into manageable pieces. For example if you are asked to develop a School Management System for Lincoln, you may have apps like students, registration, courses, exam, staff, hostel, e.t.c. Each app can be developed individually and registered in the main project.

To start an app for our django project, we run the following command

**django-admin startapp blog**

This will generates an app containing

\_\_\_\_\_admin.py  
\_\_\_\_\_apps.py  
\_\_\_\_\_ \_\_init\_\_.py  
\_\_\_\_\_models.py  
\_\_\_\_\_views.py  
\_\_\_\_\_test.py

*You need to create a **urls.py** in your blog app by yourself. So create it.*

**admin.py:** use for registering your models to the django admin


**apps.py:** apps.py is a configuration file common to all Django apps.

**\_\_init\_\_.py:** use for indicating a python package

**models.py:** Django web applications access and manage data through Python objects referred to as models. Models define the *structure* of stored data, including the field *types* and possibly also their maximum size, default values, selection list options, help text for documentation, label text for forms, etc. The definition of the model is independent of the underlying database — you can choose one of several as part of your project settings. Once you've chosen what database you want to use, you don't need to talk to it directly at all — you just write your model structure and other code, and Django handles all the dirty work of communicating with the database for you.

### **Registering your app to the project**

by default a django project does't know the existence of an app, untill you register it to your main project. Do this by adding it to INSTALLED\_APPS section of the project settings.py file



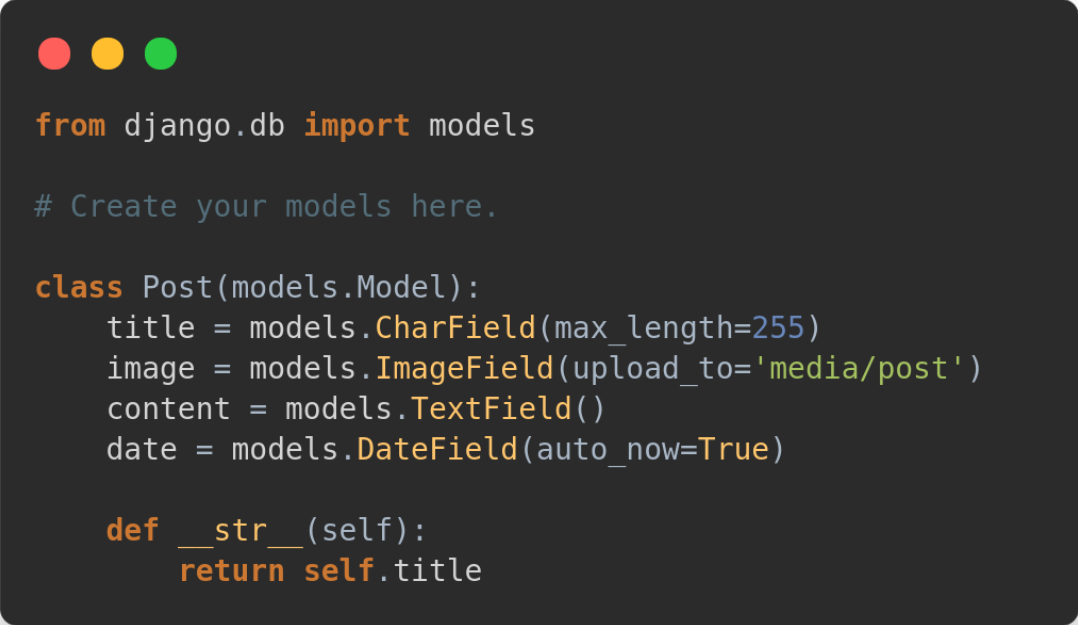
```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'blog'  
]
```

### **Creating the blog post model**

Before you jump in and start coding the models, it's worth taking a few minutes to think about what data we need to store and the relationships between the different objects.

We need to information about a blog post, so we need **title, image, content and date** for each post.

A django model is basically a class, containing member variables or attributes and functions. So for our model post, it will be a class with four attributes, and one function (`__str__`) for defining the string representation of an object.



```
from django.db import models

# Create your models here.

class Post(models.Model):
    title = models.CharField(max_length=255)
    image = models.ImageField(upload_to='media/post')
    content = models.TextField()
    date = models.DateField(auto_now=True)

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

**Title:** since title of our blog will be like a sentence, we use the `models.CharField` and specify the maximum length it could take to be 255

**image:** to support uploading image we need to use the `models.ImageField` and specify the folder to which the images will be saved. For this we specify `media/post` (you don't have to create the folder yourself, Django will create it automatically when you try to upload an image from your app).

To use image in Django, you need to also install another package called `Pillow` using

`pip install pillow`

More configurations need to be made later on serving the images.

**content:** the blog content is basically much, so we use `TextField` as an alternative to `CharField` to support larger text.

**Date:** `models.DateField` is used for date. You can optionally pass an `auto_now=True` argument to automatically use the current date.


**`__str__(self)`:** is a function that returns the string representation of a particular blog post.

**After creating your model you need to run `makemigrations` and `migrate`. You should**

**always do this whenever you create new model or modify an existing model.**

### **Registring model to admin**

Django already comes with admin out of the box. You can create new object of a certain model as well as edit and delete. But before a model will be accessed in the admin you need to register it in admin.py file of your app.



```
from django.contrib import admin
from blog.models import Post

# Register your models here.
admin.site.register(Post)
```

You need to first import the model in the form:

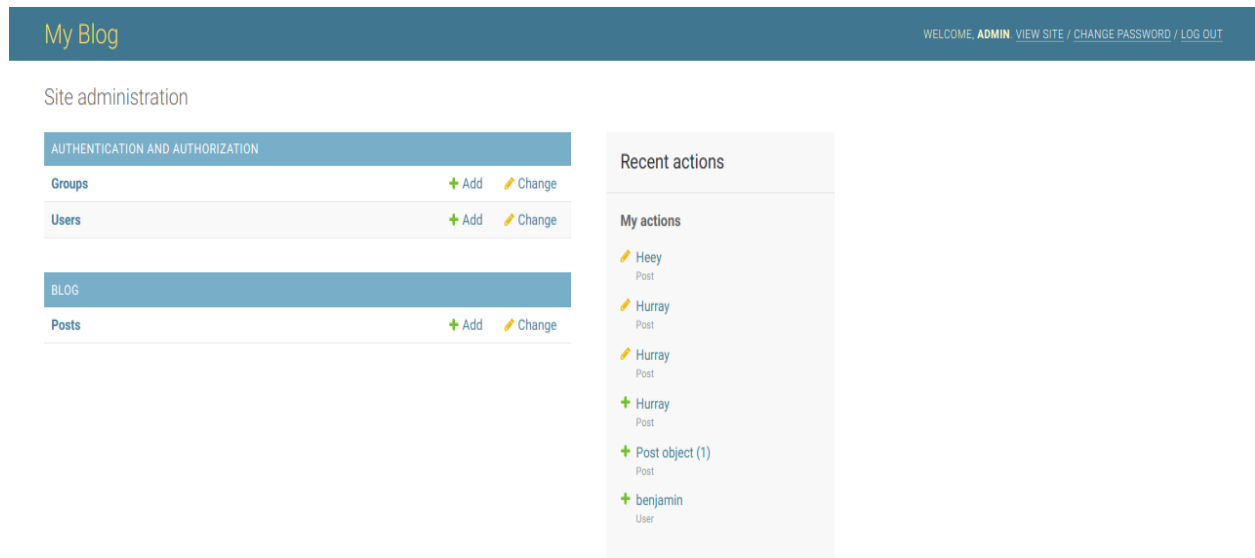
***from appname.models import Model***

in our case the appname is blog and the model is Post, that is why we have

***from blog.models import Post***

after importing, then register the model to the admin with

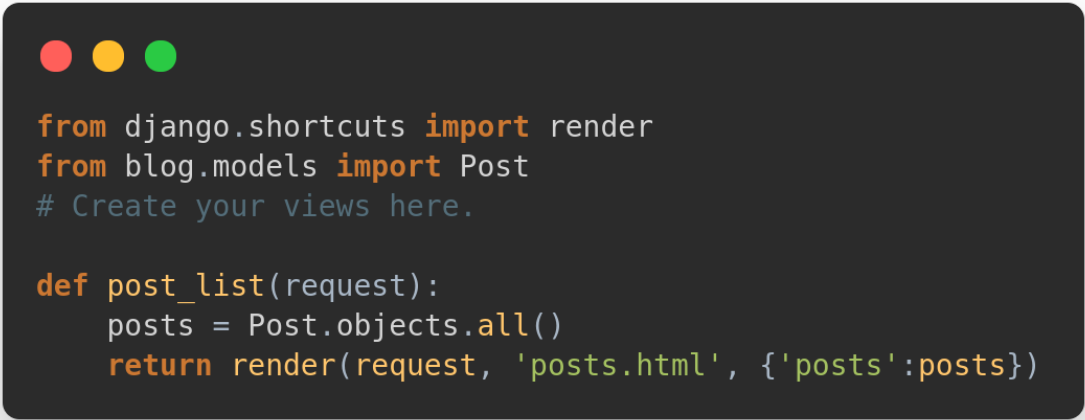
***admin.site.register(Post)***



The admin list each apps with their models. In the image above two apps are listed with their corresponding models.

## Creating views

views are basically python functions or classes that recieves request from client, process that request and return the result to client by rendering it through a template. For this project, clients will visit our app to view a list of blog posts. So we need to create view that process request.



```
from django.shortcuts import render
from blog.models import Post
# Create your views here.

def post_list(request):
    posts = Post.objects.all()
    return render(request, 'posts.html', {'posts':posts})
```

- First we need to import our Post model inorder to return a list of those post
- then create a function that will take request as an argument and return the list of posts and a template to display the posts.

### **Create urls for the app**

URL short for uniform resource locator, are locations of informations in the world wide web. Views in django processes the information needed by the client. But the views on their own cannot perform their operations, they need to be accessed first before they serve their purpose. So Urls are used to invoke the views inorder to carryout operation. Therefore each view in django must have a coresponding url.

Now for a client to view our posts we need to create a url for the view responsible for returning the lists of the posts. Django app doest not come with a urls.py file, you need to create it yourself inside the blog app.





```
from django.urls import path
from blog.views import post_list

urlpatterns = [
    path('', post_list, name='post_list')
]
```

We need to first import that view from our views.py file:

***from blog.views import post\_list***

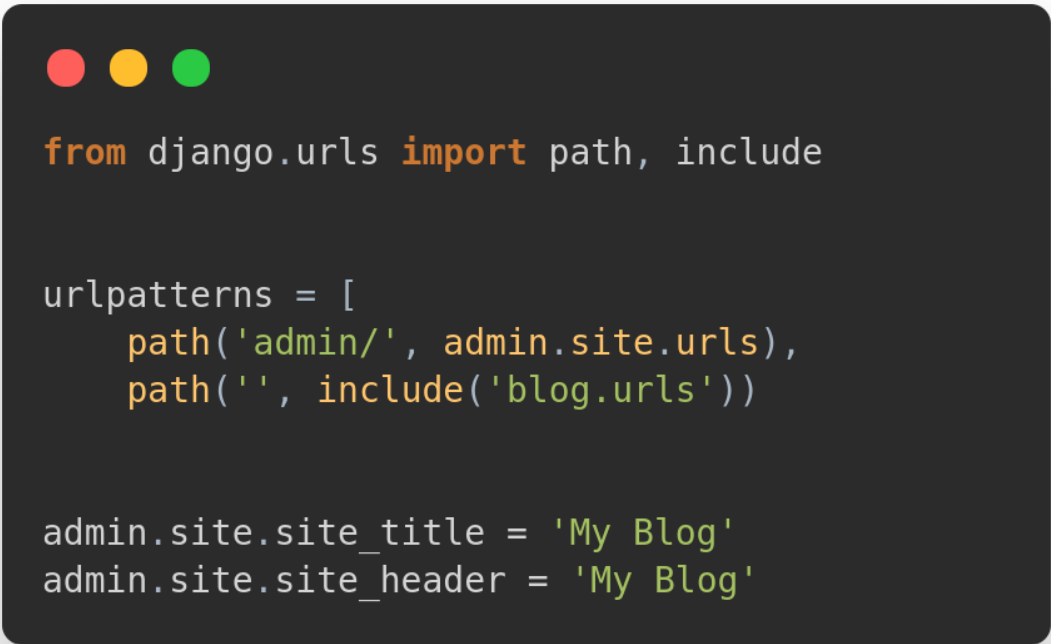
Django uses a list **urlpatterns** to contain all your urls. A particular url is specified by calling a function called path and supplying three arguments:

- The first argument is the path. For example if it is:
  - ‘/about’ the client need to go tot 127.0.0.1:8000/about in the browser
  - ‘/posts’ the client need to go tot 127.0.0.1:8000/posts
  - ‘/company/contact’ the client need to go tot 127.0.0.1:8000/company/contact
  - ‘’ when its empty that means the client need to go the address without appending any to the route 127.0.0.1:8000
- The second argument is the view which the path is url is created for
- The third is a keyword argument of the name you want to give to the url

### **Register App Urls To Project Urls**

Each app in django has its own urls. But you need to register them to the project url so that the

urls will be mapped to the lists of the project urls.



```
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('blog.urls'))

admin.site.site_title = 'My Blog'
admin.site.site_header = 'My Blog'
```

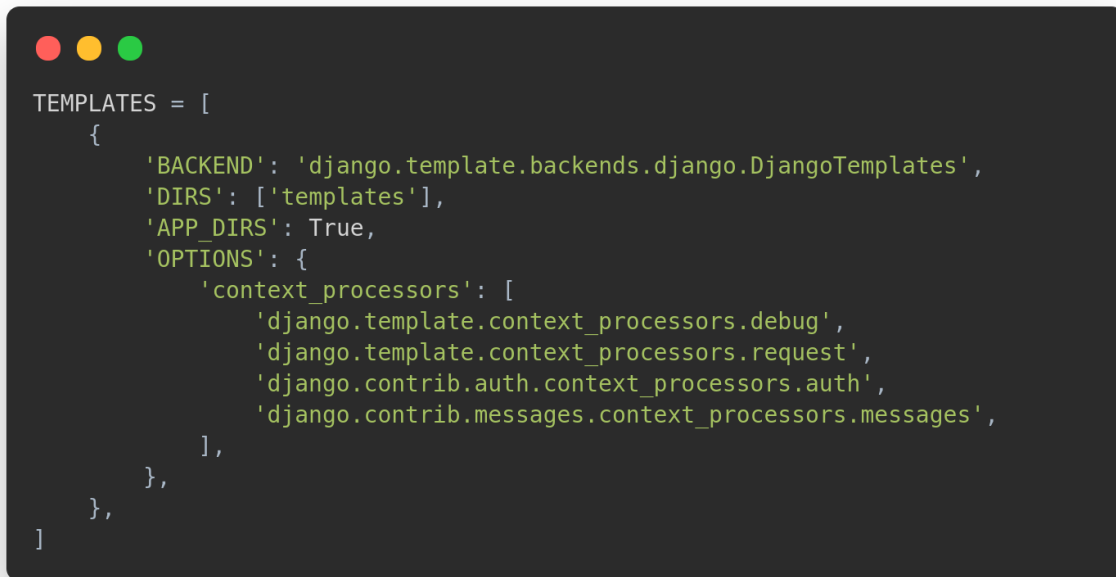
To register the app urls all what you will do is to include them to the project urls. First you need to import include. Include is a function in the same module with path. So just update the path include from

```
from django.urls import path
to
from django.urls import path, include
```

### **Creating The Templates**

create a template folder named 'templates' in the root directory where manage.py file is. Then register it to the project by assigning it as a value of **DIRS** in the **TEMPLATES** section of

settings.py file.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The code is written in a light green monospace font. It defines the TEMPLATES setting as a list of dictionaries. The first dictionary contains the following keys: 'BACKEND' (set to 'django.template.backends.django.DjangoTemplates'), 'DIRS' (set to ['templates']), 'APP\_DIRS' (set to True), and 'OPTIONS' (a dictionary with a 'context\_processors' list). The 'context\_processors' list contains four strings: 'django.template.context\_processors.debug', 'django.template.context\_processors.request', 'django.contrib.auth.context\_processors.auth', and 'django.contrib.messages.context\_processors.messages'. The code is properly indented to show the nested structure.

```
TEMPLATES = [  
    {  
        'BACKEND': 'django.template.backends.django.DjangoTemplates',  
        'DIRS': ['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',  
            ],  
        },  
    ],  
]
```

### **Creating our posts.html file**

Remember in our `post_list()` view we return `posts.html` file as part of the render. So now let's create that file inside the **templates** folder. This is where we will display the lists of posts, that the view function returns.

```
<html>
  <head>
    <title> My Blog</title>
  </head>

  <body>
    <div class="post-list">
      {% for post in posts%}
        <div class="post">
          <h1>{{post.title}}</h1>
          
          <p>{{post.content}}</p>
          <small>{{post.date}}</small>
        </div>
      {% endfor %}
    </div>
  </body>
</html>
```

We create a div for the list of posts. Then inside that div we loop through the list of posts and create a div for each post containing its title, img, content and date. Dont forget to add endfor, you use this templates to indicate the end of a for loop.

### Handling Static and media files

Static files are the CSS, Javascript and Images we use in our application. Django recommends serving them separately and by convention uses a directory called “**static**” to house those files. Create a static folder in the root directory where manage.py file is. Create sub directories for css, javascript, and img depending on the static files you are going to use.


In Django, files which are uploaded by the user are called **Media or Media Files**. Here are some examples:

- A user uploaded image, pdfs, doc files etc while publishing a post.
- Images of products in an e-commerce site.
- User's profile image. etc...

To handle static and media files, two things need to be done: configuring them in settings.py file and registering their urls in urls.py file of the project

### *Configuring static and media files in settings.py*

add the following lines at the end of settings



```
STATIC_URL = '/static/'

STATICFILES_DIRS = [
    os.path.join(BASE_DIR, 'static')
]

MEDIA_URL = 'media/'

MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

STATIC\_URL: base url of static files

STATICFILES\_DIR: the directory of the static files

MEDIA\_URL: base url of media files

MEDIA\_ROOT: the directory of the media files

### *registering static and media urls in project's urls*

```
from django.contrib import admin
from django.urls import path, include
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('blog.urls'))
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)

admin.site.site_title = 'My Blog'
admin.site.site_header = 'My Blog'
```

first import settings from django.conf and static from django.conf.urls.static. After that attach the media url to the project urlpatterns by adding a +

*+ static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)*

### **Including static files in html**

To include a static file like css in django is slightly different from the conventional way. Django has a templating engine, that has its own way of including static files in HTML.

Lets take for example that we create a styles.css file in css directory inside static directory and wants to include our posts.html file.

```

{% load static %}
<html>
  <head>
    <title> My Blog</title>
    <link rel="stylesheet" href="{% static 'css/style.css' %}">
  </head>

  <body>
    {% for post in posts%}
    <h1>{{post.title}}</h1>
    
    <p>{{post.content}}</p>
    <small>{{post.date}}</small>
    {% endfor %}
  </body>
</html>

```

For every file that you want to use static files, you have to first load the static files with the template tag **{% load static %}** at the top of the file.

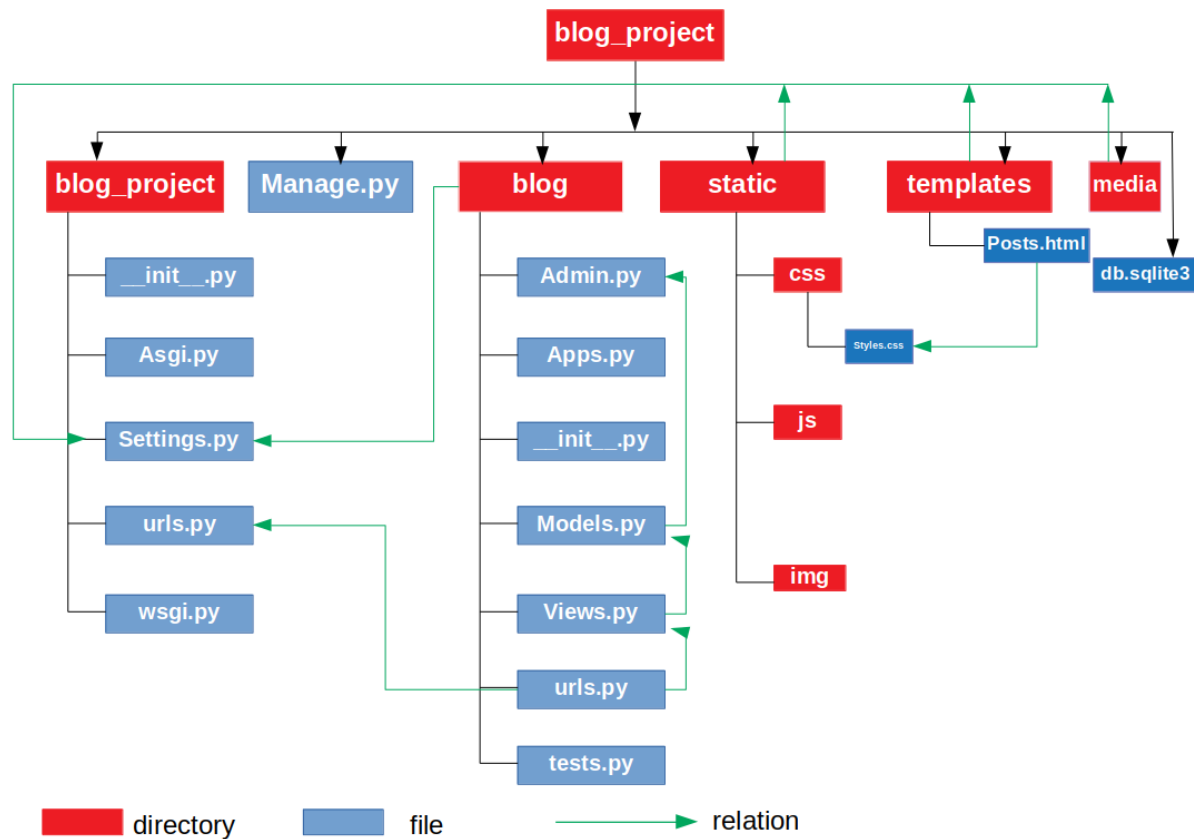
Instead of including css as

```
<link rel="stylesheet" href="static/css/styles.css" >
```

use

```
<link rel="stylesheet" href="{% static 'css/styles.css' %}">
```

## Graphical



## Source

The files for this tutorial can be found here for reference

<https://github.com/iAmKabiru/django-blog-lincoln.git>

## Further Reading

<https://docs.djangoproject.com/en/3.1/>

<https://djangogirls.org/>

<https://books.agiliq.com/projects/django-api-polls-tutorial/en/latest/>

[https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Tutorial local library website](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Tutorial_local_library_website)