Django Tutorial

# Introduction

This is a tutorial that follows Corey Schafer’s Django series videos.

Video series: <https://www.youtube.com/watch?v=UmljXZIypDc&list=PL-osiE80TeTtoQCKZ03TU5fNfx2UY6U4p>

Youtube Channel : <https://www.youtube.com/channel/UCCezIgC97PvUuR4_gbFUs5g>

Code snippets : <https://github.com/CoreyMSchafer/code_snippets/tree/master/Django_Blog/snippets>

# Basics

## Basic Django start commands:

django-admin startproject projectname # creates new project

python manage.py runserver # runs the server

python manage.py migrate # creates a a SQlite3 database for your project.

This will create a new django project.

Making a new app within our django project.

A app basically means a part of our site . For eg. I created a blog app that represent the blog part of our website:

We create a new app with the following way:

We navigate to our project root folder and type in:

**python manage.py startapp blog**



## How do we add paths to our website :

We navigate to our newly created app ‘views.py’ module and add the following:

Think of views as a module that represents what will be shown when you use the functions from it.

from django.shortcuts import render  
from django.http import HttpResponse

def home(request):  
 return HttpResponse('<h1>Blog Home Page</h1>')  
  
def about(request):  
 return HttpResponse('<h1>About Page</

These are functions that will run when we got on home and about page (if we assign those functions to them)

Next step is making a new .py module called ‘urls’ which will contain the path and the functions to the pages.

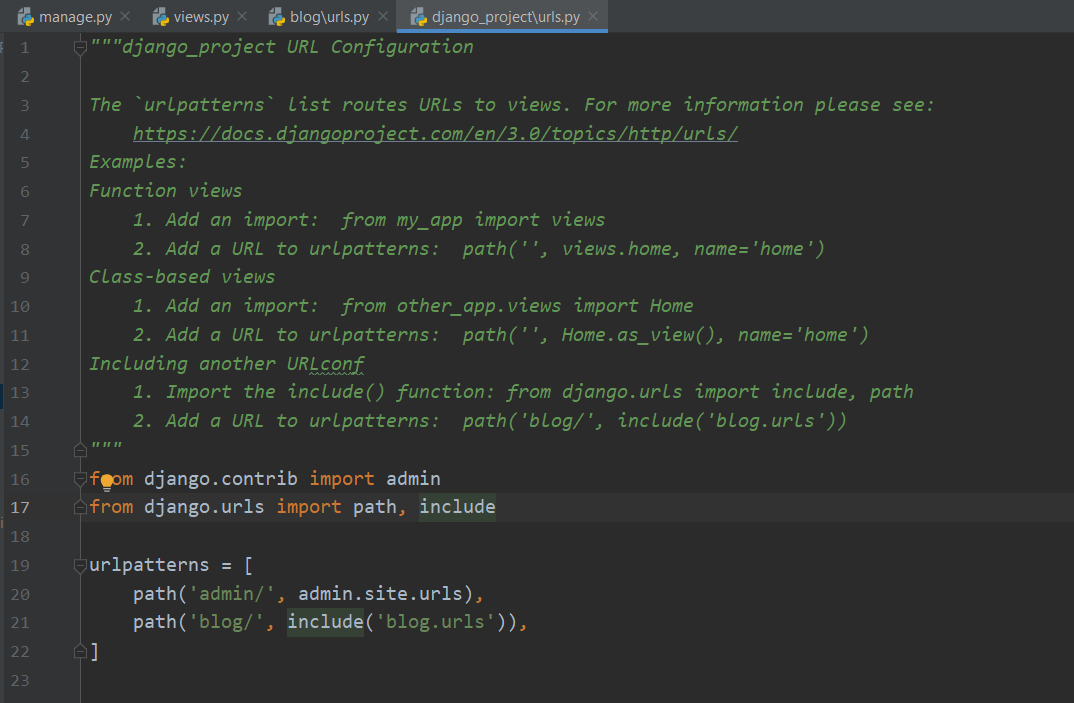
from django.urls import path # path function from django.urls  
from . import views # '.' means from current folder  
  
urlpatterns = [  
 path('', views.home, name='blog-home'),  
 path('about/', views.about, name='home-about')  
]

The upper code basically makes our website have two more ‘pages’:

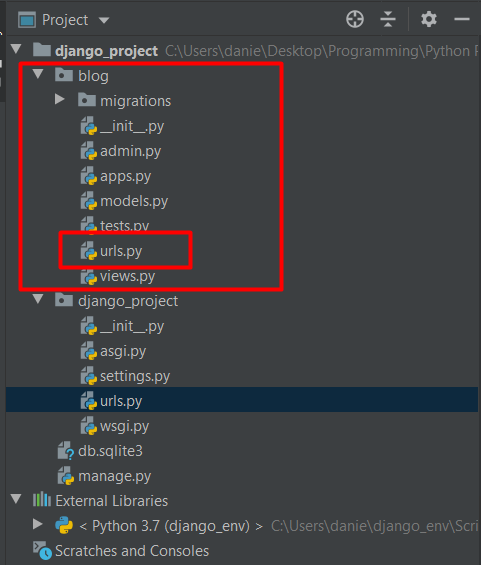
/blog/ # when it’s on blog homepage it will execute views.home function

/blog/about # when it’s on the about page it will execute views.about function

To make it work we have to add this path to the main project URL Module:



from django.contrib import admin  
from django.urls import path, include  
# The include() function basically references a python module in our case called blogs.url or in other words it MAPS it to our blogs.urls  
urlpatterns = [  
 path('admin/', admin.site.urls),  
 path('blog/', include('blog.urls')),  
]



# Templates

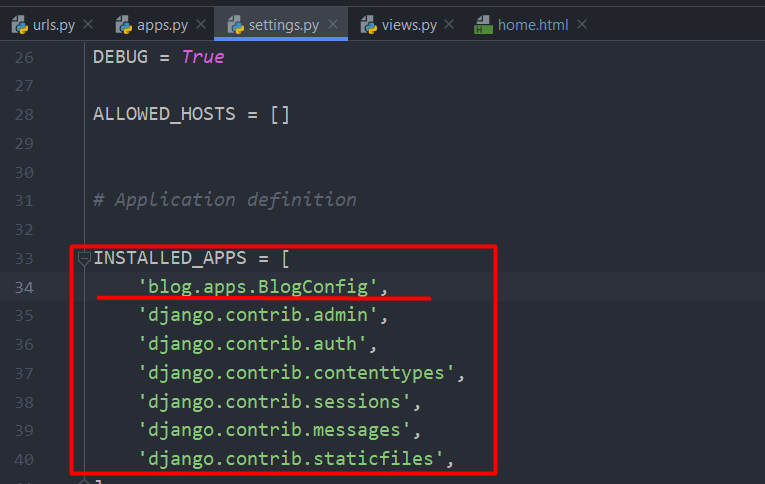
## Adding our app config class to the main projects settings module

First we need to add our apps config class to the main project settings.py module.

myapp/apps.py , down below is our class config file

*class* BlogConfig(AppConfig):  
 name = 'blog'

Now we add this to our main project settings.py module



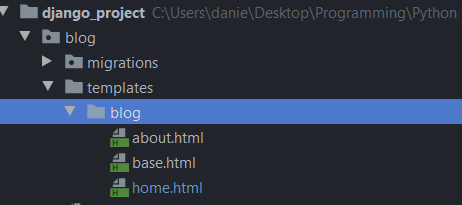
## Making templates

In order to make templates work , we have to add a templates folder inside the app we want to use the templates in our case ‘Blog’ and it should look something like this

eg. desiredapp/templates/desiredapp/

# Note : We have to make another folder in there with our apps name.

Here we are going to add the templates we want to use



Next we go to our views.py folder and make sure that we use render() on these templates

## Django render() function:

*from* django.shortcuts *import* render

*from* django.http *import* HttpResponse

*def* about(request):  
 *return* render(request, 'blog/about.html')

# Render includes HttpResponse in it’s function

# Basically here when we the user navigates to our blog page this function will handle the request and load about.html file

## Adding dummy data and using it in our templates:

Below we add some dummy data and use it in our blog/templates/blog/home.html file

*from* django.shortcuts *import* render  
*from* django.http *import* HttpResponse  
  
posts=[  
 {  
 'author': 'Ghirasim Daniel',  
 'title': 'Blog Post 1',  
 'content': 'First post content',  
 'date\_posted': '31 December, 2019'  
 },  
 {  
 'author': 'Jane Doe',  
 'title': 'Blog Post 2',  
 'content': 'Second post content',  
 'date\_posted': '01 January, 2020'  
 },  
  
]  
  
*def* home(request):  
 context = {  
 'posts': posts  
 }  
 *return* render(request, 'blog/home.html', context)

Breaking it down in steps:

1. We make a list of dictionaries containing some dummy data
2. We go into our home function and make a new dictionary called context where we make a key called ‘posts’ and the value we assign to it is our posts list of dictionaries
3. Then we add the context dict as an argument to our return statement

## Accessing data in our templates .html files:

This is how our home.html file looks:

<!DOCTYPE html>  
<html>  
<head>  
 <title>Title</title>  
</head>  
<body>  
 {% for post in posts%}  
 <h1>{{post.title}}</h1>  
 <p>By {{post.author}} on {{post.date\_posted}}</p>  
 <p>{{post.content}}</p>  
 {% endfor %}  
</body>  
</html>

### Code blocks in Django:

Code blocks in Django are represented like this :

{% %} – This means we are writing a code in the HTML file

### FOR LOOP in Django:

A Django for loop looks like this:

{% for post in posts%}

{% endfor %}

### VARIABLES in Django:

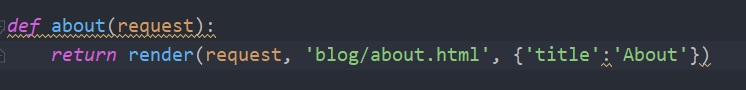
<h1>{{post.title}}</h1>

The double curly braces means that we want to access a variable

### IF ELSE statements in Django:

<head>  
 {% if title %}  
 <title>Django blog - {{ title }}</title>  
 {% else %}  
 <title>Django Blog</title>  
 {% endif %}  
</head>

We define title in our views.py:



With the IF ELSE statement we say:

If we have a title print ‘Django Blog – {title}’ else print ‘Django Blog’

And it will look like this if we have a title:



and if we don’t have one:



## Template Inheritance - Base Template – Making our life easier

Making a base template helps us reduce the amount of code we have to write . We can make a base.html file from which our other .html **will inherit from** so if we modify the base.html file the other ones will automatically get modified.

We add to this base.html file all the things that are going to be constant troughout all of our .html files for eg:

<head>  
 {% if title %}  
 <title>Django blog - {{ title }}</title>  
 {% else %}  
 <title>Django Blog</title>  
 {% endif %}  
</head>

We want this if statement for all of our website.

### How does a base template work

This is how our base.html looks after some editing:

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"  
 "http://www.w3.org/TR/html4/loose.dtd">  
<html>  
<head>  
 {% if title %}  
 <title>Django blog - {{ title }}</title>  
 {% else %}  
 <title>Django Blog</title>  
 {% endif %}  
</head>  
<body>  
 {% block content %}{% endblock %}  
</body>  
</html>

Between the block content code we are going to set each or .html files information.

for eg. home.html will look like this:

{% extends 'blog/base.html' %}   
{% block content %}  
 {% for post in posts%}  
 <h1>{{post.title}}</h1>  
 <p>By {{post.author}} on {{post.date\_posted}}</p>  
 <p>{{post.content}}</p>  
 {% endfor %}  
{% endblock content %}

We deleted everything that is found in base.html and are that we are left with is the actual code we want to run on the page.

{% extends 'blog/base.html' %}

This tells our home.html file that it **extends** base.html and after that we add:

{% block content %}  
 # We add our home.html code here  
{% endblock content

{% block content %} this refers to the CONTENT BLOCK we can have several blocks in a template file.

like {% block posts %} {% block pictures %} (have to verify this).

This is how our about page looks.

{% extends 'blog/base.html' %}  
{% block content %}  
 <h1>About Page!</h1>  
{% endblock content %}

Notice we reduced the code drastically by using template inheritance.

## Bootstrap

Bootstrap is a extremely popular library which makes it easy to add some nice styles to your website.

Basically we drag and drop the Bootstrap starter template into our base.html file . (It does some magic and it works).

#### Starter template:

<!doctype html>

<html lang="en">

<head>

<!-- Required meta tags -->

<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

<!-- Bootstrap CSS -->

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous">

<title>Hello, world!</title>

</head>

<body>

<h1>Hello, world!</h1>

<!-- Optional JavaScript -->

<!-- jQuery first, then Popper.js, then Bootstrap JS -->

<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo" crossorigin="anonymous"></script>

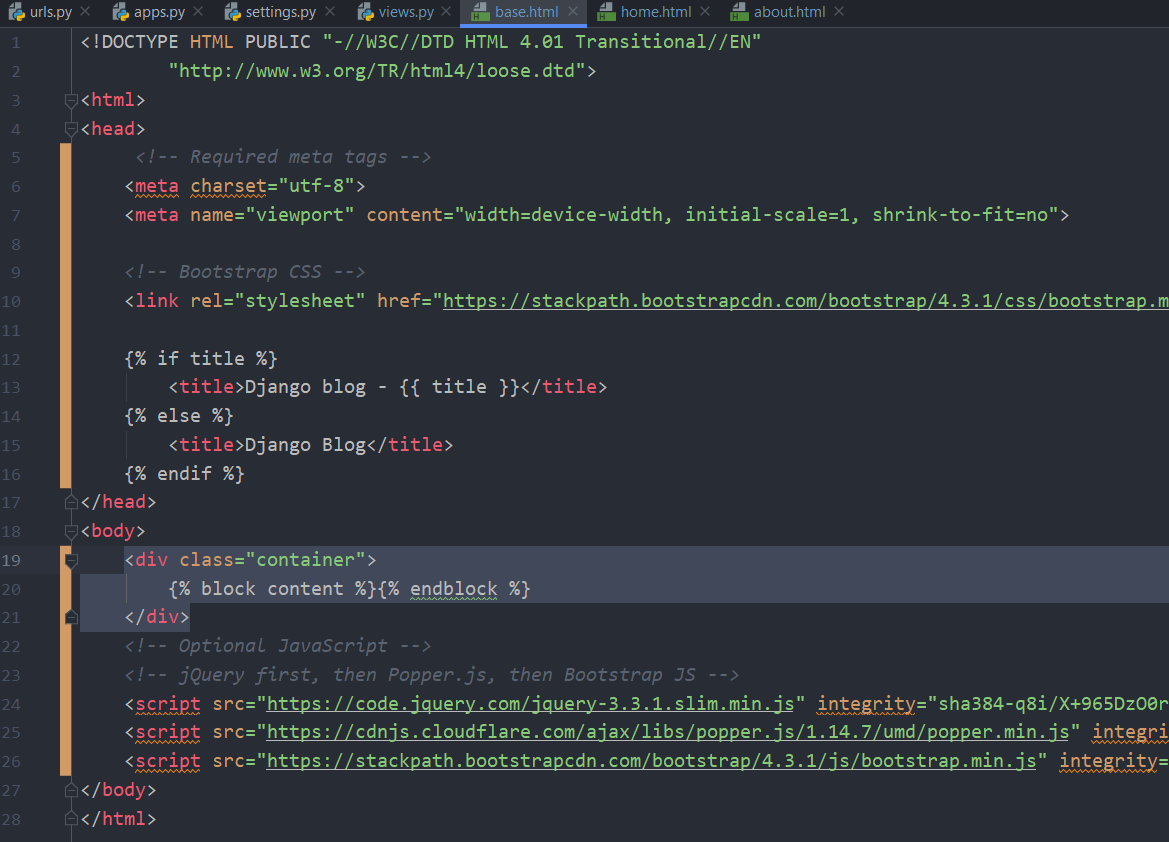
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js" integrity="sha384-UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1" crossorigin="anonymous"></script>

<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js" integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM" crossorigin="anonymous"></script>

</body>

</html>

#### Adding bootstrap to our base.html



Notice we added our block content between <div> tags . I don’t really know the magic behind this but after the first div tag we add a class called ‘class=”container”’ (This gives our content some good padding and spacing)

After adding this I noticed the text moves slightly to the right.

## Adding a navigation bar

This is the template for a navigation bar:Top of Form

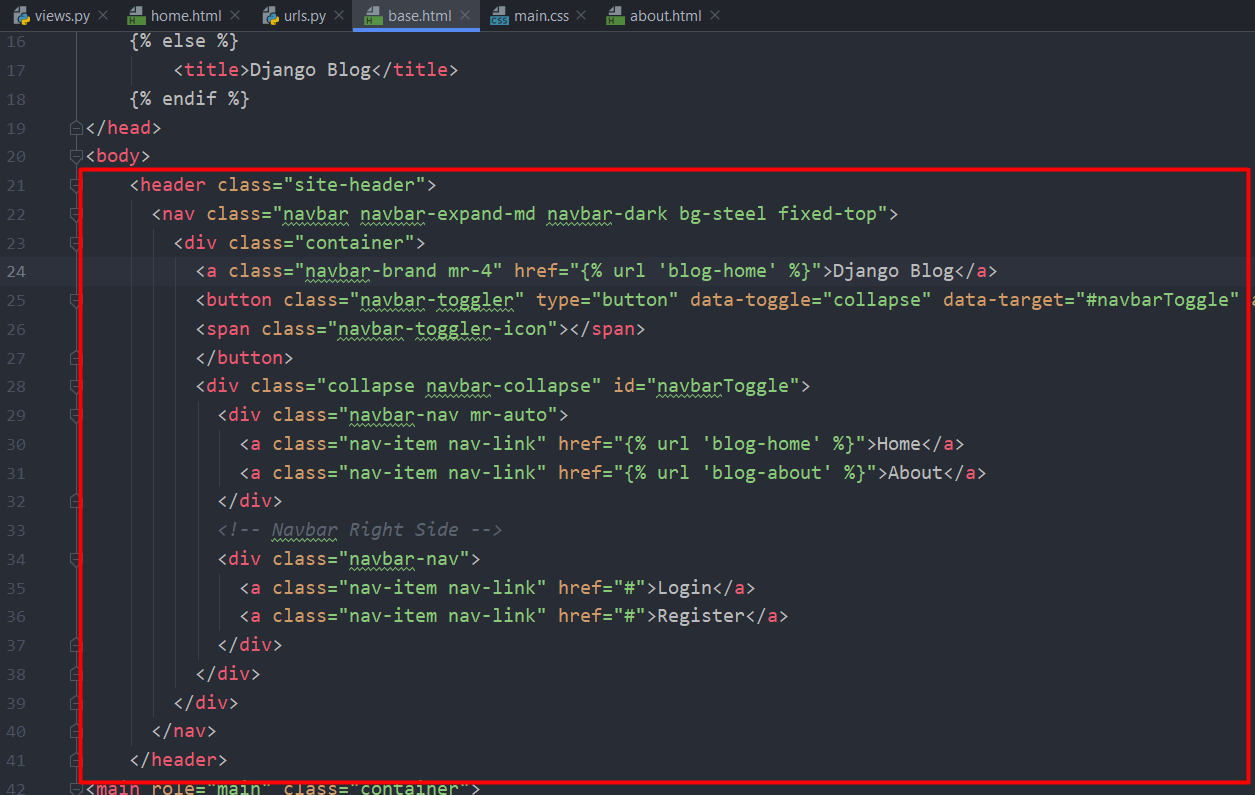
Bottom of Form

Top of Form

Bottom of Form

|  |  |
| --- | --- |
|  | <header class="site-header"> |
|  | <nav class="navbar navbar-expand-md navbar-dark bg-steel fixed-top"> |
|  | <div class="container"> |
|  | <a class="navbar-brand mr-4" href="/">Django Blog</a> |
|  | <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarToggle" aria-controls="navbarToggle" aria-expanded="false" aria-label="Toggle navigation"> |
|  | <span class="navbar-toggler-icon"></span> |
|  | </button> |
|  | <div class="collapse navbar-collapse" id="navbarToggle"> |
|  | <div class="navbar-nav mr-auto"> |
|  | <a class="nav-item nav-link" href="/">Home</a> |
|  | <a class="nav-item nav-link" href="/about">About</a> |
|  | </div> |
|  | <!-- Navbar Right Side --> |
|  | <div class="navbar-nav"> |
|  | <a class="nav-item nav-link" href="#">Login</a> |
|  | <a class="nav-item nav-link" href="#">Register</a> |
|  | </div> |
|  | </div> |
|  | </div> |
|  | </nav> |
|  | </header> |

We are going to add this to the base.html file , we are going this between the body tags of our base.html

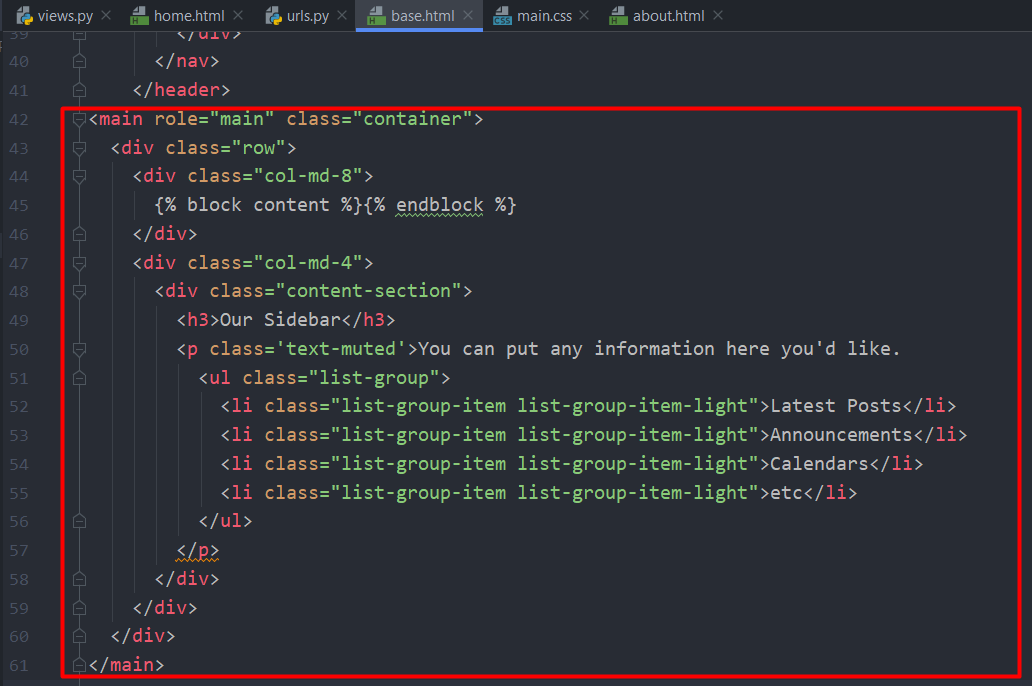


## Content Block Styling – Adding a main section for content block styling.

We are going to replace our content block section of base.html with some prewritten code to make it look nicer :

|  |
| --- |
| <main role="main" class="container"> |
|  | <div class="row"> |
|  | <div class="col-md-8"> |
|  | {% block content %}{% endblock %} |
|  | </div> |
|  | <div class="col-md-4"> |
|  | <div class="content-section"> |
|  | <h3>Our Sidebar</h3> |
|  | <p class='text-muted'>You can put any information here you'd like. |
|  | <ul class="list-group"> |
|  | <li class="list-group-item list-group-item-light">Latest Posts</li> |
|  | <li class="list-group-item list-group-item-light">Announcements</li> |
|  | <li class="list-group-item list-group-item-light">Calendars</li> |
|  | <li class="list-group-item list-group-item-light">etc</li> |
|  | </ul> |
|  | </p> |
|  | </div> |
|  | </div> |
|  | </div> |
|  | </main> |

Our newly modified base.html should look something like this

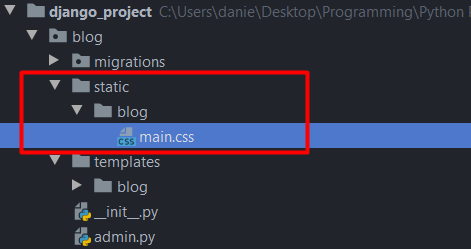


Notice we added {% block content %} {% endblock %} in a <div> tag

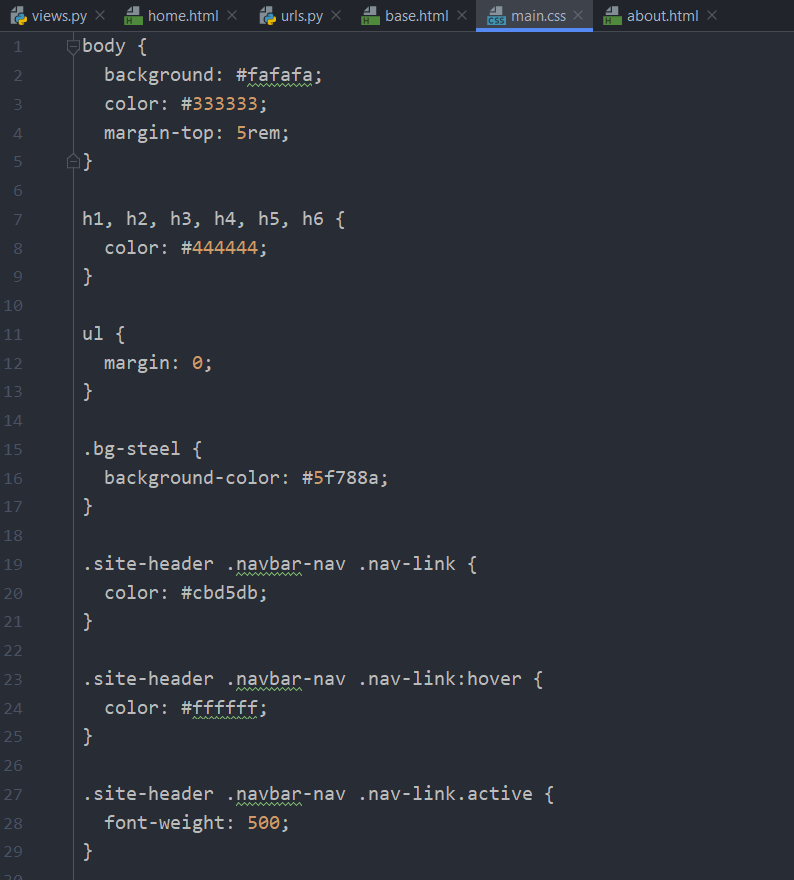
## Adding CSS files

In django , static files like .CSS and .JAVASCRIPT have to be located in a static directory within our app

eg. myapp/static/myapp/



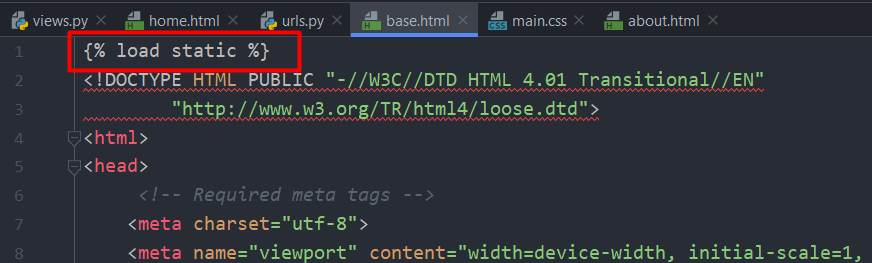
The main.css file looks like this:



### How to include .CSS file

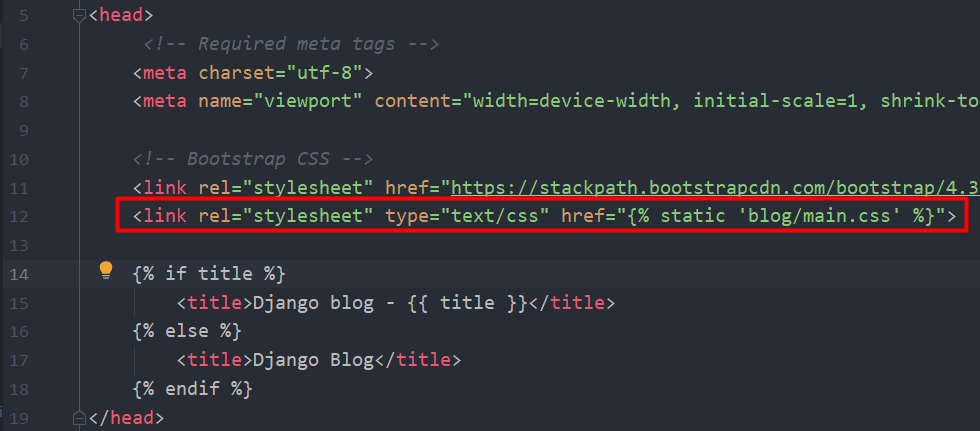
1. We load in our .CSS file from our static directory by adding

{% load static %} to the top of our base.HTML file.



1. We add the next line of code:

<link rel="stylesheet" type="text/css" href="{% static 'blog/main.css' %}">



This basically gives style to our base.html file

### Styling the articles with our new .CSS file

Our new .CSS file has styling for each article posted on our blog we just need to add the functionality to our page.



We are going to add this line within our for loop. Note that it has the second part of our for loop code already inserted

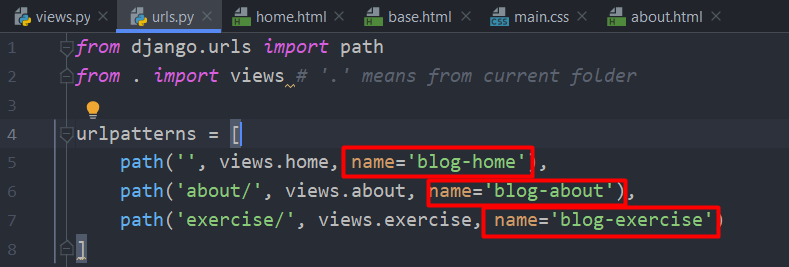
{% extends 'blog/base.html' %}  
{% block content %}  
 {% for post in posts%}  
 <article class="media content-section">  
 <div class="media-body">  
 <div class="article-metadata">  
 <a class="mr-2" href="#">{{ post.author }}</a>  
 <small class="text-muted">{{ post.date\_posted }}</small>  
 </div>  
 <h2><a class="article-title" href="#">{{ post.title }}</a></h2>  
 <p class="article-content">{{ post.content }}</p>  
 </div>  
 </article>  
 {% endfor %}  
{% endblock content %}

## Making our templates more flexible

We are going to edit some “href=” variables in the navigation part of our base.html code in order to make site navigation much easer if we change the paths of our website in the future.



We modified the “href” to link us to the page name not the page path so in the future if we want to move Django blog to a different path we won’t have to manually modify the path because we give it it’s name. Example below:



So basically we are referecing the name of the page not the path. Django will handle the hardwork.

## Adding a HYPER-LINK (Clickable thingie magic)

So if you want to add a new Hyperlink to your website like : Login , Register etc. Just do the following.

<a class="nav-item nav-link" href="#">Login</a>  
<a class="nav-item nav-link" href="#">Register</a>  
<a class="nav-item nav-link" href="https://www.youtube.com/watch?v=dQw4w9WgXcQ">Magic</a>

I don’t know exactly how it works , it might be some Bootstrap or CSS magic. But basically you add another <a> </a> tag (which is a hyperlink tag according to the interwebs) and add your stuff.

On the HREF I added a link to a very special youtube video.

# Creating the database

In order to advance we have to create a database by writing the following commands into the CMD prompt or terminal.

python manage.py migrate

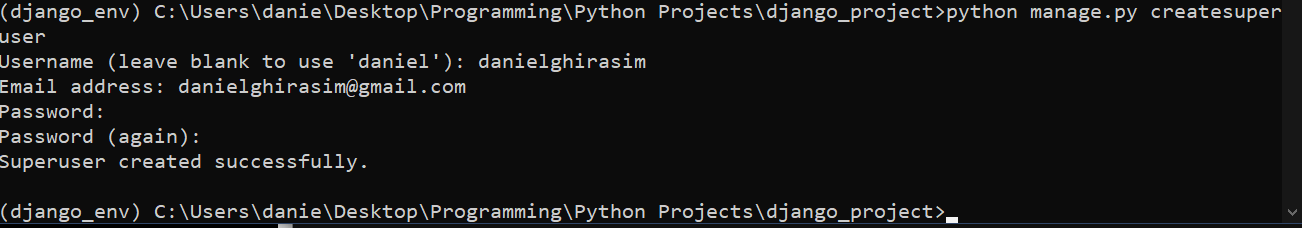
This command basically creates an SQlite3 database for us (more on that later)

# Admin Page

## Creating an admin user

First we have to create an admin user using CMD prompt or terminal.

python manage.py createsuperuser



After this you can access the admin page by going to:

localhost:8000/admin/ - And typing in the admin account you just created

# Working with Databases in Django

## Some information

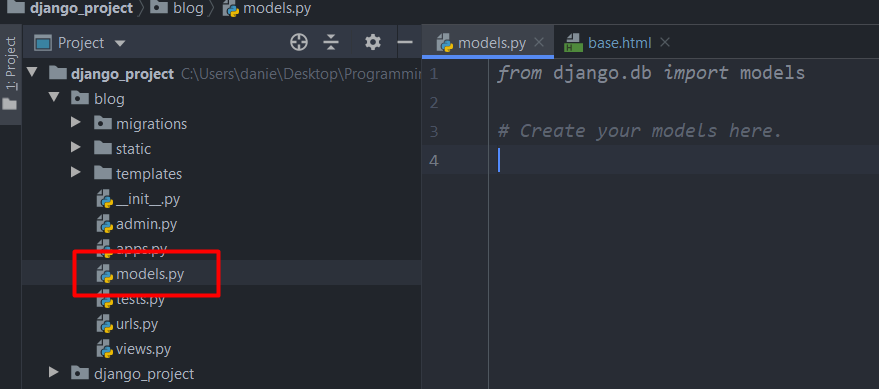
Django has it’s own built in ORM (Object Relational Mapper) which basically allows you to access the database data in an object oriented manner.

What’s cool about this is that it allows you to use different databases without changing your code

Django supported databases:

* [PostgreSQL](https://docs.djangoproject.com/en/3.0/ref/databases/#postgresql-notes)
* [MariaDB](https://docs.djangoproject.com/en/3.0/ref/databases/#mariadb-notes)
* [MySQL](https://docs.djangoproject.com/en/3.0/ref/databases/#mysql-notes)
* [Oracle](https://docs.djangoproject.com/en/3.0/ref/databases/#oracle-notes)
* [SQLite](https://docs.djangoproject.com/en/3.0/ref/databases/#sqlite-notes)

We can represent our databases as classes but in Django we call the **models .** We can have a look at our models.py module here :



Here we are going to make a post model (basically a post class) which will inherit from django.Model class

**Each class is going to be it’s own table in the DATABASE and each attribute will be a different field in the database.**

## Adding databases

We refer to databases as models in Django .

Creating a database is as easy as adding a new class in models.py that inherits from django.models and it should look something like this

*from* django.db *import* models  
*from* django.utils *import* timezone  
  
*# Create your models here.*"""Each class is going to be a table in our database"""  
*class* Post(models.Model):  
 *"""Each attribute is a field in the database"""* title = models.CharField(max\_length=100)

content = models.TextField()  
 date\_posted = models.DateTimeField(default=timezone.now)

#max\_lenght = length of charfield

# date\_posted = models.DateTimeField(auto\_now=*True*) auto\_now = True sets our date\_posted to the time it was posted but we are going to use the upper example because we want to store the timezone aswell . timezone.now is a function but we didn’t put () yet because we don’t want to execute it just yet.

We are also going to create a user attribute but since we already have a user database we are going to import that in this file.

Users and Posts are going to have a relationship , specifically this **will be called a 1 to many relationship** because a User can have multiple post and a post can’t have multiple users (here atleast) and to do this in django we use a **Foreign Key**

We are going to import Users since it already has a database created by django . The import location is :

*from* django.contrib.auth.models *import* User

After that we are going to add:

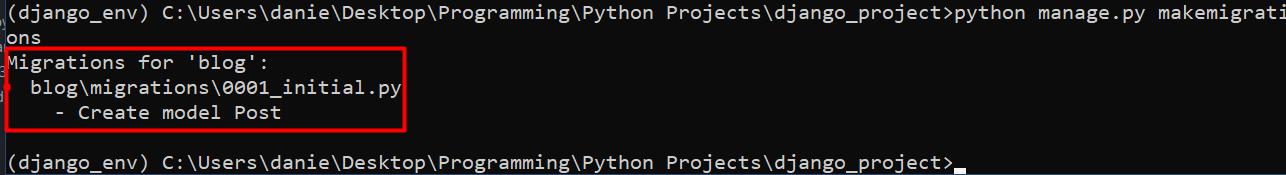
author = models.ForeignKey(User, on\_delete=models.CASCADE())

# on\_delete = models.CASCADE() means that when you delete a USER it will delete all his posts.

In the end our models.py should look something like this” *from* django.db *import* models  
*from* django.utils *import* timezone  
*from* django.contrib.auth.models *import* User  
  
*# Create your models here.*"""Each class is going to be a table in our database"""  
*class* Post(models.Model):  
 *"""Each attribute is a field in the database"""* title = models.CharField(max\_length=100)  
 content = models.TextField()  
 date\_posted = models.DateTimeField(default=timezone.now)  
 author = models.ForeignKey(User, on\_delete=models.CASCADE)

After all this to make the changes we have to make migrations (Update the database).

We do this by entering python manage.py makemigrations into the terminal or command prompt and it should look something like this :

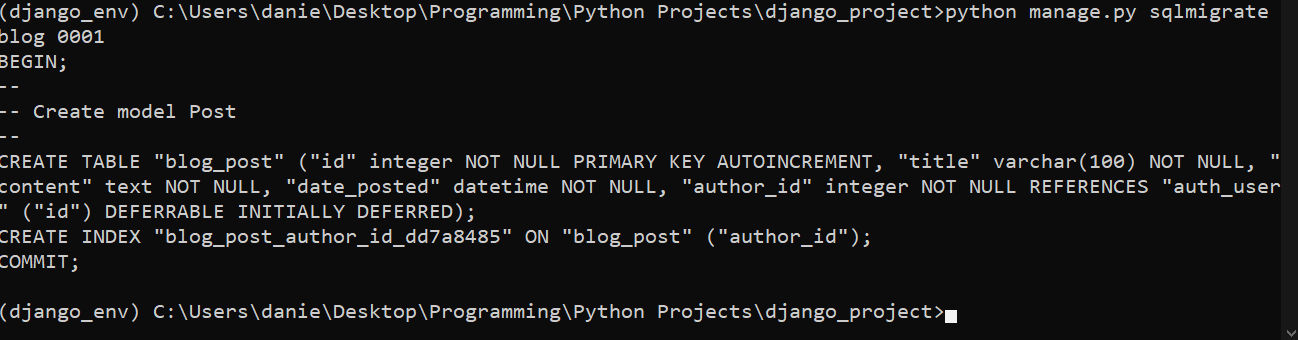


## SQL Code command (views the actualy SQL CODE)

To view the actual SQL code that is run we have to type :

python manage.py sqlmigrate blog 0001

It should look something like this:



Django actually takes the information we entered and runs the command in SQlite 3 to make a table and field for our class , which makes our life much easier.

## Using CMD prompt to access databases

To access various information about our databases we can enter the next line into the command prompt:

**python manage.py shell**

**to exit the shell type : exit()**

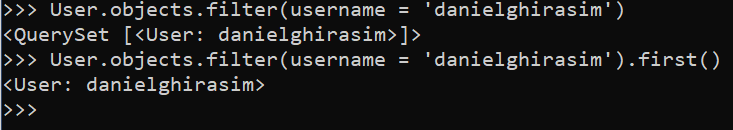
This will open a python shell in our command prompt where we can enter commands:

from djang.contrib.auth.models import User (this imports our user database)

**User.objects.all() – Will list the contents of our User database**

**User.objects.first() – or last() will list the first or last entry in our database**

**User.objects.filter(username=’user’) – will filter for entered username**



We can also use:

User.objects.filter(username=’danielghirasim) and it will give us a querry set . If we add first() to the end of the function it will give us the first username

We can set the upper mentioned function to a variable and then we can access more information from it ex:



Some commands we can run on this:

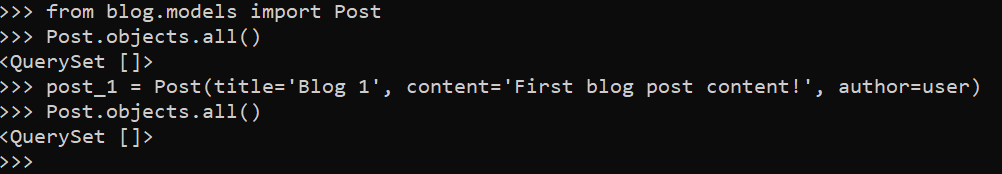
user.id – shows id

user.pk – shows primary key (id)

## Creating a post from Terminal

First we import our Post class that we just made .

from blog.models import Post



So in the picture above we created a post . Notice that we haven’t filled anything to our date\_created field but since we set it to timezone.now, this should automatically get the time we created post\_1.

After that we querry the Post database again and we can see we that the querry is still empty, this is because we have to migrate our post\_1 to the database by using the following command.

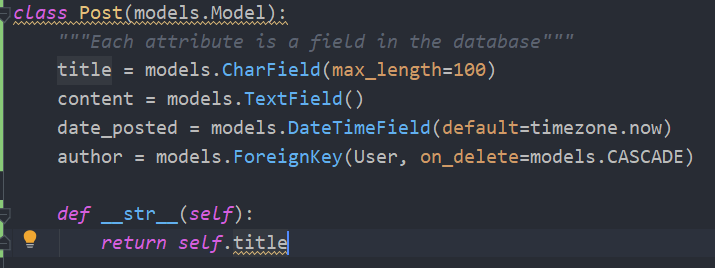


After that we querry the DB again and we should see that it was added.



### Getting more descriptive querries

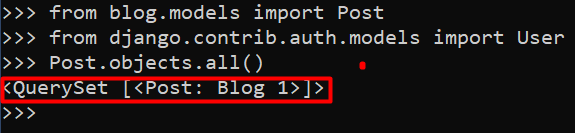
To get more descriptive querries we have to add some code to our Post class we created before.



This dunder method basically applies the \_\_str\_\_ method to the Post class (self = Post class) and returns self.title

So it basically returns a string of Post.title.

After that we have to restart the shell by using exit() and reentering the imports and the commands and the querry should look like this



To acess all of our Post content we are going to set the post to a variable:

**post = Post.objects.first() – gives us the first post**

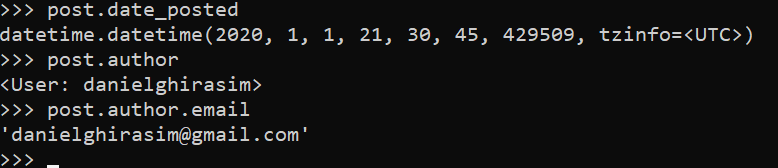
**post.content – prints content**

**post.title – prints title**

**post.author – prints author**

We can even access author data from here with the example below.

**post.author.email – this is gonna print the authors email so we don’t have to run a separate command for that aswell.**



### Getting all USER posts with a simple command

We can get all the posts that a user posted by doing the following:

user = User.objects.get(username=’username’)

user.post\_set.all() 🡪 This command will show all the posts created by the user we choose

We can also create posts while we have this user as a variable by using the following command line:



Using this method we don’t have to run the save function anymore and we don’t have to exit out of the shell. Also notice we didn’t enter the user in the function parameters since we already have the user defined.

## Connecting our Post database to our website

Here we are going to replace our dummy data with our data within our database.

We are going to do this the following way:

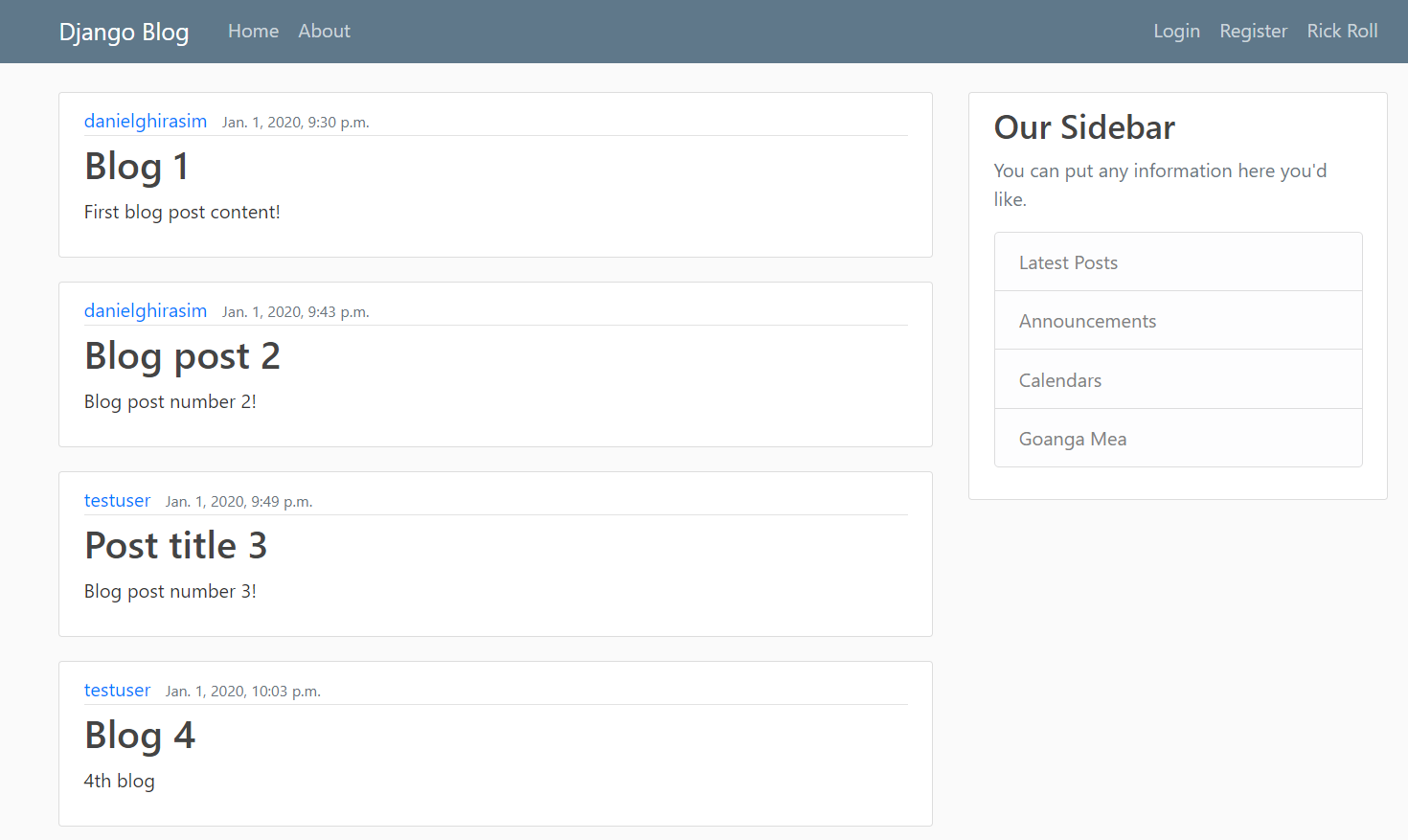
1. We are going to import our Post database into our views.py module

*from* .models *import* Post

*def* home(request):  
 context = {  
 'posts': Post.objects.all()  
 }  
 *return* render(request, 'blog/home.html', context)

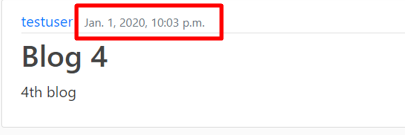
As long as our dummy data’s dictionary keys were the same as our database fields everything should work fine without having to modify anything in our home.html file . If they are different we just have to go into our home.html and modify the FOR LOOP.

Aaaaaand it works !!!



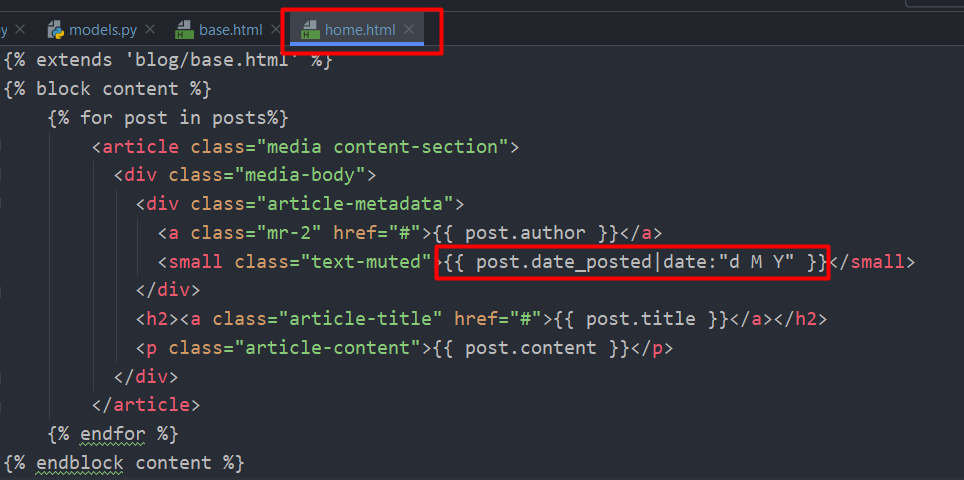
## Changing how our Date Time looks

We are going to change how our date time looks to make it a little bit neater.



To do this we have to go to our home.html file and modify a bit of code by using Django documentation of dates as a reference (<https://docs.djangoproject.com/en/3.0/ref/templates/builtins/#date>):

The code should look like this:



After we modified it , it should look something like this :

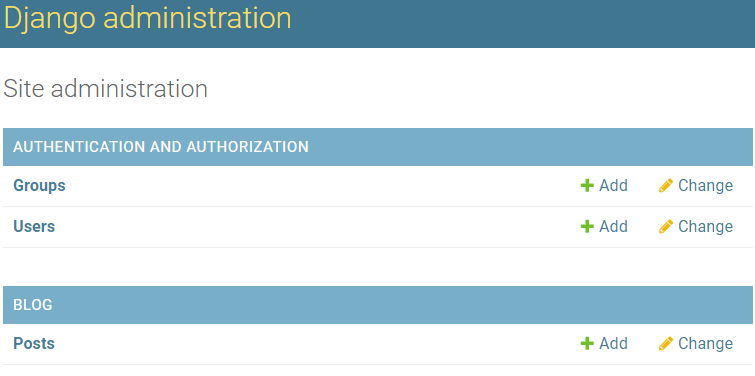


## Registering Databases (models) to the admin page

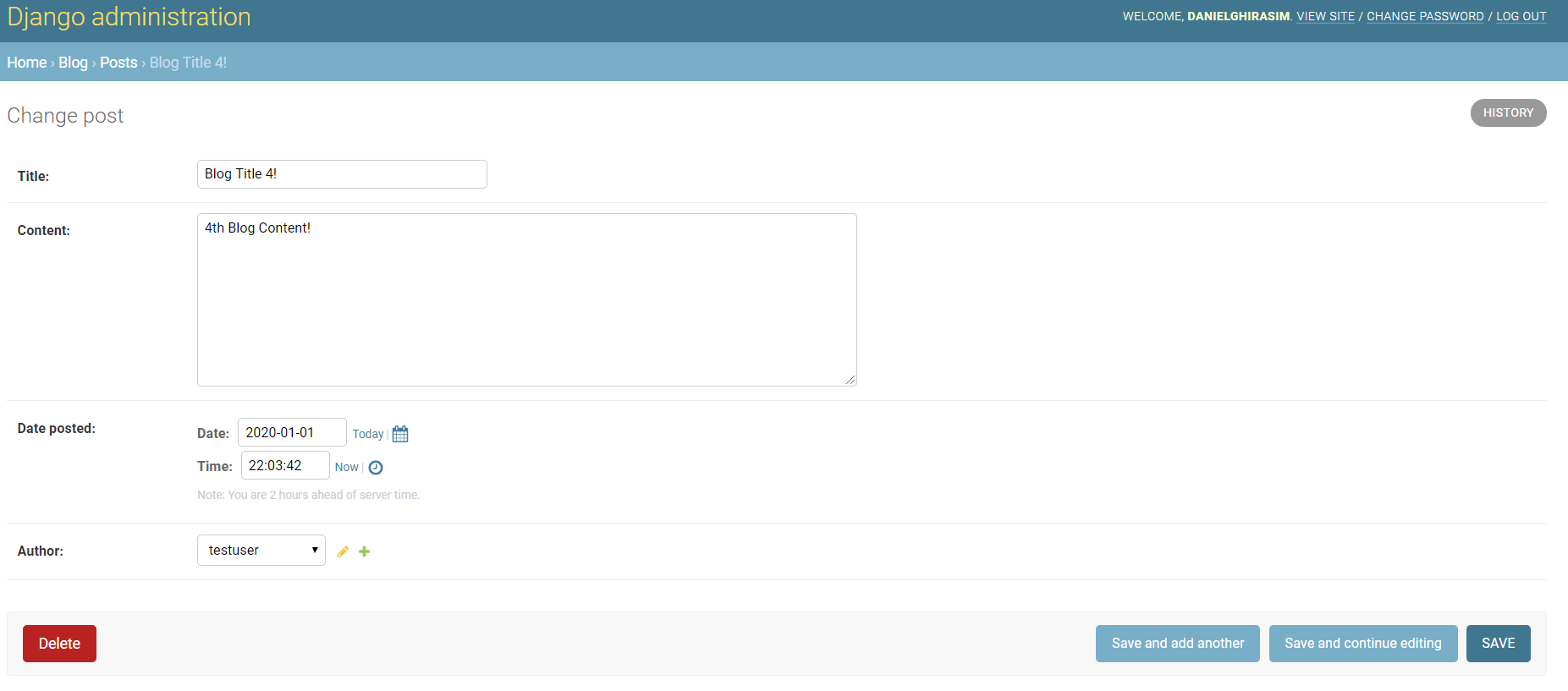
So when we enter the admin page we want to register our Post database in order to modify it . This is a very simple process thanks to Django is as easy as going into the admin.py module and entering a few lines of code.

*from* django.contrib *import* admin  
*from* .models *import* Post  
  
admin.site.register(Post)

This will add an editable Post section to our admin page and it’s freakin’ cool how easy it is.



The editing section looks like this:

s

# User Registration

In this part we are going to learn how to make forms and validate user input to make a user registration page.

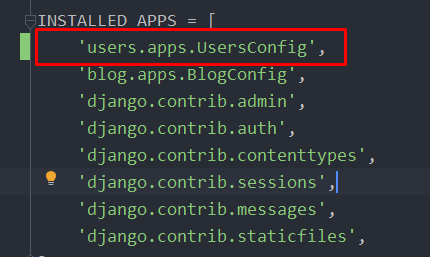
The best thing to do is to create a new app inside of our project for this.

## Making a new app for users

So we are going to go into our terminal and write:

Python manage.py startapp users

This will create a users app for us and we have to put the config class into our main folders settings.py.



**Don’t forget the comma since this is a list.**

## Adding a new path for register.html

We are going to add path to our register page in our main directory in the urls.py file but this time we are going to directly import the views.py module from our newly made app

*from* django.contrib *import* admin  
*from* django.urls *import* path, include  
*from* users *import* views *as* user\_views  
  
urlpatterns = [  
 path('admin/', admin.site.urls),  
 path('register/', user\_views.register, name='register'),  
 path('', include('blog.urls')),  
]

This tells the website that whenever somebody accesses mysite.com/register, the website needs to run our register function from users.views

## Creating a submission form

Luckily we don’t have to reinvent the wheel. Django already has a registration form premade for us with all the bells and whistles regarding security.

We are going to navigate to our users/views.py and do the following

*from* django.shortcuts *import* render  
*from* django.contrib.auth.forms *import* UserCreationForm  
*# UserCreationForm is a premade Django creation form class.  
  
def* register(request):  
 form = UserCreationForm() *# We are gonna make an instance for the class we just imported  
 return* render(request, 'users/register.html', {'form': form})

We made an instance of the UserCreationForm class and assigned it to a variable inside our register function.

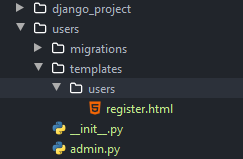
And within the function we make a dictionary with a key called ‘form’ and the value is the instance of our class.

.

Next we have to take care of our 'users/register.html' which is not yet defined.

## Creating a user template

We are going to do the same thing as we did in our blog app . We are going to a create a folder within our Users app called template and within that folder another folder called users (this is the Django convetion). It should look something like this:



Next we are going to work on register.html

## Register.html

Here we are going to edit our register.html.

In order to make our lives easier we are going to copy our code from the about.html (from blog app) which extends our base.html

A nice thing about Django framework is that we can use the base.html file from other apps as well like we are going to do below.

{% extends 'blog/base.html' %}  
{% block content %}  
 <h1>About Page!</h1>  
{% endblock content %}

We are going to edit this piece of code until we get what we desire from the register.html page.

Our modified code should look something like this

{% extends 'blog/base.html' %}  
{% block content %}  
 <div class="content-section">  
 <form method="POST"> # We will come back to this later  
 {% csrf\_token %}  
 <fieldset class="form-group">  
 <legend class="border-bottom mb-4">Join Today</legend>  
 {{ form }}  
 </fieldset>  
 <div class="form-grou">  
 <button class="btn btn-outline-info" type="submit" >Sign Up</button>  
 </div>  
 </form>  
 <div class="border-top pt-3">  
 <small class="text-muted">  
 Already Have an Account? <a class="ml-2" href="#"> Sign in </a>  
 </small>  
 </div>  
 </div>  
{% endblock content %}

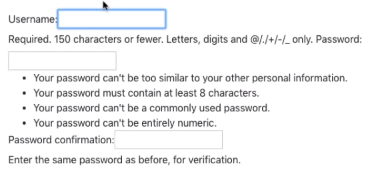
## {% csrf\_token %} - Security

This is a Django security must. Basically this line of code prevents us from some sort of online attack that can be run on registration pages (don’t really have details on it). Without this our form will not work.

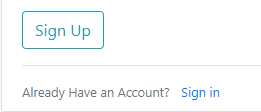
## Register page code meaning

The classes you see in each tag come from **BOOTSTRAP** it’s pretty cool (it basically magic autoformats your stuff.)

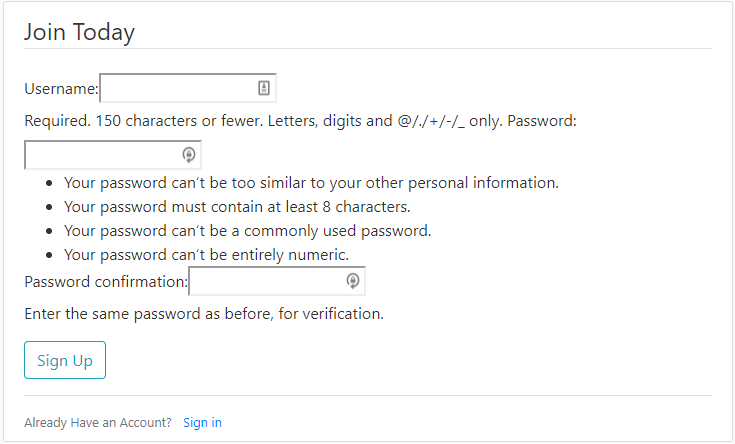
{{ form }} – This makes our form look like this:



Our Sign-Up and already have an account? Button look like this:

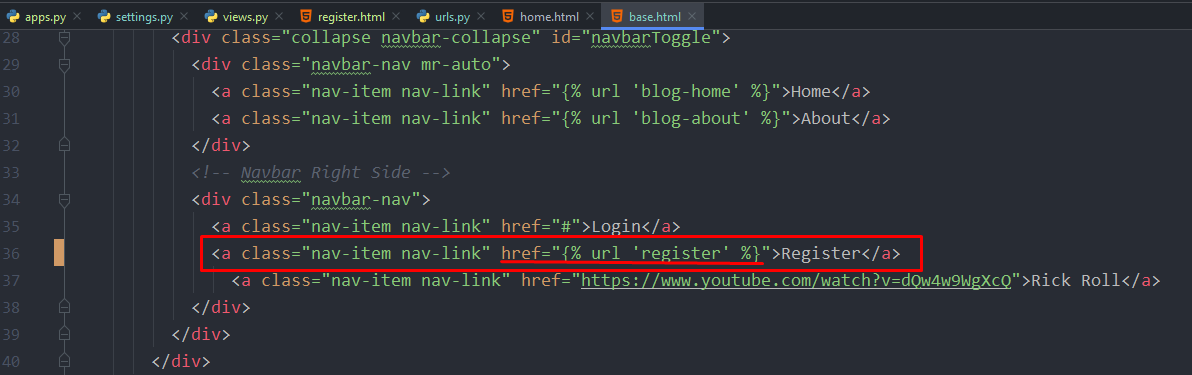


And our overall register page looks like this (for now):



## Adding register.html to our base.html as a ref

Next we will add our register.html url to the base.html file so whenever we click on the register button on our homepage it will redirect us to the register.html website.



This is as simple as adding {% url ‘register.html’ %} at href.

## Explaining register.html

Register.html

{% extends 'blog/base.html' %}  
{% block content %}  
 **<**div class="content-section"**>  
 <**form method="POST"**>** {% csrf\_token %}  
 **<**fieldset class="form-group"**>  
 <**legend class="border-bottom mb-4"**>**Join Today**</**legend**>** {{ form.as\_p }}  
 **</**fieldset**>  
 <**div class="form-group"**>  
 <**button class="btn btn-outline-info" type="submit" **>**Sign Up**</**button**>  
 </**div**>  
 </**form**>  
 <**div class="border-top pt-3"**>  
 <**small class="text-muted"**>** Already Have an Account? **<**a class="ml-2" href="#"**>** Sign in **</**a**>  
 </**small**>  
 </**div**>  
 </**div**>**{% endblock content %}

Explained:

1. We extend this from our base.html file . A cool thing in Django is that we can extend from other apps like we did here, we extended from our blog/base.html.
2. After that we are going make a new div with some custom bootstrap classes. Within that div we are going to make a <form> tag with method=’POST’. This refers to the http.post method which is a way of sending information to the website. (request is when we get information, post is when we send information)
3. Next we add our csrf\_token which is a django security measure for each form. Our website will not run without this.
4. We add a fieldset tag, add some bootstrap formatting to it then we pass in our form variable. Notice we added {{ form.as\_p }} this basically formats our fieldset a bit nicer. (We will further improve on this with crispy forms later on)

## Creating a register view function

Here we are going to create a function that will render us a user registration page. We don’t have to reinvent the wheel here so we are going to use a built-in form from Django.

from django.shortcuts import render  
from django.contrib.auth.forms import UserCreationForm  
*# UserCreationForm is a premade Django creation form class.*def register**(**request**)**:  
 form = UserCreationForm**()** *# We are gonna make an instance for the class we just imported* return render**(**request, **'users/register.html'**, **{'form'**: form**})**

We don’t have our register.html template created yet and also we are going to pass in our form to a dictionary so we can access it within our register.html.

Now we are going to make the register.html template. As by Django convention we are going to create the following directory tree:

Users>templates>users>register.html

## Setting the path to our register page

We could have created another urls.py module in our Users app like we did with our blog and then include() in the Django\_project/urls.py but we are going to do it another way for flexibility.

We are going to navigate to our Django\_project/urls.py and import our users/views.py

from users import views as user\_views

We imported this as user\_views in order to avoid future conflicts if we import another views module.

After this we are going to include this in our urlpatterns list.

urlpatterns = **[** path**('admin/'**, admin.site.urls**)**,  
 **path('register/', user\_views.register, name='register'),**  
 path**(''**, include**('blog.urls'))**,  
**]**

## Taking a look at our current register() function

def register**(**request**)**:  
 form = UserCreationForm**()** *# We are gonna make an instance for the class we just imported* return render**(**request, **'users/register.html'**, **{'form'**: form**})**

So the problem with our current register method is that every time someone makes a request to our register page they will get a empty UserCreationForm and when they press sign-up nothing happens.

There are different type of http requests we can have, the most common we used until now are get() request which just “get” the information from the website. The one we are going to use now is the POST request . Remember we typed method=POST in our register.html page this means that the method we want to use for those fields is post and since we didn’t specify where we want to post that data, it actually just posts it back to this route.

So what are we going to do now is add some conditional statements so when we receive a post request it validates that data and if it’s a get request then it displays our register page.

if request.method == **'POST'**:  
 form = UserCreationForm**(**request.POST**)**else:  
 form = UserCreationForm**()** *# We are gonna make an instance for the class we just imported*return render**(**request, **'users/register.html'**, **{'form'**: form**})**

So once again. If we get a POST request then it instantiates a UserCreationForm with that post data and if we get any other type of request it just instantiates an empty form. The next step is validating our usercreationform

## Validating our user creating form

def register**(**request**)**:  
 if request.method == **'POST'**:  
 form = UserCreationForm**(**request.POST**)** if form.is\_valid**()**:  
 username = form.cleaned\_data.get**('username')**

**# more to add** else:  
 form = UserCreationForm**()** *# We are gonna make an instance for the class we just imported* return render**(**request, **'users/register.html'**, **{'form'**: form**})**

In order to validate our user data we are going to use the is\_valid() function which basically tells us if our form is valid then we are going to grab that data. To do this we simply type username = form.cleaned\_data.get(‘username’) and our validated form data will be in form.cleaned\_data DICTIONARY and this will convert the data nicely to python for us.

## Flash messages

Flash messages are printed out messages in our website that tells us if something was successful, unsuccessful etc.

First thing is to import messages

from django.contrib import messages

Types of flash messages:

Messages.debug

Messages.info

Messages.success

Messages.warning

Messages.error

Since we are going to display a message if our form is valid we are going to use the success message

def register**(**request**)**:  
 if request.method == **'POST'**:  
 form = UserCreationForm**(**request.POST**)** if form.is\_valid**()**:  
 username = form.cleaned\_data.get**('username')** messages.success**(**request, **f'Account {**username**} created succesfully.')  
 # more to add** else:  
 form = UserCreationForm**()** *# We are gonna make an instance for the class we just imported* return render**(**request, **'users/register.html'**, **{'form'**: form**})**

Our messages take in a request parameter and we can also add a message.

## Redirect function

If our uservalidation went ok we are going to redirect our users to the homepage. To do this we want to import the redirect function from Django by :

from django.shortcuts import render, redirect

def register**(**request**)**:  
 if request.method == **'POST'**:  
 form = UserCreationForm**(**request.POST**)** if form.is\_valid**()**:  
 username = form.cleaned\_data.get**('username')** messages.success**(**request, **f'Account {**username**} created succesfully.')** return redirect**('blog-home')** else:  
 form = UserCreationForm**()** *# We are gonna make an instance for the class we just imported* return render**(**request, **'users/register.html'**, **{'form'**: form**})**

This should all work but we have to update our template to work with our flash messages. We are going to add this to our base template so any flash messages appear on every page that inherits from base.

## Adding flash messages to our base.html

In order to display the flash messages we have to add the functionality to our base.html.

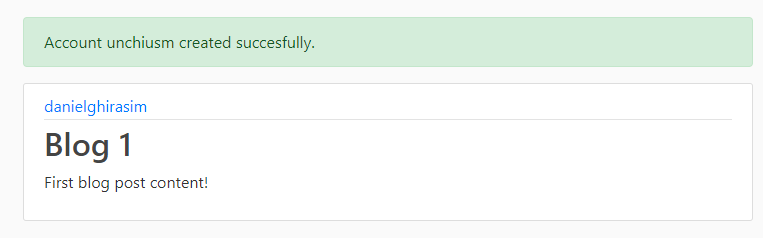
**<**main role="main" class="container"**>  
 <**div class="row"**>  
 <**div class="col-md-8"**>** {% if messages %}  
 {% for message in messages %}  
 **<**div class="alert alert-{{ message.tags }}"**>** {{ message }}  
 **</**div**>** {% endfor %}  
 {% endif %}  
 {% block content %}{% endblock %}  
 **</**div**>  
 <**div class="col-md-4"**>  
 <**div class="content-section"**>  
 <**h3**>**Our Sidebar**</**h3**>**

A good place to place these flash messages are just above our content. A nice thing about Django that the message names coincide with bootstrap message formats. Se we can type the following:

**<**div class="alert alert-{{ message.tags }}"**>**

And bootstrap will automatically recognize it if it’s a success or failure message.

This is how a success method looks with BOOTSTRAP formatting.



Notice we got redirected back to the homepage with a neat little message.

We didn’t actually create a new user yet we are going to that below. Here we just tested if our uservalidation works and we know that our forms are validating correctly.

## Saving the user if the form validates

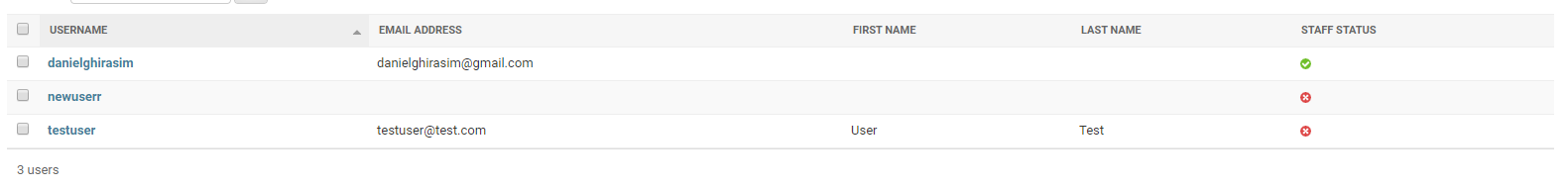
Next we want to save the form if the data validates.

Saving the user it’s just as simply as adding:

Form.save() # this will automatically hash the password for us and all the other background stuff.

def register**(**request**)**:  
 if request.method == **'POST'**:  
 form = UserCreationForm**(**request.POST**)** if form.is\_valid**()**:  
 form.save**()** username = form.cleaned\_data.get**('username')** messages.success**(**request, **f'Account {**username**} created succesfully.')** return redirect**('blog-home')** else:  
 form = UserCreationForm**()** *# We are gonna make an instance for the class we just imported* return render**(**request, **'users/register.html'**, **{'form'**: form**})**

And now the user was created. We can check in the admin panel too.



## Adding E-mail field to our registration field

If we go to our current register.html we can see that we print out all the forms as one:

**<**fieldset class="form-group"**>  
 <**legend class="border-bottom mb-4"**>**Join Today**</**legend**>** {{ form.as\_p }} # We print out as one  
**</**fieldset**>**

So how do we add another field to this?

To do this we are going to need to create a new form that inherits from our UserCreationForm. Next we are going to create a new file in our Users directory called forms.py

### Forms.py – explained

from django import forms  
from django.contrib.auth.models import User  
from django.contrib.auth.forms import UserCreationForm  
  
  
class UserRegisterForm**(**UserCreationForm**)**:  
 email = forms.EmailField**(**required=True**)** class Meta:  
 model = User  
 fields = **['username'**, **'email'**, **'password1'**, **'password2']**

First we have to do some imports.

1.We are going to import forms so we can add our email a Django premade form

2.We import our User model (database) because we want this from to interact with that model.

3.We import the default UserCreationForm because we want to inherit from that.

Next.

We are going to make a new class called UserRegisterForm that inherits from UserCreation form.

We are going to add a variable called email that is equal to from.EmailField(required=True). This is a django form for emails and required=True tells Django that this is a required field.

### Class Meta

Explaining meta.

This class meta gives us a nested namespace for configurations and keeps the configurations in oneplace.

Within this class we are going to specify the model we want this form to interact with, in our case User

class Meta:  
 model = User  
 fields = **['username'**, **'email'**, **'password1'**, **'password2']**

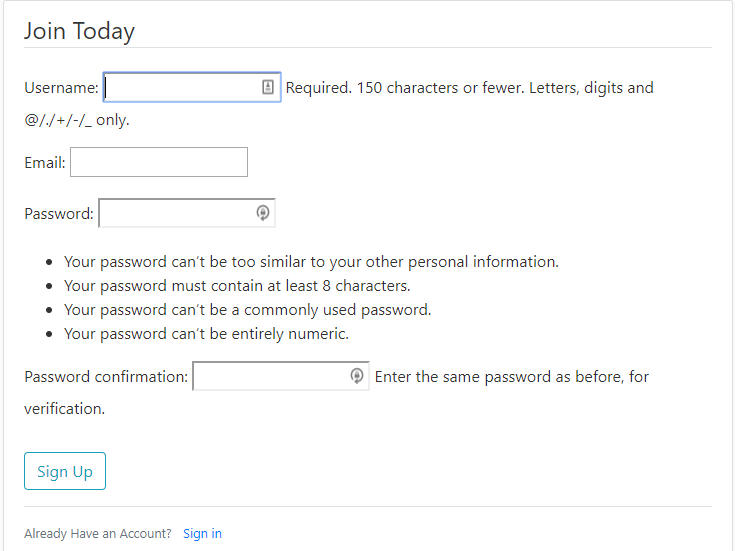
And next we are going to set the fields . These are the fields that are going to be shown in our form and in what order. (pass1 and pass2 are the matching passwords)

### Changing default form with our new form

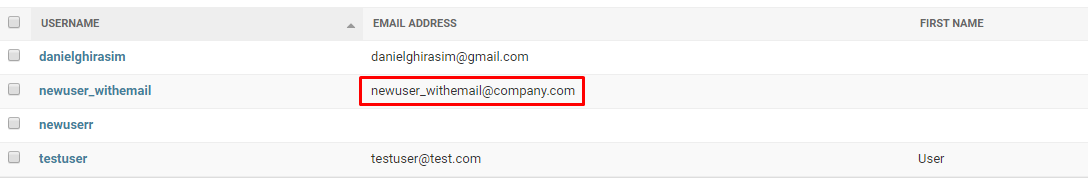
Now we are going to use this form in our views.py and import our new UserRegisterForm class and change it with the old registration form.

from django.shortcuts import render, redirect  
from django.contrib import messages  
from .forms import **UserRegisterForm**  
*# UserCreationForm is a premade Django creation form class.*def register**(**request**)**:  
 if request.method == **'POST'**:  
 form = **UserRegisterForm(**request.POST**)** if form.is\_valid**()**:  
 form.save**()** username = form.cleaned\_data.get**('username')** messages.success**(**request, **f'Account {**username**} created succesfully.')** return redirect**('blog-home')** else:  
 form = **UserRegisterForm()** *# We are gonna make an instance for the class we just imported* return render**(**request, **'users/register.html'**, **{'form'**: form**})**

Now if we reload our browser we can see the changes took place.



Now if create new user it should save it’s email address to our Users database.



## Crispy Forms – Fixing our forms

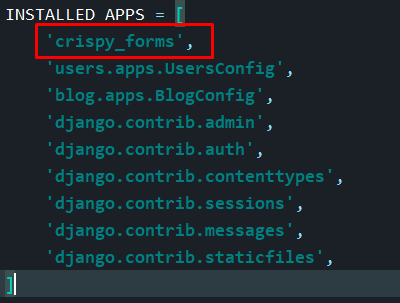
We are going to use a 3rd party library in order to make our forms look nicer from within our register.html file. Crispy forms will do all the formatting for us and also will highlight the in red a field if something went wrong.

Crispy forms will style our forms in a bootstrap fashion.

1. Install crispy forms.

Pip install Django-crispy-forms

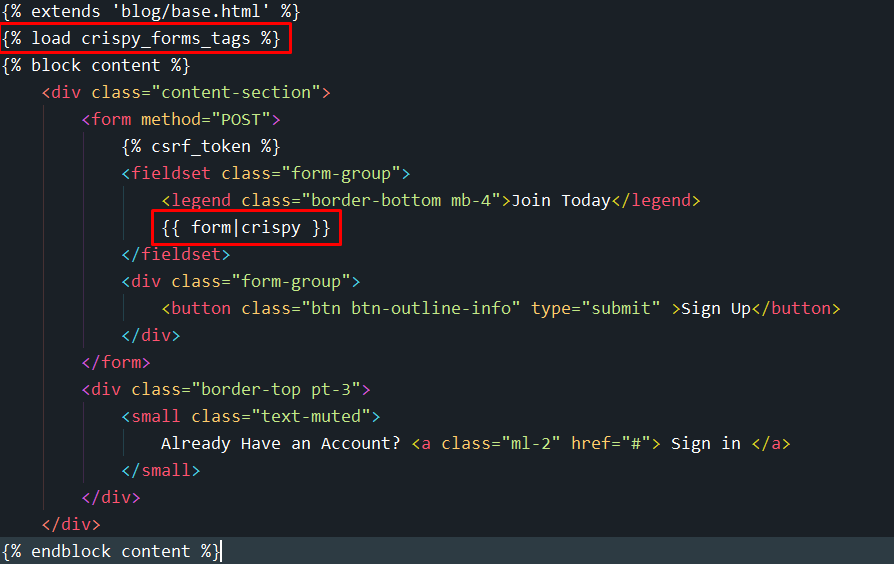
1. Once this is installed we need to tell Django that this is an installed app



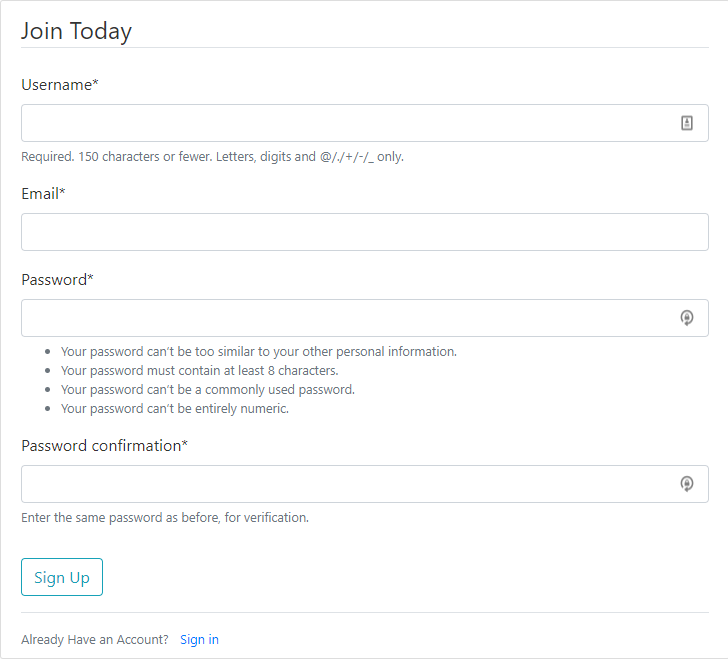
1. Then we are going to tell crispy forms which CSS framework we want to use. It defaults to BOOTSTRAP 2 but here we are using BOOTSTRAP 4. In order to do this we are going to scroll to the bottom of our settings page and add the following:

CRISPY\_TEMPLATE\_PACK = **'bootstrap4'**

1. Next we are going to load this into our template and use it. Open register.html



We have to add {% load crispy\_forms %} just above our block content and modify our form.p variable to {{ form|crispy}} and if we save this we can see that our registration page looks very nice now.



# Login-Logout System

Here we are going to design our login and logout functionality of our website. Luckily Django has most of this functionality taken care of by default.

We are going to start by importing Django’s default login and logout views into our main project urls.py module.

from django.contrib import admin  
from django.urls import path, include  
from users import views as user\_views  
from django.contrib.auth import views as auth\_views  
  
urlpatterns = **[** path**('admin/'**, admin.site.urls**)**,  
 path**('register/'**, user\_views.register, name=**'register')**,  
 path**('login/'**, auth\_views.LoginView.as\_view**()**, name=**'login')**,  
 path**('logout/'**, auth\_views.LogoutView.as\_view**()**, name=**'logout')**,  
 path**(''**, include**('blog.urls'))**,  
**]**

LoginView and Logout View are classed based views. We haven’t seen these yet but will make some of our own later. These will handle the form and logic all for us but they won’t handle the templates. Now we are going to tell Django where to look for our login and logout template (by default it looks in a foldername registration/login.html and logout.html)

path**('login/'**, auth\_views.LoginView.as\_view**(**template\_name=**'users/login.html')**, name=**'login')**,  
path**('logout/'**, auth\_views.LogoutView.as\_view**(**template\_name=**'users/logout.html')**, name=**'logout')**,

Here we tell Django to look for the templates in the specified directory.

## Creating login.html and logout.html

The next step is creating our login and logout html files in users/templates/users/…

After we created this we are going to use a slightly modified version of our registration HTML.

### Login.html

{% extends 'blog/base.html' %}  
{% load crispy\_forms\_tags %}  
{% block content %}  
 **<**div class="content-section"**>  
 <**form method="POST"**>** {% csrf\_token %}  
 **<**fieldset class="form-group"**>  
 <**legend class="border-bottom mb-4"**>**Login Page**</**legend**>** {{ form|crispy }}  
 **</**fieldset**>  
 <**div class="form-group"**>  
 <**button class="btn btn-outline-info" type="submit" **>**Login**</**button**>  
 </**div**>  
 </**form**>  
 <**div class="border-top pt-3"**>  
 <**small class="text-muted"**>** Do you want to create an account? **<**a class="ml-2" href="{% url 'register' %}"**>** Register **</**a**>  
 </**small**>  
 </**div**>  
 </**div**>**{% endblock content %}

We want to keep crispy forms in order to nicely format our login page and we modified some other stuff to properly represent a login page.

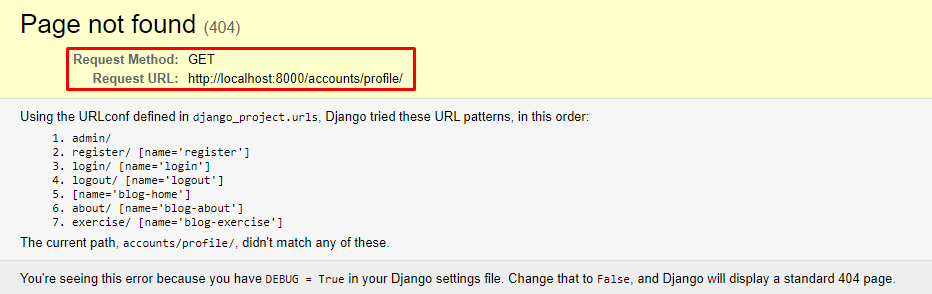
### Logout.html

{% extends 'blog/base.html' %}  
{% block content %}  
 **<**h1**>**You have been logged out.**</**h1**>  
 <**small class="text-muted"**>** Do you want to log back in? **<**a class="ml" href="{% url 'login' %}"**>** Login **</**a**>  
 </**small**>**{% endblock content %}

For now we removed the crispy forms part since we won’t be needing it here. Also we added a ‘Do you want to log back in part’ which redirects us to our login page.

### LOGIN REDIRECT to homepage and current issues

Next we are going to try to log in with our new login page but we will encounter an error.



This is because when we log in, Django redirects us to it’s default url : website.com/accounts/profile. Currently we don’t have the profile part of our site setup right now so we are going to redirect the user to the home page.

How to fix.

We are going to open up our projects settings file and add the following.

LOGIN\_REDIRECT\_URL = **'blog-home'**

Now when we log in it will redirect us to blog-home (our homepage).

Now this is actually logging in our user to the website but we can’t see that yet because we don’t have any visual feedback setup yet. If you now navigate to website.com/admin it won’t prompt you a password.

## Redirecting Newly created users to login page.

Our current registration function is redirecting the users who have created a new account to the home page. We want to change this to redirect them to the login page because this will make more sense. To do that we are going to go back to our views.py and changes some stuff around.

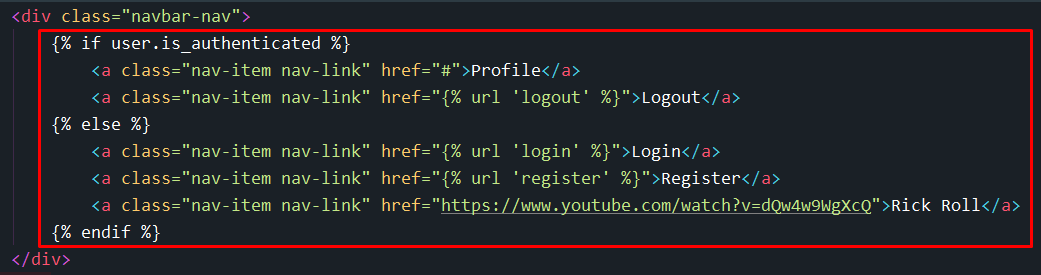
def register**(**request**)**:  
 if request.method == **'POST'**:  
 form = UserRegisterForm**(**request.POST**)** if form.is\_valid**()**:  
 form.save**()** username = form.cleaned\_data.get**('username')** messages.success**(**request, **f'Account {**username**} created succesfully! You are now able to log in.')** return redirect**('login')** else:  
 form = UserRegisterForm**()** *# We are gonna make an instance for the class we just imported* return render**(**request, **'users/register.html'**, **{'form'**: form**})**

## Changing the Navbar for logged in users

We want to change our navigation bar in Django for authenticated and non authenticated users. The auth users shouldn’t see LOGIN and REGISTER navbars only something like PROFILE or LOGOUT.

Luckily this is very easy to implement in Django.

We are going to navigate to our base.html and modify some code.



## How to put a restriction on certain routes first

The next thing we are going to do is put a restriction on certain routes so that you can only go to that site if you are currently logged in.

Ex.

Lets say you directly want to navigate to website.com/profile without being logged in. When a user does that and it’s not logged in we want to prompt a log in screen for that.

## Creating the profile.html page

We are going to head into our views.py module and make a new function that renders /profile/

def profile**(**request**)**:  
 return render**(**request, **'users/profile.html')**

Now we create the template for it.

Profile.html – we keep it simple for now

{% extends 'blog/base.html' %}  
{% block content %}  
 **<**h1**>**Current user {{ user.username }}**</**h1**>**{% endblock content %}

And let’s not forget to add the path to our urls.py module.

path**('profile/'**, user\_views.profile, name=**'profile')**,

## Accessing profile only if user is logged in

With our current setup everyone can access website.com/profile/ and we want to make a setting in order to let only authenticated users to go to website.com/profile/ this is extremely easy to do in Django.

**To do this we are going to use a login required decorator that Django provides for us.**

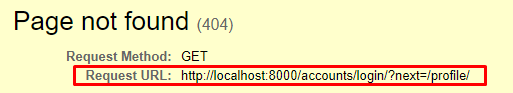
**We are going to our users/views.py module , import the decorator and add that decorator to our profile staticmethod.**

from django.contrib.auth.decorators import login\_required

@login\_required  
def profile**(**request**)**:  
 return render**(**request, **'users/profile.html')**

# This doesn’t work for our classed based views, we are going to cover that later

**After doing this we are still getting an error if we try it out. It’s telling us that the page doesn’t exist.**



**Because Django looks for the route in a default location . So we need to tell Django where to find our login route and it’s as simple as going to the settings.py module, scroll to the end and add.**

LOGIN\_URL = **'login'**

**We need to add this because we changed where our default login.html is.**

### **“?next=” in url**

When we got to website/profile.html without being logged in we are going to notice an added parameter in our URL.



**This is a Django feature. Basically this is keeping track of the page that we were trying to access and it will direct us to the page after the login. So after we log in we will be redirected to the profile page since that is the page we were trying to access.**

# Creating a User Profile and Picture

In this part we are going to create the part of our website that handles user profiles and pictures.

To start off we are going to head into users/models.py and we are going to extend the current User model that Django provides for us.

from django.db import models  
from django.contrib.auth.models import User  
  
class profile**(**models.Model**)**:  
 user = models.OneToOneField**(**User, on\_delete=models.CASCADE**)** image = models.ImageField**(**default=**'default.jpg'**, upload\_to=**'profile\_pics')** def \_\_str\_\_**(**self**)**:  
 return **f'{**self.user.username**} Profile.'**

1.After importing User we are going to make a new class that inherits from models.Models

2.We are going to establish a one to one relationship with our User model with models.OneToOneField() which means:

One user can have on profile

And

One profile is associated with one user

We also add on\_delete=models.CASCADE which deletes our profile picture if our user is deleted.

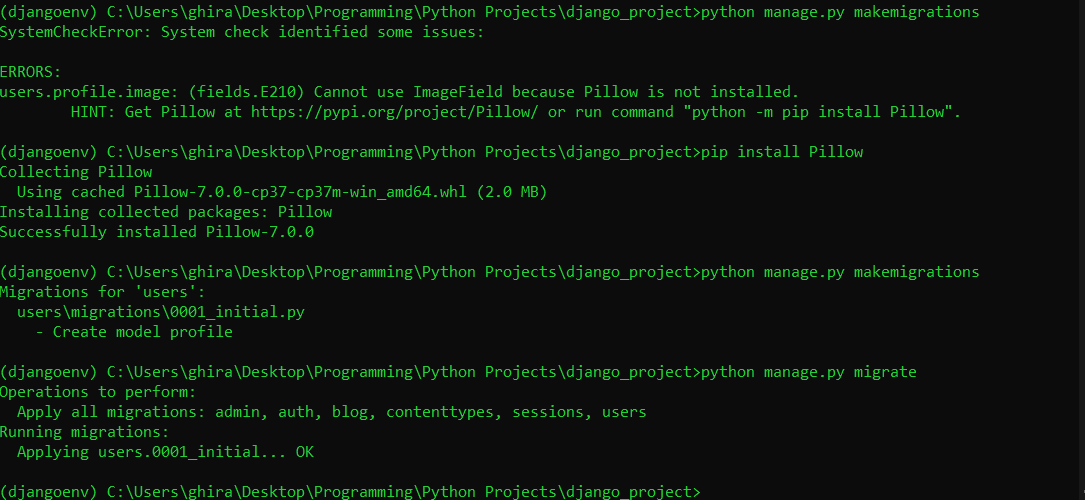
3.We are going to make a new variable called image that is equal to models.ImageField() . This is a Django build in model and the parameters we pass in are default=’default.jpeg’ which makes our default format a .jpg and upload\_to=’foldername’ (this basically creates a folder for our profile pictures)

4.We add a \_\_str\_\_ dunder method so when run the print function on the profile it will return as the username.

**Now all we have left is to makemigrations in our commandline followed by python manage.py migrate.**

This will throw us an exception because we need another library called Pillow which handles our image manipulation. To do this we just have to type:

Pip install Pillow 🡪 after this the makemigrations and migrate command should work.



## Don’t forget to register this in our admins.py file

We are going to register our new Profile class in the admin.py file so we can access it trough the admin page.

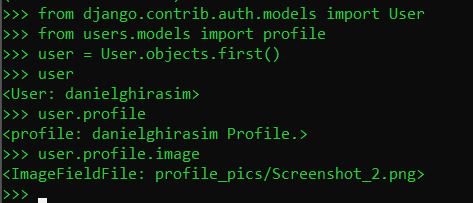
And we went over this before but quick recap:

From .models import Profile

Admin.site.register(profile)

## Adding user profile pictures from admin page

Now we are going to navigate to our Django admin page select profile and add a picture to a profile. This isn’t gonna show yet because we don’t have the functionality implemented, but if you add a profile picture and go to python manage.py shell and import profile model then you can check that Django already uploaded the image to our website.



We can also access more attributes like:

Image.width

Image.height

Image.url

We are going to use the image.url in order to display the images in our website, because we are going to pass in the url location in our html file.

## Changing the location of the saved images

We want to change the location of where our images are saved in order to reduce clutter in our main Django folder.

In order to do this we are going to navigate to our settings.py folder and add a couple lines of code at the bottom.

MEDIA\_ROOT = os.path.join**(**BASE\_DIR, **'media')**MEDIA\_URL = **'/media/**

This is basically telling that our MEDIA\_ROOT is going to be /djangowebsite/media and MEDIA\_URL is the public URL of that directory.

Further explained when we upload a profile picture then it is going to place a profile\_pics folder inside the media folder within our root. This applies for future profiles.

## Displaying our profile.html properly

Next we are going to add some HTML code to our profile page in order to look nicer . This will include an image, email, username etc.

We are going to edit our profile.html with the following HTML code:

{% extends 'blog/base.html' %}  
{% block content %}  
**<**div class="content-section"**>  
 <**div class="media"**>  
 <**img class="rounded-circle account-img" src="userimage.jpg"**>  
 <**div class="media-body"**>  
 <**h2 class="account-heading"**>**Username**</**h2**>  
 <**p class="text-secondary"**>**username@email.com**</**p**>  
 </**div**>  
 </**div**>  
</**div**>**{% endblock content %}

We are going to modify this file further in order to make the images display.

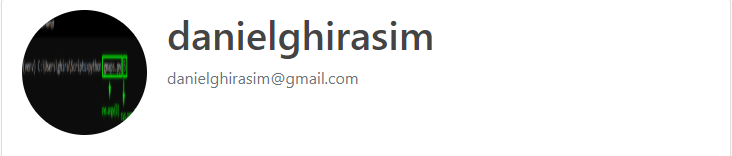
{% extends 'blog/base.html' %}  
{% block content %}  
**<**div class="content-section"**>  
 <**div class="media"**>  
 <**img class="rounded-circle account-img" src="{{ user.profile.image.url }}"**>  
 <**div class="media-body"**>  
 <**h2 class="account-heading"**>**{{user.username}}**</**h2**>  
 <**p class="text-secondary"**>**{{user.email}}**</**p**>  
 </**div**>  
 </**div**>  
</**div**>**{% endblock content %}

We still have to do some more editing in our urls.py to make Django see our images.

Go to mainproject/urls.py and add the following

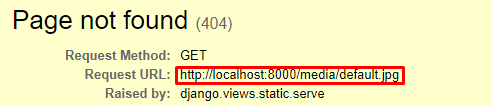
from django.contrib import admin  
from django.urls import path, include  
from users import views as user\_views  
from django.contrib.auth import views as auth\_views  
from django.conf import settings  
from django.conf.urls.static import static  
  
urlpatterns = **[** path**('admin/'**, admin.site.urls**)**,  
 path**('register/'**, user\_views.register, name=**'register')**,  
 path**('login/'**, auth\_views.LoginView.as\_view**(**template\_name=**'users/login.html')**, name=**'login')**,  
 path**('logout/'**, auth\_views.LogoutView.as\_view**(**template\_name=**'users/logout.html')**, name=**'logout')**,  
 path**('profile/'**, user\_views.profile, name=**'profile')**,  
 path**(''**, include**('blog.urls'))**,  
**]  
  
if settings.DEBUG:  
 urlpatterns += static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)**

Outcome:



## Uploading a default image for profiles with no images.

We should log in to a user with no profile image, and right click on the profile picture and check where it leads us so we can find out the location of our default.jpg.



We are going to place our default image to that location.

## Django signals - Every new user gets a profile

Currently we have to do this from the admin page. In order to do this automatically we are going to make something called a Django signal.

This is a bit more complicated.

We are going to crate a file inside our users app called signals.py

from django.db.models.signals import post\_save # This is a signal that gets fired after an object is saved  
from django.contrib.auth.models import User *# The sender – sending signal*from django.dispatch import receiver *# The receiver – receiving signal (get’s the signal and then performs some task)*from .models import profile  
  
@receiver**(**post\_save, sender=User**)**def create\_profile**(**sender, instance, created, \*\*kwargs**)**:  
 if created:  
 profile.objects.create**(**user=instance**)**@receiver**(**post\_save, sender=User**)**def save\_profile**(**sender, instance, \*\*kwargs**)**:  
 instance.profile.save**()**

Explained (Corey Schafer Django Part 8 27:30):

Post\_save 🡪 An object is created in our database

User 🡪 our default user model

Receiver 🡪 This receives the signal (a decorator)

Profile 🡪 Is the class from which we want to create an instance

If a New User is instantiated from the User class and then send the signal post\_save and this signal is going to be received by the receiver , the RECEIVER is our create\_profile function which takes all of those arguments from which instance is the new instance of our User and created Boolean. So if created true then create a new Profile Object (instance) for the new user instance.

## SAVE PROFILE – DJANGO SIGNAL

@receiver**(**post\_save, sender=User**)**def save\_profile**(**sender, instance, \*\*kwargs**)**:  
 instance.profile.save**()**

## SIGNALS – IMPORT READY FUNCTION

Users/apps.py

Users/apps.py

class UsersConfig**(**AppConfig**)**:  
 name = **'users'** def ready**(**self**)**:  
 import users.signals

We are going to create a ready method and we are going to import our users.signals in order for it to work

# Updating User Profile

In this section we are going to make some changes to our user profile so we can change our user data.

## Making the Forms

Head into forms.py

We are going to create something called a model form. This allows us to create a form that works with a specific database model. In this case we want a form that will update our user model.

from django.contrib.auth.forms import UserCreationForm  
from .models import profile

class UserUpdateForm(forms.ModelForm):  
 email = forms.EmailField()  
  
 class Meta:  
 model = User  
 fields = ['username', 'email']

First we import the models and forms we want to work with : User, Profile and UserCreationFrom.

We made an UserUpdateForm that inherits from form.ModelForm. We are going to pass in email and to in class Meta we are going to specify the model, in our case User and the fields we want to edit.

For the profile picture we are going to make another class because the profile picture is in our profile model not in our User model

class ProfileUpdateForm(forms.ModelForm):  
 class Meta:  
 model = profile  
 fields = ['image']

Here we don’t pass any anything into ProfileUpdateForm, I don’t know exactly why as of now. Next we are going to specify the model again we want to work with which is model = profile and in fields we are going to pass a single item list [‘image’].

## Updating our Views.py with the new forms.

We are going to modify our profile view with the following.

@login\_required  
def profile(request):  
 if request.method == 'POST':  
 u\_form = UserUpdateForm(request.POST, instance=request.user)  
 p\_form = ProfileUpdateForm(request.POST,  
 request.FILES,  
 instance=request.user.profile)  
 if u\_form.is\_valid() and p\_form.is\_valid():  
 u\_form.save()  
 p\_form.save()  
 messages.success(request, 'Successfully updated profile.')  
 return redirect('profile')  
 else:  
 u\_form = UserUpdateForm(instance=request.user)  
 p\_form = ProfileUpdateForm(instance=request.user.profile)  
  
  
 context = {  
 'u\_form': u\_form,  
 'p\_form': p\_form,  
 }  
  
 return render(request, 'users/profile.html', context)

Explained:

First we are going to import the forms from forms.py

Next, we are going to make a context dictionary with our forms like we did back in our post view. In this context we are going to pass it to our render function.

In order to display our current username and email in the empty fields we are going to add (instance=request.user) and (instance=request.user.profile) kwargs to the forms

After that we are going to check if the method is ‘POST’ if it is then we add (request.POST, instance=request.user) to user\_form

For our profile form we are going to add another request called request.FILES in order to properly upload the images.

After that we are going to check if both forms are valid and if they are we are going to save them by u\_form.save(), we are also going to send a django success message followed by a redirect back to our profile page

Why do we need to redirect back to our profile page? Because if stay on our page without redirect our request will still remain .POST and the browser will throw that annoying ‘Form submission’ warning. So in order to avoid this we just redirect back to our profile page which that starts a request.GET() and nothing will appear.

## Modifying our profile.html page

Here we are going to modify our profile.html page in order to properly display our forms. Crispy Forms will come in handy here. Essentially we are going to take a part of our register.html form and redo it to fit our profile page.

Profile.html

! Don’t forget to add {% load crispy\_forms\_tags %}

{% extends 'blog/base.html' %}  
{% load crispy\_forms\_tags %}  
{% block content %}  
<div class="content-section">  
 <div class="media">  
 <img class="rounded-circle account-img" src="{{ user.profile.image.url }}">  
 <div class="media-body">  
 <h2 class="account-heading">{{user.username}}</h2>  
 <p class="text-secondary">{{user.email}}</p>  
 </div>  
 </div>  
 <form method="POST" enctype="multipart/form-data">  
 {% csrf\_token %}  
 <fieldset class="form-group">  
 <legend class="border-bottom mb-2">Update Profile</legend>  
 {{ u\_form |crispy }}  
 {{ p\_form |crispy }}  
 </fieldset>  
 <div class="form-group">  
 <button class="btn btn-outline-info" type="submit" >Update</button>  
 </div>  
 </form>  
</div>  
{% endblock content %}

## Resizing large images to improve speed and space

We are going to navigate our models.py and do some changes on our profile model.

Basically we are going to override our Profile models default save method.

First from PIL (pillow) import Image

from PIL import Image  
  
class profile(models.Model):  
 user = models.OneToOneField(User, on\_delete=models.CASCADE) # CASCASDE - if user deleted the profile  
 image = models.ImageField(default='default.jpg', upload\_to='profile\_pics')  
  
 def \_\_str\_\_(self):  
 return f'{self.user.username} Profile.'  
  
 def save(self):  
 super().save()  
 img = Image.open(self.image.path)  
  
 if img.height > 300 or img.width > 300:  
 output\_size = (300, 300)  
 img.thumbnail(output\_size)  
 img.save(self.image.path)

super().save() – This runs the save method of our parent class and after that we are going to change the saved image size to something smaller and save it again.

## Adding images to our posts

A nice thing to do to make our website look nicer is to add the users profile picture next to each post. Doing this is fairly simple

We are going to navigate to our home.html file since that’s the place where our posts are located:

{% extends 'blog/base.html' %}  
{% block content %}  
 {% for post in posts%}  
 <article class="media content-section">  
 <img class ='rounded-circle article-img' src="{{ post.author.profile.image.url }}">  
 <div class="media-body">  
 <div class="article-metadata">  
 <a class="mr-2" href="#">{{ post.author }}</a>  
 <small class="text-muted">{{ post.date}}</small>  
 </div>  
 <h2><a class="article-title" href="#">{{ post.title }}</a></h2>  
 <p class="article-content">{{ post.content }}</p>  
 </div>  
 </article>  
 {% endfor %}  
{% endblock content %}

# Creating, Updating and Deleting Post - ClassBasedViews

In this part of the documentation we are going to look at how to create, delete, update, view posts with Django built in ClassBasedViews.

We are going to head into our blog/views.py module and we are going to import

from django.views.generic import ListView, DetailView, CreateView, UpdateView, DeleteView

These are all generic class based views and are going to do most of the functionality for us with very little code.

So far we’ve been using function based views. Our URL patterns are directed towards a certain view which are these functions, then the views handle the logic for the routes and render our templates.

ClassedBasedViews – have a lot of built-in functionality that will try to handle the backend stuff for us.

First we are going to replace our def home function view with django’s built-in ListView from Django.views.generic since our homepage is displaying a list of posts, also we are going to use the DetailView CBV in order to display in detail each of our post. (We are going to make a template for each one of these views)

## DEFAULT DJANGO NAMING CONVETION

Django default naming convention for templates are:

# <app>/<model>\_viewtype.html

Ex: blog/post\_list.html

## ListView

We are using list views for a list type view like in our blog, we want our homepage to be a list of posts. After applying this we should see no change in our layout.

Blog/views.py

class PostListView(ListView):  
 model = Post  
 context\_object\_name = 'posts'  
 template\_name = 'blog/home.html'  
 ordering = ['-date\_posted']

blog/urls.py

path('', PostListView.as\_view(), name='blog-home'),

**In this example we are going to somewhat customize the default ClassBasedView because we don’t want to use djangos default template name and context\_object\_name. By default class based views look for templates with a certain naming pattern.**

We made a new class called **PostListView** that inherits from **ListView**, we are going to pass in the model which it refers to , the **context\_object\_name** is posts (we have to set this in order for Django to know what the variable name is in our template , we set it to posts because it was like this before and know it will just loop over it) and let’s not forget about ordering, we can set the ordering by typing **ordering = [‘date\_posted’]** from latest to newest and [‘-date\_posted’] for newest to latest.

When we are done with then our homepage should be the same as before.

**We haven’t made a new template here because we already had one**

## DetailView

We are going to use DetailView for the detailed version of each post.

class PostDetailView(DetailView):  
 model = Post

This time we are going to use the Django default naming convention and also going to apply something we haven’t used before.

We want each post to look like this:

mywebsite.com/post/1 – should be post 1

mywebsite.com/post/2 – should be post 2

…..

Basically we want the ID or PrimaryKey of our post to be part of the route. So our route should look like this:

path('post/<int:pk>/', PostDetailView.as\_view(), name='post-detail'),

int means we only expect primary keys to be integers(we leave it as PK because that’s what the default DetailView expects, we can change it with an attribute in our newly create class if we want) and the Django naming convetion is .

**Here we going to use the Django default naming convetion**

**<app>/<model>\_<viewtype>.html**

**So in our case it is going to be**

**Blog/post\_detail.html – By setting this, Django will automatically find it without having to specify a template name in the subclass**

**Also our context\_object\_name by default is object so in our html template we are going to use object.author, object.date\_posted etc**

Post\_detail.html – (slightly modified version of post\_detail.html)

{% extends 'blog/base.html' %}  
{% block content %}  
 <article class="media content-section">  
 <img class ='rounded-circle article-img' src="{{ object.author.profile.image.url }}">  
 <div class="media-body">  
 <div class="article-metadata">  
 <a class="mr-2" href="#">{{ object.author }}</a>  
 <small class="text-muted">{{ object.date\_posted}}</small>  
 {% if object.author == user %}  
 <div>  
 <a class="btn-secondary btn-sm mt-1 mb-1" href="{% url 'post-update' object.id %}">Update</a>  
 <a class="btn-danger btn-sm mt-1 mb-1" href="{% url 'post-delete' object.id %}">Delete</a>  
 </div>  
 {% endif %}  
 </div>  
 <h2 class="article-title">{{ object.title }}</h2>  
 <p class="article-content">{{ object.content }}</p>  
 </div>  
 </article>  
{% endblock content %}

Now we have pages for individual pages for our posts.

## Adding links for individual posts

Next we are going to add links for our individual posts on the homepage.

<div class="media-body">  
 <div class="article-metadata">  
 <a class="mr-2" href="#">{{ post.author }}</a>  
 <small class="text-muted">{{ post.date\_posted}}</small>  
 </div>  
 <h2><a class="article-title" **href="{% url 'post-detail' post.id%}">{{** post.title }}</a></h2>  
 <p class="article-content">{{ post.content }}</p>  
</div>

Post.id makes sure that that gets passed in into the primary key in the url. This is the information from post/<int:pk>/

path('post/<int:pk>/', PostDetailView.as\_view(), name='post-detail')

## CreateView

This is going to be a view with a form where we create a new post, the only other thing that we provide is the field that we want to be in the form.

Views.py

class PostCreateView(LoginRequiredMixin, CreateView):  
 model = Post  
 fields = ['title', 'content']  
  
 def form\_valid(self, form):  
 form.instance.author = self.request.user  
 return super().form\_valid(form)

urls.py

from django.urls import path  
from . import views # '.' means from current folder  
from .views import PostListView, PostDetailView, PostCreateView, PostUpdateView, PostDeleteView  
  
urlpatterns = [  
 path('', PostListView.as\_view(), name='blog-home'),  
 path('post/<int:pk>/', PostDetailView.as\_view(), name='post-detail'),  
 path('post/<int:pk>/update/', PostUpdateView.as\_view(), name='post-update'),  
 path('post/<int:pk>/delete/', PostDeleteView.as\_view(), name='post-delete'),  
 path('post/new/', PostCreateView.as\_view(), name='post-create'),  
 path('about/', views.about, name='blog-about'),  
 path('exercise/', views.exercise, name='blog-exercise')  
]

Our form template name is going to be blog/post\_create.py by naming convetion. This is going to be really similar to other templates with forms that we used before in the documentation like register.html, so we are going to grab that and slightly modify it.

Post\_forms.html

{% extends 'blog/base.html' %}  
{% load crispy\_forms\_tags %}  
{% block content %}  
 <div class="content-section">  
 <form method="POST">  
 {% csrf\_token %}  
 <fieldset class="form-group">  
 <legend class="border-bottom mb-4">Create New Post</legend>  
 {{ form|crispy }}  
 </fieldset>  
 <div class="form-group">  
 <button class="btn btn-outline-info" type="submit" >Submit</button>  
 </div>  
 </form>  
 </div>  
{% endblock content %}

If we try to create a new post right now it won’t work. Because we are trying to create a post but our author is not defined yet.

We want the author of the post to be the current logged in user. In order to do that we are going to override our form\_valid method from our Post model

def form\_valid(self, form):  
 form.instance.author = self.request.user  
 return super().form\_valid(form)

We set the author with:

Form.instance.author = self.request.user

What this says basically “hey that form that you are trying to submit , before you do that take that author and set it == to our logged in user” and once we do that we can validate our form with : super().form\_valid(form).

After this we will still get an error because we don’t have a redirect url, we have to set this within our Post model:

from django.urls import reverse

class Post(models.Model):  
 """Each attribute is a field in the database"""  
 title = models.CharField(max\_length=100)  
 content = models.TextField()  
 date\_posted = models.DateTimeField(default=timezone.now)  
 author = models.ForeignKey(User, on\_delete=models.CASCADE)  
  
 def \_\_str\_\_(self):  
 return self.title  
  
 def get\_absolute\_url(self):  
 return reverse('post-detail', kwargs={'pk': self.pk}) # This redirects us to the detail page

**Here we want to use the reverse function because we want our defaultview to handle the redirect in order to do this we want to give it back a string version of our url for this we are going to use the reverse function.**

**This get absolute method basically tells Django how to find the url of each specific instance of a post AND WE RETURN THE PATH TO A SPECIFIC POST WITH REVERSE (RETURNS FULL PATH AS A STRING) .**

**To do it it needs the view name post-detail and the primary key so it can translate it to:**

**Mywebapp.com/post/PK == Mywebapp.com/post/2**

**We can also create a success\_url in the view and set it to go there if we want**

## Django Mixins

Django mixins are like @login\_required decorators for our normal view functions but for ClassedBasedViews we use mixins

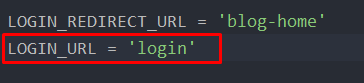
from django.contrib.auth.mixins import LoginRequiredMixin, UserPassesTestMixin

class PostCreateView(LoginRequiredMixin, CreateView):  
 model = Post  
 fields = ['title', 'content']  
  
 def form\_valid(self, form):  
 form.instance.author = self.request.user  
 return super().form\_valid(form)

urls.py

path('post/new/', PostCreateView.as\_view(), name='post-create')

We can see the we use here the LoginRequiredMixin, what this does is when we navigate to WEBSITE.COM/CREATE it will navigate us to the login page. How does it know to navigate us there? We set the LOGIN\_URL variable in our settings file.



UserPassesTestMixin

## Post Update View

class PostUpdateView(UserPassesTestMixin, LoginRequiredMixin, UpdateView):  
 model = Post  
 fields = ['title', 'content']  
  
 def form\_valid(self, form):  
 form.instance.author = self.request.user  
 return super().form\_valid(form)  
  
 def test\_func(self):  
 post = self.get\_object()  
 if self.request.user == post.author:  
 return True  
 return False

path('post/<int:pk>/update/', PostUpdateView.as\_view(), name='post-update'),

For our template Django is going to use the same template as create view ,

TO be continued

Corey Schafer video 34:57

# Pagination

Here we are going to add pagination to our website

First we upload some dummy posts in order to properly set up our pagination

## Paginator explained – Shell commands

Paginator is a Django feature that let’s us paginate our website easily.

Demonstration in shell

>>> from django.core.paginator import Paginator

>>> posts = ['1','2','3','4','5','6','7','8','9','10']

>>>p = Paginator(posts , 2) # paginates our posts, with 2 posts on a page

>>> p.num\_pages # Number of total pages

5

>>> for page in p.page\_range: # p.page\_range is an attribute (range of pages)

... print(page)

...

1

2

3

4

5

>>>

>>> p1 = p.page(1)

>>> p1

<Page 1 of 5>

>>> p1.object\_list # Returns the objects on that list

['1', '2']

>>> p2 = p.page(2)

>>> p2.has\_previous

<bound method Page.has\_previous of <Page 2 of 5>>

>>> p2.has\_previous() # Checks for previous page

True

>>> p2.has\_next() # Checks for next page

True

## Adding pagination to our home template

First we have to go into our views.py and add :

class PostListView(ListView):  
 model = Post  
 context\_object\_name = 'posts'  
 template\_name = 'blog/home.html'  
 ordering = ['-date\_posted']  
 paginate\_by = 5 # Number of posts on each page

Next we navigate to our home.html and add our pagination logic. (Just below our for loop of the posts)

{% if is\_paginated %}  
  
 {% if page\_obj.has\_previous %}  
 <a class='btn btn-outline-info mb-4' href="?page=1">First</a>  
 <a class='btn btn-outline-info mb-4' href="?page={{ page\_obj.previous\_page\_number }}">Previous</a>  
 {% endif %}  
  
 {% for num in page\_obj.paginator.page\_range %}  
 {% if page\_obj.number == num %}  
 <a class='btn btn-info mb-4' href="?page={{ num }}">{{ num }}</a>  
 {% elif num > page\_obj.number|add:'-3' and num < page\_obj.number|add:'3' %}  
 <a class='btn btn-outline-info mb-4' href="?page={{ num }}">{{ num }}</a>  
 {% endif %}  
 {% endfor %}  
 {% if page\_obj.has\_next %}  
 <a class='btn btn-outline-info mb-4' href="?page={{ page\_obj.next\_page\_number }}">Next</a>  
 <a class='btn btn-outline-info mb-4' href="?page={{ page\_obj.paginator.num\_pages }}">Last</a>  
 {% endif %}  
  
{% endif %}

It should look something like this:



## Displaying each post by user

What we want to do here: When we click on each username we want Django to send us to a different page that contains all the posts by that specific user. To do this we are going to create another class based view similar to our PostListView.

Blog/views.py

from django.contrib.auth.models import User

from django.shortcuts import render, get\_object\_or\_404

class UserPostListView(ListView):  
 model = Post  
 context\_object\_name = 'posts'  
 template\_name = 'blog/user\_posts.html'  
 paginate\_by = 5  
  
 def get\_queryset(self):  
 user = get\_object\_or\_404(User, username=self.kwargs.get('username'))  
 return Post.objects.filter(author=user).order\_by('-date\_posted')

The get\_object\_or\_404 is a method that when we search for a post from a user that doesn’t exists, it throws us a 404.

And the get\_queryset(self): is a method that we override that links individual user posts to each user. So the user variable gives us a list of each post of a specific user if they exist else it throws a 404 and when we access myblog.com/user/username it makes a querry to our database filtering all the posts from that user.

Now we modify our blog/urls.py

Don’t forget to import the new UserPostsViewList from views.py and now we add:

urlpatterns = [  
 path('', PostListView.as\_view(), name='blog-home'),  
 path('user/<str:username>', UserPostListView.as\_view(), name='user-posts'),

We added <str:username> to our path since we know username is a string.

## User\_posts template.

Here we are going to copy our home.html and make some modifications. We use home.html because we want to keep the pagination.

We use view.kwargs.username because the username is in the url and we can get that username from our view function (the username is a kwarg in the view function).

And page\_obj.paginator.count returns the sum of each post in each page.

{% extends 'blog/base.html' %}  
{% block content %}  
<h1 class="mb-3">Posts by: {{ view.kwargs.username }} ({{ page\_obj.paginator.count }})</h1>  
 {% for post in posts%}  
 <article class="media content-section">  
 <img class ='rounded-circle article-img' src="{{ post.author.profile.image.url }}">  
 <div class="media-body">  
 <div class="article-metadata">  
 <a class="mr-2" href="{% url 'user-posts' post.author.username %}">{{ post.author }}</a>  
 <small class="text-muted">{{ post.date\_posted}}</small>  
 </div>  
 <h2><a class="article-title" href="{% url 'post-detail' post.id%}">{{ post.title }}</a></h2>  
 <p class="article-content">{{ post.content }}</p>  
 </div>  
 </article>  
 {% endfor %}  
 {% if is\_paginated %}  
  
 {% if page\_obj.has\_previous %}  
 <a class='btn btn-outline-info mb-4' href="?page=1">First</a>  
 <a class='btn btn-outline-info mb-4' href="?page={{ page\_obj.previous\_page\_number }}">Previous</a>  
 {% endif %}  
  
 {% for num in page\_obj.paginator.page\_range %}  
 {% if page\_obj.number == num %}  
 <a class='btn btn-info mb-4' href="?page={{ num }}">{{ num }}</a>  
 {% elif num > page\_obj.number|add:'-3' and num < page\_obj.number|add:'3' %}  
 <a class='btn btn-outline-info mb-4' href="?page={{ num }}">{{ num }}</a>  
 {% endif %}  
 {% endfor %}  
 {% if page\_obj.has\_next %}  
 <a class='btn btn-outline-info mb-4' href="?page={{ page\_obj.next\_page\_number }}">Next</a>  
 <a class='btn btn-outline-info mb-4' href="?page={{ page\_obj.paginator.num\_pages }}">Last</a>  
 {% endif %}  
  
  
 {% endif %}  
{% endblock content %}

## Modifying other templates (when click on user take us to each users posts)

We want to head to home.html , post\_detail.html and modify our href to the author name to lead us to their posts.

In post\_detail.html, since we use the context name of objects we are going to add this to the href:

<a class="mr-2" href="{% url 'user-posts' object.author.username %}">{{ object.author }}</a>

In our home.html

<a class="mr-2" href="{% url 'user-posts' post.author.username %}">{{ post.author }}</a>

Here we use post since we are in a for loop that prints each post in posts (custom contenxt\_object\_name defined by us)

# Password Reset

In order to reset our password we are going to use 4 default classbasedviews from Django.

## PasswordResetView

This view sends us to a page within our website that displays a form with our email address.

The view looks likes this

Urls.py

path('password-reset/',  
 auth\_views.PasswordResetView.as\_view(template\_name='users/password\_reset.html'),  
 name='password\_reset'),

We will have to create a view for this, we are gonna use our register.html file as a starting point

{% extends 'blog/base.html' %}  
{% load crispy\_forms\_tags %}  
{% block content %}  
 <div class="content-section">  
 <form method="POST">  
 {% csrf\_token %}  
 <fieldset class="form-group">  
 <legend class="border-bottom mb-4">Reset your password</legend>  
 {{ form|crispy }}  
 </fieldset>  
 <div class="form-group">  
 <button class="btn btn-info" type="submit" >Confirm password reset</button>  
 <a class="btn btn-outline-info" href="?=blog-home" >Cancel</a>  
 </div>  
 </form>  
 </div>  
{% endblock content %}

## PasswordResetDoneView

This view is just a simple .html file that sends us a message that tells us that instructions were sent to the email we requested.

Urls.py

path('password-reset/done',  
 auth\_views.PasswordResetDoneView.as\_view(template\_name='users/password\_reset\_done.html'),  
 name='password\_reset\_done'),

Our template is straightforward

Password\_reset\_done.html

{% extends 'blog/base.html' %}  
{% block content %}  
 <div class="alert alert-info">  
 An email has been sent with instructions to reset your password.  
 </div>  
{% endblock content %}

## PasswordResetConfirmView

This is the confirm view which sends us an email and before it does this it needs two parameters uidb64 and a token in order for it to work.

This is because Django is trying to create an url with both of those values in it and wants to check if the user who made that request is the user who is trying to access that page. Basically it’s adding security to this route

Uidb64 – The users id encoded in base64

Token – The token to check if the password is valid

path('password-reset-confirm/<uidb64>/<token>/', auth\_views.PasswordResetConfirmView.as\_view(template\_name='users/password\_reset\_confirm.html'),  
 name='password\_reset\_confirm'),

### Adding E-mail functionality to django

Next step here is to setup our user and email we want to use with Django. In this example we are going to use a G-mail address but we can also set it up to use our localhost for it (<https://docs.djangoproject.com/en/2.1/topics/email/#configuring-email-for-development>).

In order to do this we have to go into our project/settings.py module and add some lines of code:

EMAIL\_USE\_TLS = True  
DEFAULT\_FROM\_EMAIL = 'email@gmail.com'  
SERVER\_EMAIL = 'email@gmail.com'  
EMAIL\_HOST = 'smtp.gmail.com'  
EMAIL\_PORT = 587  
EMAIL\_HOST\_USER = 'email@gmail.com'  
EMAIL\_HOST\_PASSWORD = 'password'  
EMAIL\_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'

It is recommended to use os environments for these variables.

Also don’t forget if you are using G-mail to enable less secure apps authentication within your google account.

For windows you can go into Advanced System Settings and setup environment variables that you can access with the OS library.

Ex:

Import Os

EMAIL\_HOST\_USER = os.environ.get(‘os user env name’)

EMAIL\_HOST\_PASSWORD = os.environ.get(‘os user env password’)

This way when you share your code with someone or upload it to github, nobody can snatch your email and password.

## PasswordResetDoneView

This is yet another simple view and template that displays after we successfully reset our password

Urls.py

path('password-reset-complete',  
 auth\_views.PasswordResetCompleteView.as\_view(template\_name='users/password\_reset\_complete.html'),  
 name='password\_reset\_complete'),

and our template.

Password\_reset\_done.html

{% extends 'blog/base.html' %}  
{% block content %}  
 <div class="alert alert-info">  
 An email has been sent with instructions to reset your password.  
 </div>  
{% endblock content %}