Django Web Development with Python Introduction

* Models - Django Tutorial
* Admin and Apps - Django Tutorial
* Views and Templates - Django Tutorial
* CSS - Django Tutorial
* User Registration - Django Tutorial
* Messages - Django Tutorial
* User Login and Logout - Django Tutorial
* Foreign Keys with Models - Django Tutorial
* Working with Foreign Keys - Django Tutorial
* Dynamic sidebar - Django Tutorial
* Deploying to a Server - Django Tutorial

pip install django

Make sure you're pointing to pip for python 3. Throughout this tutorial, I am going to mainly be using the syntax that you'll find useful on Linux. For example, to reference Python 3 on Linux, you would say python3. On Windows, you would instead do something like py -3.7, which would reference Python 3.7 specifically. Eventually, you are almost certain to launch your website on a Linux server. For this reason, I am going to use the Linux syntax, but it's highly common to develop locally first, and you can use any operating system for this.

You may find that, if you're viewing this tutorial long after I have covered it, things have changed with Django. You can either check the comments section of the videos, or grab the same version of Django that I am using, by doing:

pip install django==2.1.4

I would only do that for learning purposes, however. You should always try to use the latest version since it will include important changes, like security fixes.

I will be using sublime-text 3 as my editor, but you can use whatever editor you want. You can also use whatever operating system you want. I have developed with Django on Windows, Mac, and Linux. They all work just fine.

Alright, assuming you have Django installed, let's get started! With your installation of Django, you should now have a command line keyword: django-admin. You can use this to start a new project. Django considers all websites to be a collection of apps. Consider a website that has a forum, a shop, and a blog. Each of those three things would be considered its own "app." A collection of these apps is your project. So, let's start a project. In your terminal/cmd.exe, do:

django-admin startproject mysite

You can call the mysite bit whatever you want to call your project, but it seems to be a pretty consistent convention to call your project mysite, so I will stick with that.

The startproject command will create a new directory called whatever you called your project, In our case, that's called mysite.

our project's directory is called mysite and your primary app is also called mysite. The only real role of this "primary app" as I am going to call it is to link your other apps. You shouldn't really be doing much in there besides managing settings and urls mostly.

Django is meant to be highly modular. When you develop an app like some forums for one website, you should be able to easily take your forum app to another website seemlessly and implement it nearly immediately.

Okay, so with this in mind, let's add our first actual app to our project. To do this, we will use manage.py. This is a helper python script that will allow you to do things within your project. First, we'll use it to create another app. In your terminal/command line, navigate to the dir where this manage.py file is, then do:

python3 manage.py startapp main

You should see that a new directory has been created called main. So now the top levels of your project's structure are something like:

main

-main (directory, your "primary" app)

-mysite (directory, this is your new app)

-manage.py (helper python script)

Okay great. Let's go ahead and run our server now. We will do this with manage.py. You should probably open a separate terminal/command prompt to run the server within. We will keep the server running through much of the development.

To run the server, do:

python3 manage.py runserver

You should see something like:

python3 manage.py runserver

Performing system checks...

System check identified no issues (0 silenced).

You have 15 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.

Run 'python manage.py migrate' to apply them.

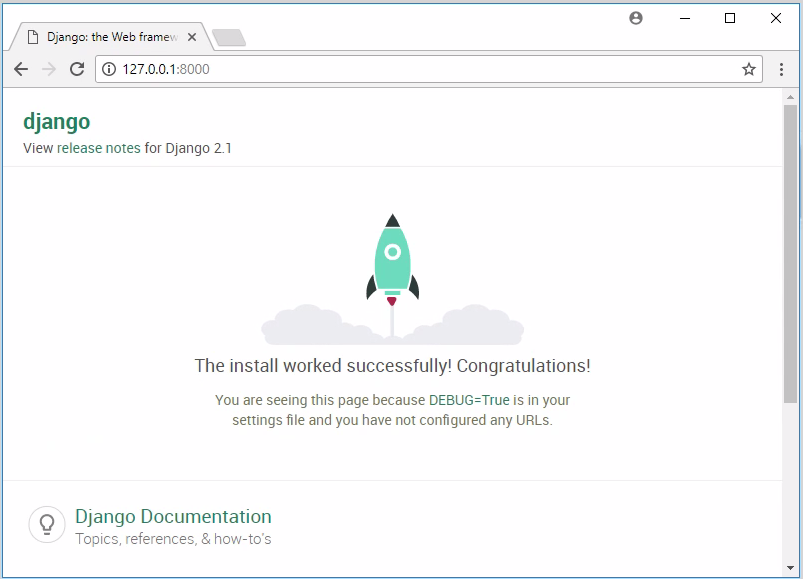
January 10, 2019 - 18:50:09

Django version 2.1.5, using settings 'mysite.settings'

Starting development server at http://127.0.0.1:8000/

Quit the server with CTRL-BREAK.

For now, we can ignore the migrations thing, we'll talk about that later on. We can see that our development server is now running at http://127.0.0.1:8000. Open a browser and head to that address. You should see something like:



This is just the default "working" message that you will see if you're not handling for the home page yourself yet. Let's replace this with something of our own! I will leave the server running in a separate terminal from now on.

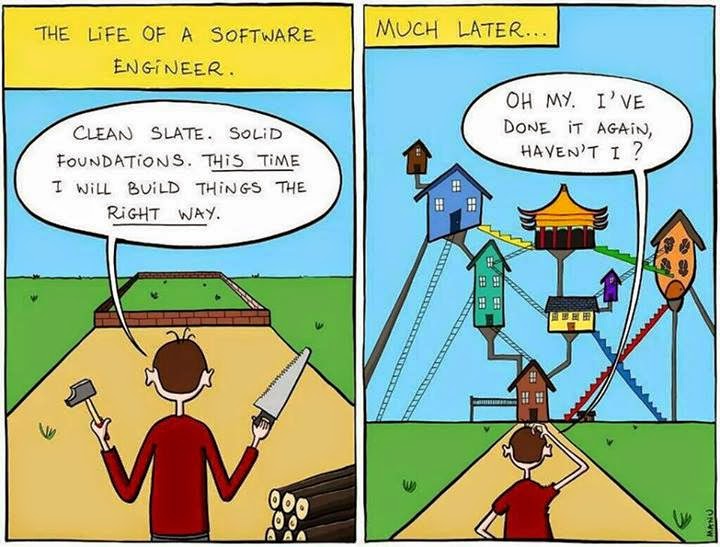
Django uses what's called a Model View Controller paradigm. Every page you visit on a Django application is likely using these three things to serve you data.

* Model: Your database abstraction, which will contain objects that are mapped to your database. For example, we'll have a "Tutorial" model/object, a "User model/object," a "Tutorial Series" model/object...etc. All you need to do is define these models and Django handles the rest for you. You can even change your models down the line and, through migrations, Django can help you get it done within seconds, rather than the likely hour...or more... it would take you to do this yourself.
* View: How you will represent the data. This is where you will actually render things to a user.
* Controller: How you map URLs to views.

While we call it an MVC (model, view, controller), you can imagine it moreso working in reverse. A user will visit a URL, your controller (urls.py) will point to a specific view (views.py). That view can then (it doesn't actually HAVE to) interface with your models.

We can simplify this slightly, however, and just have the controller point to your view, and the view can just return a string, just so we can get a feel for how things connect.

When first starting out with something like Django, all these connections and things we need to do in order to do seemingly simple things can seem daunting, and like it's not worth it. With other frameworks, displaying some simple text can be done in moments, and doesn't require all this fiddling about. Where Django shines is not with simple websites. It shines when you've got a website that has grown in time, and you keep wanting to add features...etc, but then you've got a huge mess of code. Django helps you to keep things organized, and forces you to immediately follow good practices which will scale over time. This is annoying at first, but worth it long term. I like to share the following comic:



We always start projects with the best of intentions, but, we tend to have poor foresight. The best thing Django can do for you is the very thing that you may find annoying initially: Abstraction. Over time if you wanted to add more attributes to your users? It would be super simple with Django. Say you wanted to add more functionality like a more advanced nav bar? How about search bar? How about a full redesign? Through Django, this would all be trivial.

Okay, so let's display some simple text at the homepage. To do this, we don't need a model, but we do need the view (which will dictate the text we want to display) and the controller to point to the view, based on the URL. Because I think of things in the order of the user, I would first start with the controller. So we need to tell Django that homepage should return some view. Django begins in your "primary" app when looking for URLs. So first, let's go to: mysite/mysite/urls.py:

from django.contrib import admin

from django.urls import path

urlpatterns = [

path('admin/', admin.site.urls),

]

So we've only got one URL here, and that's apparently to some administration page, which isn't our focus right now. So, what we need to do is point this URL to a view. That said, Django sees websites as a collection of apps. So actually the urls.py file inside your "primary" app is usually just going to point to your apps. How do we point to an app? We just point to that app's urls.py file! So let's just add that:

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path("", include('main.urls')),

path('admin/', admin.site.urls),

]

Aside from adding the extra path, don't forget to also import "include" from django.urls.

Okay, so now django knows that, when the path is basically empty (the homepage), to look inside of the main app at its urls.py file to see if that file (the controller) points to a view, which would be some function inside of that app's views.py.

We have yet to modify our main app's urls.py file, and we also have not made any view. Let's do both! Let's navigate into our mysite/main app. We can see that there are already various things here...but there's no urls.py! Let's add that now:

mysite/main/urls.py

from django.urls import path

from . import views

app\_name = 'main' # here for namespacing of urls.

urlpatterns = [

path("", views.homepage, name="homepage"),

]

We import path the same as before. We locally import our view. We specify the app name (this isn't required, but you might as well just get in the habit of doing it. Later it will become very useful when we want to dynamically reference URLs). Like I said, Django is highly modular, and even if you wanted to change URL paths, or use someone else's application, but maybe not the same URL paths, you can easily do it. This is also why we give a name as a parameter of the path. Okay, great! We've got the controller all set. When someone visits the homepage, Django looks first at the mysite/mysite/urls.py, seeing that it points to mysite/main/urls.py, which then points to views.homepage (so, a function called homepage inside of views.py). Do we have that? Nope. Let's do it. views.py does already exist here, however, so open that up to edit it. It should look like:

mysite/main/views.py

from django.shortcuts import render

# Create your views here.

Normally, these views will render some HTML template and pass some variables, but we're just going to make it super simple and return a straight HTTP response. To do this, we need to import it:

from django.http import HttpResponse

We told urls.py to look for a homepage function, so let's define that:

def homepage(request):

return HttpResponse("pythonprogramming.net homepage! Wow so #amaze.")

Note that we pass request. You will always pass this to your views.

Your full views.py should now be:

mysite/main/views.py

from django.shortcuts import render

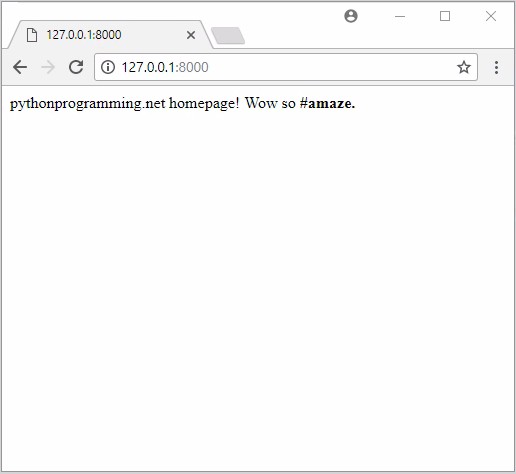
from django.http import HttpResponse

# Create your views here.

def homepage(request):

return HttpResponse("pythonprogramming.net homepage! Wow so #amaze.")

Now, go back to your browser and refresh. If you fiddled at all up to this point, you probably hit an error, and you will need to re-run the server since you will have stopped on an error most likely. Now, at the homepage, you should see:



We'll stop here, and what I suggest you do is make sure you can do all of the above, without the use of a tutorial. So, start a fresh project, add an app, configure the url.py files to point the homepage to a view, and have that view return a simple string. Things will only become more complicated from here, so it's essential you understand these basics up to this point. If you're fuzzy on anything, I strongly advise you to ask questions or Google it until you're solid on everything up to this point.

As we progress, I would suggest that you continue doing this. At the end of each tutorial, save the project. Then at the next tutorial, make a copy of the project, follow the tutorial, then try to do it to the copy without needing to consult the tutorial. You might forget import paths or function names, that's fine. What you want to make sure is that you actually understand what connections need to be made.

Alternatively, you could also work on a side-project that isn't the same as what we're doing here. As you follow along here, apply it to your own project. The most important thing is that you don't gloss over some concept that you don't understand. Join us on [Discord](https://discord.gg/sentdex) and ask for help in the #help channel if you can't figure something out.

## Models - Django Tutorial

Welcome to Part 2 of the web development with Python and Django tutorial series. In this tutorial, we're going to introduce the concept of Models, which is where the majority of the value of Django comes from.

The first model we will start with is the Tutorial model, since tutorials are the main aspect of PythonProgramming.net. So, what might some attributes of a tutorial be? Obviously we've got the tutorial content, like the writeup itself. Maybe also we'd want to have the tutorial's title specifically, and then maybe the date it was published. That should be enough to start with. Later, we might come up with more things for tutorials, such as maybe what series they are a part of, or the category it falls under, and so on. As mentioned before, with Django, you can easily add more fields to your database with very little struggle, so it's not really as important as it usually is to think ahead. So, let's build the Tutorial model. Each model will be a unique class within your app's models.py file.

Let's open up models.py from main:

mysite/main/models.py

from django.db import models

# Create your models here.

We'll start by defining our Tutorial model here:

from django.db import models

class Tutorial(models.Model):

tutorial\_title = models.CharField(max\_length=200)

tutorial\_content = models.TextField()

tutorial\_published = models.DateTimeField('date published')

def \_\_str\_\_(self):

return self.tutorial\_title

All models will inherit from models.Model. Then, we just define our fields with ...well...fields. Note that different fields are defined in different ways. We expect our title to be fairly short, so we define this as a CharField. These fields correspond to the format of our data in the actual database. You might wonder, what's the difference between a CharField, which we use for the title, and a TextField, which we use for the content itself. In general, we use the CharField for something that does have a limit to the size, and a TextField when we don't have a limit. For all of the fields, see the [django model fields documentation](https://docs.djangoproject.com/en/2.1/ref/models/fields/).

Finally, we override the the \_\_str\_\_ special method to make it a bit more readable when it's being displayed in string form, which we will see soon.

Okay, any time there has been a change in your models (either a new model, or you modified an existing model), you need to do a migration. There are two steps involved here. First, we run a makemigrations, then we run an actual migrate. We use the manage.py script for this, so let's get it done:

python3 manage.py makemigrations No changes detected

What?

So our migrations will only apply to apps that we've told Django we want "installed." This probably wont be the last time you add an app, build some models, attempt to migrate, and get this message. Hopefully, you wont be confused when it eventually happens to you on your own! So let's go into mysite/mysite/settings.py and add 'main.apps.MainConfig', to INSTALLED\_APPS so it should look like:

mysite/mysite/settings.py

INSTALLED\_APPS = [

'main.apps.MainConfig',

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

]

Does ...that exist? Let's check! Go open mysite/main/apps.py:

from django.apps import AppConfig

class MainConfig(AppConfig):

name = 'main'

Yep!

Okay, let's try to make our migrations again!

python3 manage.py makemigrations

Migrations for 'main':

main\migrations\0001\_initial.py

- Create model Tutorial

Looks good. What this step actually does is it just builds the code required for the migration, it doesn't actually apply them. If you're curious, you can see all of your migrations by going to the migrations directory of an app. For example, head to mysite/main/migrations. In there, you should see 0001\_initial.py, open it up:

0001\_initial.py

# Generated by Django 2.1.5 on 2019-01-11 01:35

from django.db import migrations, models

class Migration(migrations.Migration):

initial = True

dependencies = [

]

operations = [

migrations.CreateModel(

name='Tutorial',

fields=[

('id', models.AutoField(auto\_created=True, primary\_key=True, serialize=False, verbose\_name='ID')),

('tutorial\_title', models.CharField(max\_length=200)),

('tutorial\_content', models.TextField()),

('tutorial\_published', models.DateTimeField(verbose\_name='date published')),

],

),

]

If you know SQL, then you know that's not SQL! If you want to see the exact SQL that will be run, you can also do:

python3 manage.py sqlmigrate main 0001

BEGIN;

--

-- Create model Tutorial

--

CREATE TABLE "main\_tutorial" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "tutorial\_title" varchar(200) NOT NULL, "tutorial\_content" text NOT NULL, "tutorial\_published" datetime NOT NULL);

COMMIT;

... but you probably wont be doing that. In general, you will just simple make or modify your models. Run a makemigrations and then run a migrate and you'll be done.

Cool, okay let's actually migrate then!

python3 manage.py migrate

Operations to perform:

Apply all migrations: admin, auth, contenttypes, main, sessions

Running migrations:

Applying contenttypes.0001\_initial... OK

Applying auth.0001\_initial... OK

Applying admin.0001\_initial... OK

Applying admin.0002\_logentry\_remove\_auto\_add... OK

Applying admin.0003\_logentry\_add\_action\_flag\_choices... OK

Applying contenttypes.0002\_remove\_content\_type\_name... OK

Applying auth.0002\_alter\_permission\_name\_max\_length... OK

Applying auth.0003\_alter\_user\_email\_max\_length... OK

Applying auth.0004\_alter\_user\_username\_opts... OK

Applying auth.0005\_alter\_user\_last\_login\_null... OK

Applying auth.0006\_require\_contenttypes\_0002... OK

Applying auth.0007\_alter\_validators\_add\_error\_messages... OK

Applying auth.0008\_alter\_user\_username\_max\_length... OK

Applying auth.0009\_alter\_user\_last\_name\_max\_length... OK

Applying main.0001\_initial... OK

Applying sessions.0001\_initial... OK

Whoa, that's a lot of stuff! Remember those pending migrations from before that we saw when we ran the server? That was a bunch of them, but we can also see the one for the Tutorial model:

Applying main.0001\_initial... OK

We can see that we also did some stuff for admin and auth.

Okay, so what? We can't really see what's so special about any of this. Let's add a tutorial. One quick way for us to do this at the moment is through the shell. We can access the shell through, you might be able to guess...manage.py

python3 manage.py shell

Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

(InteractiveConsole)

This gives us a quick way to interact with our website without the need to write up some view and controller just to test something. My main use for this is to do something like test a query like a filter or a get. Once we have many related models, writing the code to properly do what we want doesn't necesaarily work the first time. Rather than printing out, logging, or displaying the data in a view, we can quickly interact via the shell. Let's see a quick example. Let's import our Tutorial model.

>>> from main.models import Tutorial

>>> Tutorial.objects.all()

<QuerySet []>

We can make a new Tutorial object from here quite easily. Recall our attributes were:

* tutorial\_title
* tutorial\_content
* tutorial\_published

Two of those are just strings, one is a date. Let's import timezone:

>>> from django.utils import timezone

Now we can make a new Tutorial object by doing:

>>> new\_tutorial = Tutorial(tutorial\_title="To be", tutorial\_content="or not to be.  
  
That is the question.", tutorial\_published=timezone.now())

Now all we need to do to commit this object to our database is .save()

>>> new\_tutorial.save()

Now we can do:

>>> Tutorial.objects.all()

<QuerySet []>

So we get this thing called a QuerySet, which we can iterate over like:

>>> for t in Tutorial.objects.all():

... print(t.tutorial\_title)

...

To be

The shell winds up being more useful down the line, however, when we have many more objects and things are far more complex. I wouldn't actually recommend using the shell to actually insert data. Can you imagine writing an entire tutorial via the shell?! Yikes! Instead, you're much more likely to use the admin page for this, which is what we're going to be talking about in the next tutorial!

## Admin and Apps - Django Tutorial

Welcome to part 3 of the web development with Django and Python tutorial series. In this tutorial, we're goign to be checking out the admin functionality that comes with Django.

In order to do this, we need to first create an administrator user. To do this:

python3 manage.py createsuperuser

Username (leave blank to use 'harrisonkinsley'): sentdex

Email address: harrison@pythonprogramming.net

Password:

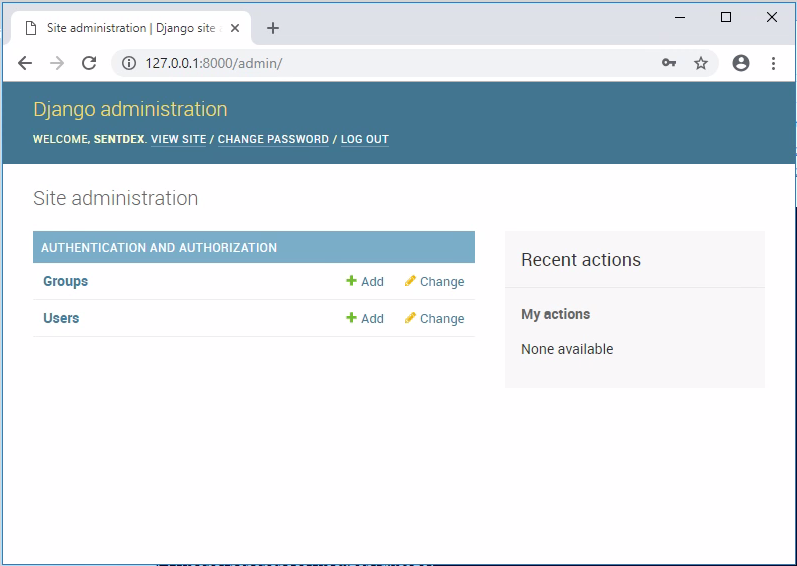
Password (again):

Superuser created successfully.

The email can just be left blank. The superuser is like other users, and Django user objects just happen to come with email fields. That said, django has built-in some reporting features that will email your superusers when exceptions are hit, which can be useful. More information on that: [Django email reports](https://docs.djangoproject.com/en/2.0/howto/error-reporting/#email-reports)

Now that we've done that, let's log in to the admin page: http://127.0.0.1:8000/admin

Log in with the user we just created, and you should see a page like:



Here is where you can interact with your models via an actual user interface. In here, you can see, modify, add, or remove entries for your models that are registered here. As you can see, it looks like the only models available to us are Groups and Users, however. That said, go ahead and click on users, then on the username you just created. As you can see, there are actually quite a few fields here. That's because this is Django's User model. Still, all these fields here, available to us to edit things, is pretty darn cool. We could even add new users right here in the admin. We're probably not going to be adding users ourselves, however. We will be adding tutorials, however, so how might we get our Tutorials model in this neat place? All we need to do for this is register them.

To do this, head to mysite/main/admin.py and add the following import:

from .models import Tutorial

Then do:

admin.site.register(Tutorial)

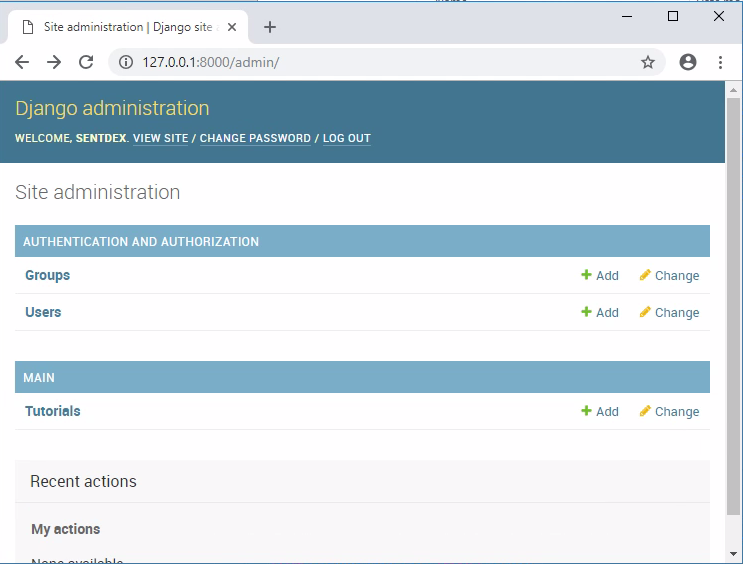
...and that's it for now. The full mysite/main/admin.py file is:

from django.contrib import admin

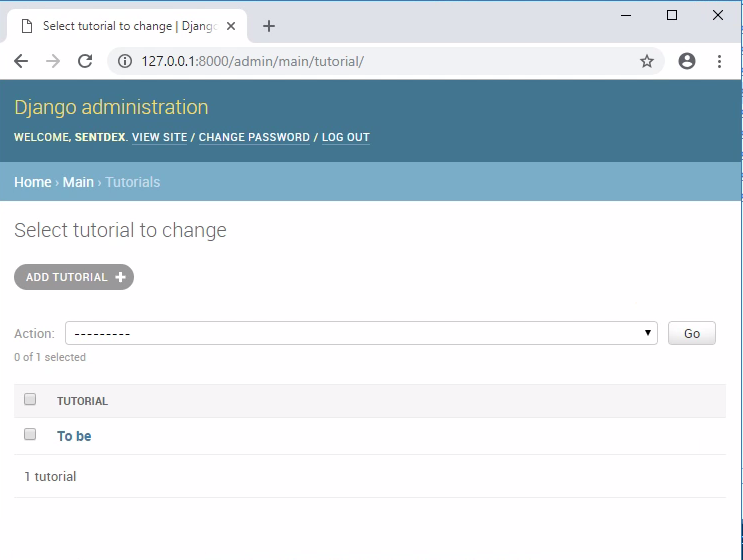
from .models import Tutorial

admin.site.register(Tutorial)

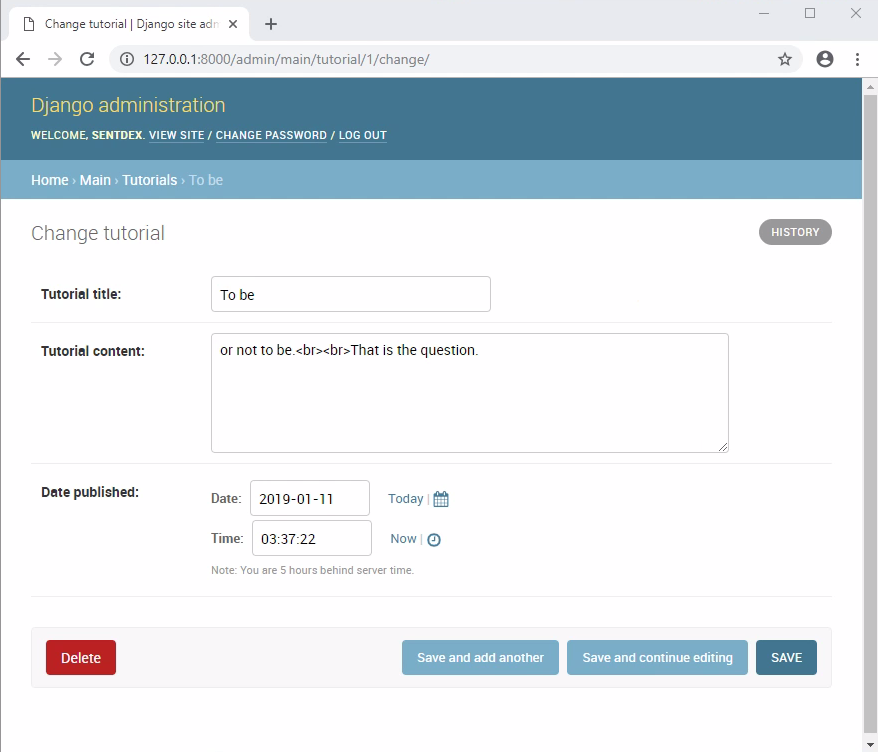
Now, refresh your website/admin page http://127.0.0.1:8000/admin/, and you should see:



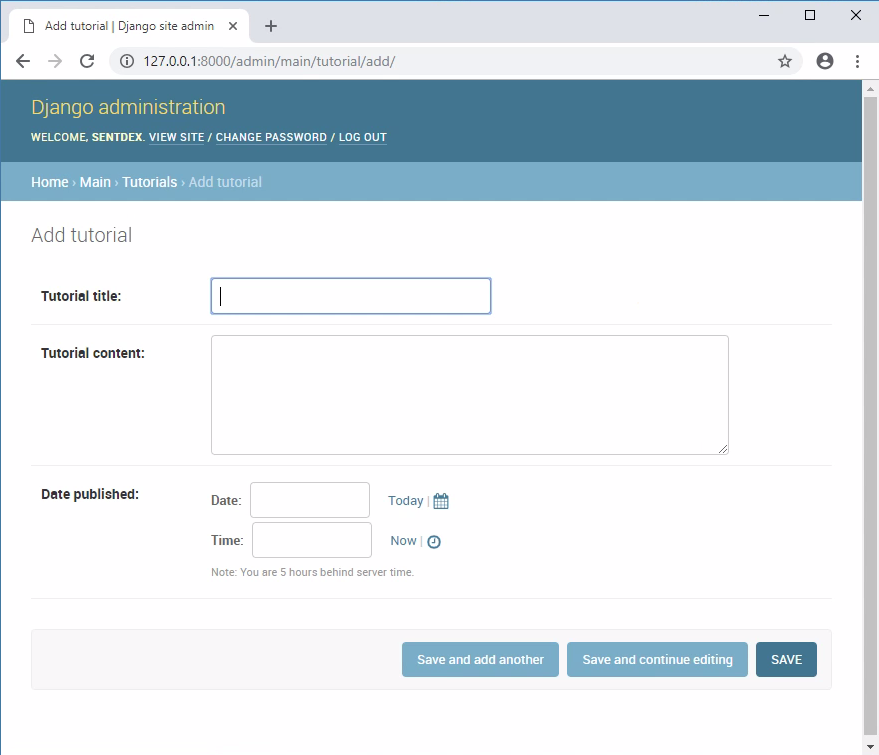
Now, click on Tutorials, and you should see the tutorial we created from the shell:



We can then click on that entry to edit it:



From that previous page, we can also choose to add a new tutorial, for example:



Notice how the fields are all kind of different. The title is a small, one "row" entry. Then the content is a larger box. The date entry boxes are the shortest, one row, and they even have calendar and clock options. To make things even cooler, the dates allow for us to just quickly click "today" and "now." This...is...AWESOME!

Django does all this through the field types we chose in our model, from there, this admin page can list out all of your models, allow you to edit them, add them, and delete them.

If you've ever needed to write an admin control panel like this, you're likely very giddy right now. This is a pretty awesome thing that you get from Django right out of the gate.

But wait, there's more! We can organize how this model is presented to us. It's not always the case that the order of the columns in the table is what we prefer. We also might not care about all of the columns. To modify how this is presented to us, let's go back to edit the admin.py file

mysite/main/admin.py

from django.contrib import admin

from .models import Tutorial

class TutorialAdmin(admin.ModelAdmin):

fields = ["tutorial\_title",

"tutorial\_published",

"tutorial\_content"]

admin.site.register(Tutorial,TutorialAdmin)

Now, we have the same information here, but the order it is presented to us has changed. We could also comment out the tutorial\_published part, like:

from django.contrib import admin

from .models import Tutorial

class TutorialAdmin(admin.ModelAdmin):

fields = ["tutorial\_title",

#"tutorial\_published",

"tutorial\_content"]

admin.site.register(Tutorial,TutorialAdmin)

And then refresh admin to see that column is now gone. So you can register whatever you need, in whatever order you like.

Another option we have is to modify how things are grouped together for organization purposes. In our case right now, we really don't have too many columns where things are confusing just yet, but eventually you might. So you can organize things for example by doing:

from django.contrib import admin

from .models import Tutorial

class TutorialAdmin(admin.ModelAdmin):

fieldsets = [

("Title/date", {'fields': ["tutorial\_title", "tutorial\_published"]}),

("Content", {"fields": ["tutorial\_content"]})

]

admin.site.register(Tutorial,TutorialAdmin)

One final thing I'd like to do is have the tutorial\_published default to now. There may be times when I want to change the published date from now, but, most often, the published time will be whatever the current time is.

mysite/main/admin.py

tutorial\_published = models.DateTimeField('date published', default=datetime.now)

Full script:

from django.db import models

from datetime import datetime

class Tutorial(models.Model):

tutorial\_title = models.CharField(max\_length=200)

tutorial\_content = models.TextField()

tutorial\_published = models.DateTimeField('date published', default=datetime.now)

def \_\_str\_\_(self):

return self.tutorial\_title

Now, for new tutorials, the date published field is auto-filled to be the current date.

But wait, there's more!

Remember what I was saying about Django being a collection of apps and part of what makes the abstraction that Django forces upon you awesome is that apps are highly modular? Let's see just how easy that can really be.

One thing my tutorials desperately could use is an editor, not really just some text field. I can write HTML in here, sure, but that would be rather tedious, especially if I made some typo and then I wouldn't see it until I push to publish it! Instead, I would like a WYSIWYG (what you see is what you get) editor. Luckily many of these exist within the Django ecosystem. The one I will make use of is a branch off of TinyMCE. To get it, we just need to do:

python3 -m pip install django-tinymce4-lite.

Now this is an app, so we need to add it to our INSTALLED\_APPS in the mysite/mysite/settings.py file:

INSTALLED\_APPS = (

...

'tinymce',

...

)

Then, somewhere in the settings.py also add:

TINYMCE\_DEFAULT\_CONFIG = {

'height': 360,

'width': 1120,

'cleanup\_on\_startup': True,

'custom\_undo\_redo\_levels': 20,

'selector': 'textarea',

'theme': 'modern',

'plugins': '''

textcolor save link image media preview codesample contextmenu

table code lists fullscreen insertdatetime nonbreaking

contextmenu directionality searchreplace wordcount visualblocks

visualchars code fullscreen autolink lists charmap print hr

anchor pagebreak

''',

'toolbar1': '''

fullscreen preview bold italic underline | fontselect,

fontsizeselect | forecolor backcolor | alignleft alignright |

aligncenter alignjustify | indent outdent | bullist numlist table |

| link image media | codesample |

''',

'toolbar2': '''

visualblocks visualchars |

charmap hr pagebreak nonbreaking anchor | code |

''',

'contextmenu': 'formats | link image',

'menubar': True,

'statusbar': True,

}

These are some configuration settings of what we might want to include. There are others you can add. See here for further configuration: [TinyMCE4-lite configurations](http://romanvm.github.io/django-tinymce4-lite/configuration.html" \t "blank)

Next, we need to add a pointer to the app so it can be referenced when we call it. To do this, let's now edit mysite/mysite/urls.py

urlpatterns = patterns('',

...

path('tinymce/', include('tinymce.urls')),

...

)

Finally, we just need to make use of TinyMCE where we want it. To do this, we need to override a form to use our TinyMCE widget. In this case, it's not just any form, however, we want to use it within the admin page. To do this, go back into our mysite/main/admin.py file, and add the follwowing imports:

from tinymce.widgets import TinyMCE

from django.db import models

The first is for our widget, the second is so we can override one of our models fields (the textfield). To do this, we'll add:

formfield\_overrides = {

models.TextField: {'widget': TinyMCE()},

}

So the full script becomes:

mysite/main/admin.py

from django.contrib import admin

from .models import Tutorial

class TutorialAdmin(admin.ModelAdmin):

fieldsets = [

("Title/date", {'fields': ["tutorial\_title", "tutorial\_published"]}),

("Content", {"fields": ["tutorial\_content"]})

]

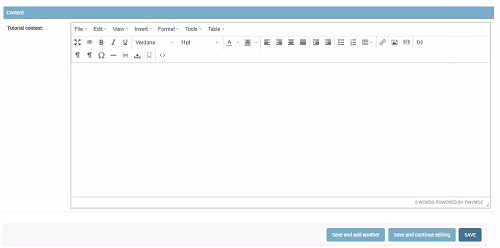
formfield\_overrides = {

models.TextField: {'widget': TinyMCE()},

}

admin.site.register(Tutorial,TutorialAdmin)

Now, refresh http://127.0.0.1:8000/admin/main/tutorial/add/



Awesome! So this editor allows us to more easily write HTML, insert code snippets...etc. We can also view the raw HTML and insert our own custom HTML as well if something we want doesn't exist in the editor.

While we're here, let's go ahead and add a quick tutorial with a code snippet just for kicks. Put whatever you want, give it a title, and save.

Let's now head back to our views to learn how our views can interact with our models, as well as how the Django templating works for displaying and working with python objects inside of HTML templates.

## Views and Templates - Django Tutorial

Welcome to part 4 of the web development with Django and Python tutorial series, where we will be talking more in depth about views, interacting with models, and templating.

To begin here, let's head into mysite/main/views.py and change our homepage function to:

def homepage(request):

return render(request = request,

template\_name='main/home.html',

context = {"tutorials":Tutorial.objects.all})

Here, we're using render, which will render an actual html file/template for us. It will also help us to pass Python objects to that template for us to work with. This will allow us to iterate over objects with loops, use if-statements...etc. To do this, we pass a dictionary where the key is the name of the variable which we will reference from within the template, and the value is the object it will represent. As you can see, we want to interact with our Tutorial model. To do this, we need to import it:

from .models import Tutorial

The full script is now:

mysite/main/views.py

from django.shortcuts import render

from django.http import HttpResponse

from .models import Tutorial

# Create your views here.

def homepage(request):

return render(request = request,

template\_name='main/home.html',

context = {"tutorials":Tutorial.objects.all})

Next, we have referenced a template that simply doesn't exist...so let's fix that next!

Django looks for templates in each of your app's directories by seeking out any directory called "templates." It then builds a list of templates, and will grab the one that matches (or throw an error if none do). The problem is, sometimes, templates will have the same name, because you forgot the names of all of the templates in some other app that maybe you wrote a while ago, or it's someone else's that you've brought in. Anyway, the way we handle for this is to actually have a directory inside of our templates directory called the same as our app name. So, let's make a new directory called templates at mysite/main/templates, then, inside of that, add another directory called main, so you will have mysite/main/templates/main/. Head there, and make a file inside called home.html. Now we can add some HTML here, along with our object. Django has its own templating, which is \*similar\* to Jinja, but some things are indeed different. To begin, let's just make this page iterate over all of our tutorials so far and display them.

The way we iterate in Django's templating is:

{% for tut in tutorials %}

Unlike Python, however, we also have to denote where the loop ends:

{% endfor %}

Generally, logic in templating is done with {% %} and then variables are encased in {{}}

Now, let's interact with our tut var. Each of these tut objects is a Tutorial object so it has our attributes for content, title, and published date. Let's check that out:

{% for tut in tutorials %}

<p>{{tut.tutorial\_title}}</p>

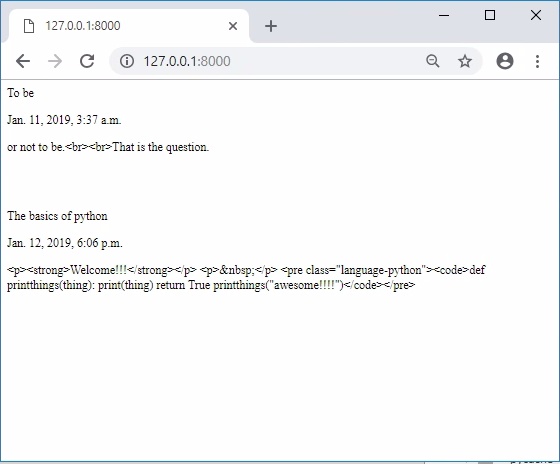
<p>{{tut.tutorial\_published}}</p>

<p>{{tut.tutorial\_content}}</p>

<br><br>

{% endfor %}

Okay, save everything and let's check it out by visiting our homepage: http://127.0.0.1:8000/



It's not the most beautiful thing I've ever seen, but still, it worked! ...sort of. We can see our HTML is being displayed as plain text. This is because Django escapes it by default. It's not quite ideal if our users can do something like...insert javascript into forms, for example. That said, these tutorials are coming from our Admin page, so they should be trust worthy for display. When this is the case, we can use the |safe flag:

<p>{{tut.tutorial\_content|safe}}</p>

{% for tut in tutorials %}

<p>{{tut.tutorial\_title}}</p>

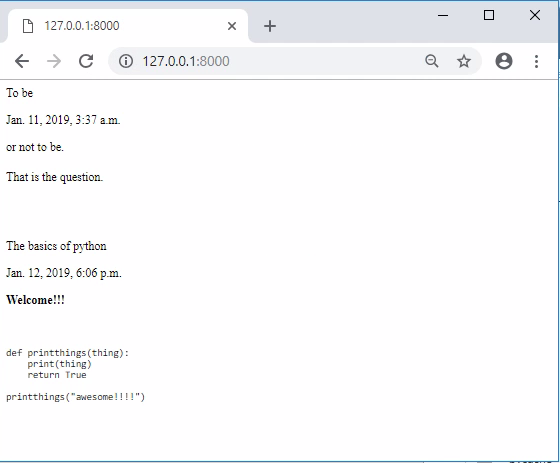
<p>{{tut.tutorial\_published}}</p>

<p>{{tut.tutorial\_content|safe}}</p>

<br><br>

{% endfor %}

Now, refresh the homepage, and you should see:



Very cool. But hmm, our code snippet doesnt actually have any syntax highlighting like it did in the editor, what gives?!

For this, we need to bring in the required CSS and javascript. We need to either load CSS/js locally, or we can also use it when it's hosted somewhere else. Often, we want to have custom javascript or CSS, so we host it locally. In this case, we've got both the CSS and js we need local for the TinyMCE, as it came with the app we installed. So, how might we use it? Django by default is going to look for static files inside of a static directory, much like it looks for templates, which also means we use the same convention : APPNAME/static/APPNAME/

TinyMCE has already done this, however, so we just need to reference it. Before we can reference static files, we need to load in the static files

mysite/main/templates/main/home.html

{% load static %}

<!-- Prism CSS -->

<link href="{% static "tinymce/css/prism.css" %}" rel="stylesheet">

</head>

<body>

{% for tut in tutorials %}

<p>{{tut.tutorial\_title}}</p>

<p>{{tut.tutorial\_published}}</p>

<p>{{tut.tutorial\_content|safe}}</p>

<br><br>

{% endfor %}

<!-- Prism JS -->

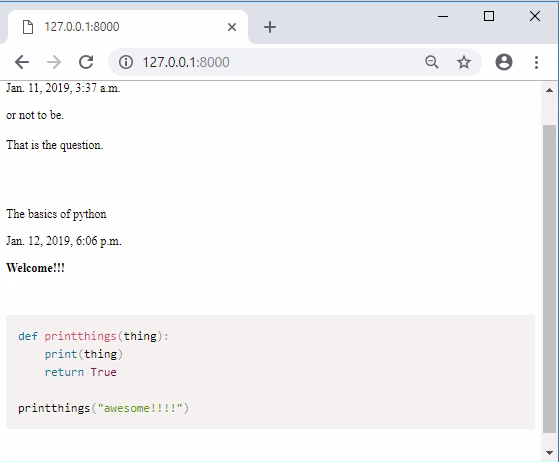
<script src="{% static "tinymce/js/prism.js" %}"></script>

</body>

We use the {% load static %} to load in the static file options. This doesn't actually load in the static files, really just the pathways to static files. Then, we can actually get the path to one with:

{% static "tinymce/css/prism.css" %}

With that, refresh the homepage and the syntax highlighting should be working:



Great, but things are still pretty darn ugly. In the next tutorial, we'll spice things up slightly with a CSS framework.

## CSS - Django Tutorial

Welcome to part 5 of the web development with Django and Python tutorial series. In this part, we're going to focus a bit on styling and design. Just like we have a web framework for Python, there are CSS frameworks (thankfully!)

The CSS framework that I currently use with this website is [**Materialize CSS**](https://materializecss.com/), so I will just use that again here.

We have two options, we can either use their hosted version, or we can download a copy and modify it. To start, lets just use their hosted version, then we can get into modifying it. To use their CSS, we just need to bring it in inside the head tags of our home.html file. The Materialize CSS framework also comes with some javascript that we can make use of, so we will bring that in as well. All you need to know about CSS is that it is used to add certain stylistic elements to certain HTML objects. Javascript, in the context of a CSS framework, is for the same purpose, mainly to add a bit more flair and interactivity. Pretty much any time you have a webpage change dynamically without the URL changing, javascript was involved.

So, to begin, let's modify our mysite/main/templates/main/home.html head tags:

<head>

{% load static %}

<!-- Prism CSS -->

<link href="{% static "tinymce/css/prism.css" %}" rel="stylesheet">

<!-- Compiled and minified CSS -->

<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/css/materialize.min.css">

<!-- Compiled and minified JavaScript for Materialize CSS -->

<script src="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/js/materialize.min.js"></script>

</head>

Great, now refresh, and you will probably just notice that your font changed for the most part, but we've brought in quite a bit that we can use now. For example, let's check out **[cards](https://materializecss.com/cards.html" \t "blank)**. They've got the following code:

<div class="row">

<div class="col s12 m6">

<div class="card blue-grey darken-1">

<div class="card-content white-text">

<span class="card-title">Card Title</span>

<p>I am a very simple card. I am good at containing small bits of information.

I am convenient because I require little markup to use effectively.</p>

</div>

<div class="card-action">

<a href="#">This is a link</a>

<a href="#">This is a link</a>

</div>

</div>

</div>

</div>

That makes these things called cards, so now let's try to incorporate this. Inside the row tags, let's setup our for loop:

<div class="row">

{% for tut in tutorials %}

<div class="col s12 m6">

<div class="card blue-grey darken-1">

<div class="card-content white-text">

<span class="card-title">Card Title</span>

<p>I am a very simple card. I am good at containing small bits of information.

I am convenient because I require little markup to use effectively.</p>

</div>

<div class="card-action">

<a href="#">This is a link</a>

<a href="#">This is a link</a>

</div>

</div>

</div>

{% endfor %}

</div>

Next, let's replace the card title with our tutorial title, then move published date and the content in. For now, I will just remove the card action bits. So now we've got:

<div class="row">

{% for tut in tutorials %}

<div class="col s12 m6">

<div class="card blue-grey darken-1">

<div class="card-content white-text">

<span class="card-title">{{tut.tutorial\_title}}</span>

<p>{{tut.tutorial\_published}}</p>

<p>{{tut.tutorial\_content|safe}}</p>

</div>

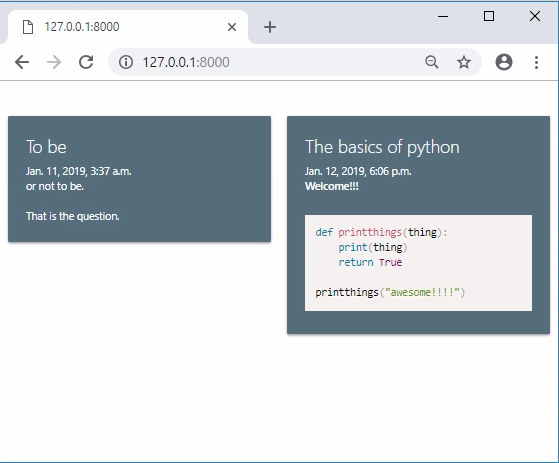
</div>

</div>

{% endfor %}

</div>

Refresh our homepage, and now we've got:



This is just a quick example of applying their CSS. It really is pretty simple to implement their elements. If you're unfamiliar with HTML/CSS there are tons of guides for it out there. I don't intend to really teach it here. Try resizing your window to be very thin, like on mobile. You should see that one of the cards pops down. This is one of the benefits of using the rows/columns structure. The class="col s12 m6" means on a small screen, this object takes up all 12 columns (12 is the max). On a medium screen, 6 columns per card, so we could fit 2 cards, and this will adjust dynamically. There's also a l option for large. So large is like full screen on a desktop, medium is tablets, and small is phone.

We can also add hoverable to our cards, for example:

<div class="card blue-grey darken-1 hoverable">

Try that, then hover over one of the cards, you should see it animate in such a way that it looks like it's lifting up. Next, lets make the published size slightly smaller:

<p style="font-size:70%">Published {{tut.tutorial\_published}}</p>

Okay, that's enough styling for one day I'd say, good work everyone!

Next, we'd probably like to do something like add a navigation bar, maybe some sort of footer. The issue is, we'd like to have this on every page, not just the home page. But we have thousands of tutorials, each of which will get their own page. Imagine adding this to every page you create. Contact page, home page, tutorials page, about page...etc. Then, what if you changed something in the nav bar? Yikes! So, instead, with Django, we can use extends and includes. Let's start with an extends example. The way it works is we create a main type of page, then pages that will maybe modify some portion of that main page (like the body), will "extend" that main type of page. So what we'll do is just copy home.html, and call the copy header.html. In the header.html, replace everything in the body tags with:

{% block content %}

{% endblock %}

So now your mysite/main/templates/main/header.html file looks like:

<head>

{% load static %}

<!-- Prism CSS -->

<link href="{% static "tinymce/css/prism.css" %}" rel="stylesheet">

<!-- Compiled and minified CSS -->

<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/css/materialize.min.css">

<!-- Compiled and minified JavaScript -->

<script src="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/js/materialize.min.js"></script>

</head>

<body>

{% block content %}

{% endblock %}

</body>

<!-- Prism JS -->

<script src="{% static "tinymce/js/prism.js" %}"></script>

Now we can modify mysite/main/templates/main/home.html to be:

{% extends 'main/header.html' %}

{% block content %}

<div class="row">

{% for tut in tutorials %}

<div class="col s12 m6">

<div class="card blue-grey darken-1 hoverable">

<div class="card-content white-text">

<span class="card-title">{{tut.tutorial\_title}}</span>

<p style="font-size:70%">Published {{tut.tutorial\_published}}</p>

<p>{{tut.tutorial\_content|safe}}</p>

</div>

</div>

</div>

{% endfor %}

{% endblock %}

So the extends bit lets django templating know that this code is going to go where ever the block tags are on the template we're extending.

"Where's the nav bar you promised?!" Okay okay, settle down, no need to riot. Let's add that next. Here's the docs for the navbar: **[Materialize CSS Navbar](https://materializecss.com/navbar.html" \t "blank)**

Here's the code for a navbar:

<nav>

<div class="nav-wrapper">

<a href="#" class="brand-logo">Logo</a>

<ul id="nav-mobile" class="right hide-on-med-and-down">

<li><a href="sass.html">Sass</a></li>

<li><a href="badges.html">Components</a></li>

<li><a href="collapsible.html">JavaScript</a></li>

</ul>

</div>

</nav>

Okay, easy enough, so we'll add this to our header.html file, inside the body tags:

<head>

{% load static %}

<!-- Prism CSS -->

<link href="{% static "tinymce/css/prism.css" %}" rel="stylesheet">

<!-- Compiled and minified CSS -->

<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/css/materialize.min.css">

<!-- Compiled and minified JavaScript -->

<script src="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/js/materialize.min.js"></script>

</head>

<body>

<nav>

<div class="nav-wrapper">

<a href="#" class="brand-logo">Logo</a>

<ul id="nav-mobile" class="right hide-on-med-and-down">

<li><a href="sass.html">Sass</a></li>

<li><a href="badges.html">Components</a></li>

<li><a href="collapsible.html">JavaScript</a></li>

</ul>

</div>

</nav>

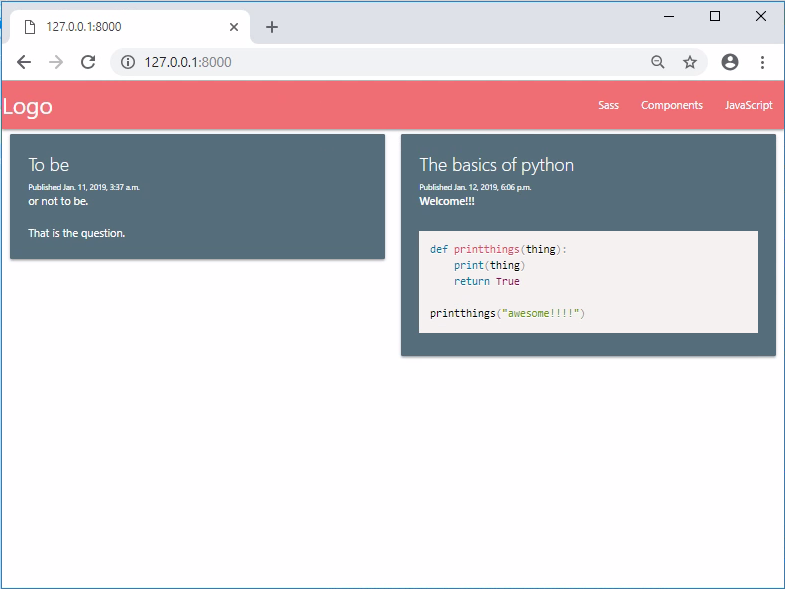
{% block content %}

{% endblock %}

</body>

<!-- Prism JS -->

<script src="{% static "tinymce/js/prism.js" %}"></script>



Let's change the URLs to be something more related to what we're doing here: home, community, register, and login:

<head>

{% load static %}

<!-- Prism CSS -->

<link href="{% static "tinymce/css/prism.css" %}" rel="stylesheet">

<!-- Compiled and minified CSS -->

<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/css/materialize.min.css">

<!-- Compiled and minified JavaScript -->

<script src="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/js/materialize.min.js"></script>

</head>

<body>

<nav>

<div class="nav-wrapper">

<a href="/" class="brand-logo">Tutorials</a>

<ul id="nav-mobile" class="right hide-on-med-and-down">

<li><a href="/">Home</a></li>

<li><a href="https://discord.gg/sentdex">Community</a></li>

<li><a href="/login">Login</a></li>

<li><a href="/register">Register</a></li>

</ul>

</div>

</nav>

{% block content %}

{% endblock %}

</body>

<!-- Prism JS -->

<script src="{% static "tinymce/js/prism.js" %}"></script>

We obviously haven't yet built out these links yet, but we'll do it eventually!

So the next thing is... I don't want these colors, I want the colors from PythonProgramming.net. How can we go about doing this? We can go the Sass route. If you don't care, about this, you do not need to follow from here. We're only going to be changing the default colors for the rest of the tutorial here.

Head to the **[getting started](https://materializecss.com/getting-started.html" \t "blank)** section and then choose the source download under "Sass" to get a .zip file.

Extract the zip file.

Navigate into the folder, and into the materialize-src directory.

Then click sass > components.

Now, since we're just going to fiddle with the colors, open up: \_color-variables.scss, using sublime text. In here, we can adjust the main colors used. This could take a lot of time, so I am just going to share the changed file, but you can feel free to change the colors and fiddle around. There are many hex color code color pickers online to help you with this.

Here's my version of the color variables file: **[\_color-variables.scss](https://pythonprogramming.net/static/downloads/django-tutorials/color-variables.scss" \t "blank)**.

Once you have your variables set, you're ready to compile to CSS. To do this, you need a Sass compiler. I personally used **[Koala](http://koala-app.com/" \t "blank)**. Download that, install.

For some reason, the drag and drop doesn't work for me on Windows. Worked fine on Mac. On Windows, I just use the + button on the top left. Navigate to the Materialize-src folder and open that. Then, you can click once on the materialize-src/sass/materialize.scss to choose it, then click compile on the bottom right.

Now you can navigate to the materialize-src/css section to grab your customized materialize.css file. Great!

Now let's take that and place it into mysite/main/static/main/css/materialize.css. Once that's done, let's reference this new css rather than the hosted one. Go into your mysite/main/templates/main/header.html and swap the hosted CSS for:

<link rel="stylesheet" href="{% static "main/css/materialize.css" %}">

Refresh the homepage and now the navbar, for example, should be a different color. I am going to go ahead and change the color of the cards now to the default white, editing mysite/main/templates/main/home.html:

{% extends 'main/header.html' %}

{% block content %}

<div class="row">

{% for tut in tutorials %}

<div class="col s12 m6">

<div class="card hoverable">

<div class="card-content">

<span class="card-title">{{tut.tutorial\_title}}</span>

<p style="font-size:70%">Published {{tut.tutorial\_published}}</p>

<p>{{tut.tutorial\_content|safe}}</p>

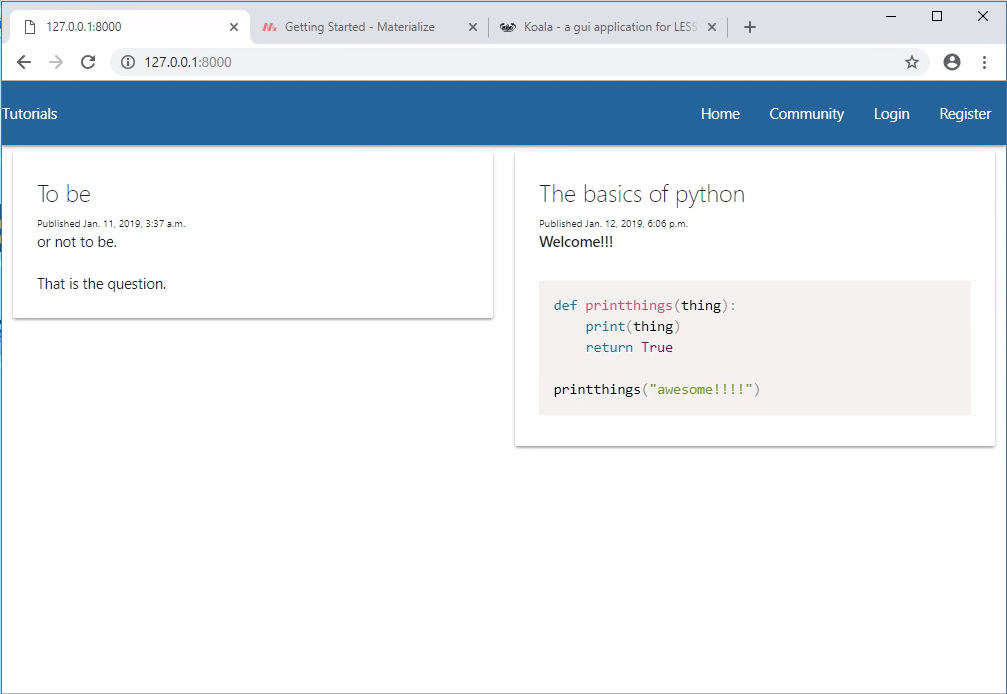
</div>

</div>

</div>

{% endfor %}

{% endblock %}



Not bad, and, while this isn't a great looking website still, this is at least better than what we had! We could continue to build it out stylistically, but I know that's not why you are all here, so, maybe later on in the series, but let's get back to to the Django stuff!

## User Registration - Django Tutorial

Hello and welcome to part 6 of the web development in Python with Django tutorials. In this tutorial, we're going to work on handling user accounts, which includes registering, logging in, and logging out. Along with this, we're also going to dynamically change the navigation bar to reflect if a user is logged in or not. Alright, let's get to it!

To begin, we need some way to create users. We can actually manually create users now in our Admin page, but of course we probably don't want to do that for every user on our website. Django already has a User model, which you may find is enough for you, or you can extend it later. Checking it out:

$ python3 manage.py shell

Python 3.7.2 (v3.7.2:9a3ffc0492, Dec 24 2018, 02:44:43)

[Clang 6.0 (clang-600.0.57)] on darwin

Type "help", "copyright", "credits" or "license" for more information.

(InteractiveConsole)

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

>>> dir(User)

['DoesNotExist', 'EMAIL\_FIELD', 'Meta', 'MultipleObjectsReturned', 'REQUIRED\_FIELDS', 'USERNAME\_FIELD', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getstate\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_setstate\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', '\_check\_column\_name\_clashes', '\_check\_field\_name\_clashes', '\_check\_fields', '\_check\_id\_field', '\_check\_index\_together', '\_check\_indexes', '\_check\_local\_fields', '\_check\_long\_column\_names', '\_check\_m2m\_through\_same\_relationship', '\_check\_managers', '\_check\_model', '\_check\_model\_name\_db\_lookup\_clashes', '\_check\_ordering', '\_check\_property\_name\_related\_field\_accessor\_clashes', '\_check\_single\_primary\_key', '\_check\_swappable', '\_check\_unique\_together', '\_do\_insert', '\_do\_update', '\_get\_FIELD\_display', '\_get\_next\_or\_previous\_by\_FIELD', '\_get\_next\_or\_previous\_in\_order', '\_get\_pk\_val', '\_get\_unique\_checks', '\_meta', '\_password', '\_perform\_date\_checks', '\_perform\_unique\_checks', '\_save\_parents', '\_save\_table', '\_set\_pk\_val', 'check', 'check\_password', 'clean', 'clean\_fields', 'date\_error\_message', 'date\_joined', 'delete', 'email', 'email\_user', 'first\_name', 'from\_db', 'full\_clean', 'get\_all\_permissions', 'get\_deferred\_fields', 'get\_email\_field\_name', 'get\_full\_name', 'get\_group\_permissions', 'get\_next\_by\_date\_joined', 'get\_previous\_by\_date\_joined', 'get\_session\_auth\_hash', 'get\_short\_name', 'get\_username', 'groups', 'has\_module\_perms', 'has\_perm', 'has\_perms', 'has\_usable\_password', 'id', 'is\_active', 'is\_anonymous', 'is\_authenticated', 'is\_staff', 'is\_superuser', 'last\_login', 'last\_name', 'logentry\_set', 'natural\_key', 'normalize\_username', 'objects', 'password', 'pk', 'prepare\_database\_save', 'refresh\_from\_db', 'save', 'save\_base', 'serializable\_value', 'set\_password', 'set\_unusable\_password', 'unique\_error\_message', 'user\_permissions', 'username', 'username\_validator', 'validate\_unique']

>>>

Noting some of those attributes, we can see there are fields for username, email, names...etc. We might not need an email for example, or names, which is fine. We'll see how we can use just what we need for this model (which you could apply to any model, really).

I tend to think in order of events that a user will go through, so my first thought is to create the actual registration page, the template. So I'll start there, but just remember with any new page that you need to consider the M, V, and C (model, view, and controller)!

So let's go into main's templates, and copy the home.html file and call the copy register.html, modifying it to be:

{% extends 'main/header.html' %}

{% block content %}

<form method="POST">

{% csrf\_token %}

{{form}}

</form>

If you already have an account <a href="/login" target="blank"><strong>login</strong></a> instead.

{% endblock %}

Alright, some basics here, the form tag just denotes that there will be a form, and the method means this will create a POST request (meaning data will be submitted), as opposed to a GET request where the hope is to get some data in return.

Next, we have

{% csrf\_token %}

This is a special token aimed at combatting something called Cross Site Request Forgery, which you can learn more about here: [Cross-Site Request Forgery (CSRF)](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)" \t "blank).

Next, we just have a simple form variable, which you should be able to guess is going to be passed as context into this template. Finally, we just have a bit of text in case the user miss-clicked to signup, and just needs to go to login instead.

Great, we have our template, now we need to code the view that was render it. Let's head to mysite/main/views.py

We will call this function register. While we could build this form, we can also import the user creation form:

from django.contrib.auth.forms import UserCreationForm

Then build the register function:

def register(request):

form = UserCreationForm

return render(request = request,

template\_name = "main/register.html",

context={"form":form})

Okay, the template is done, the view is done. That's the model and the view boxes checked. All that's left is the controller! Head to mysite/main/urls.py and let's add that now!

We just need to add the path:

path("/register", views.register, name="register"),

Full file:

from django.urls import path

from . import views

app\_name = 'main' # here for namespacing of urls.

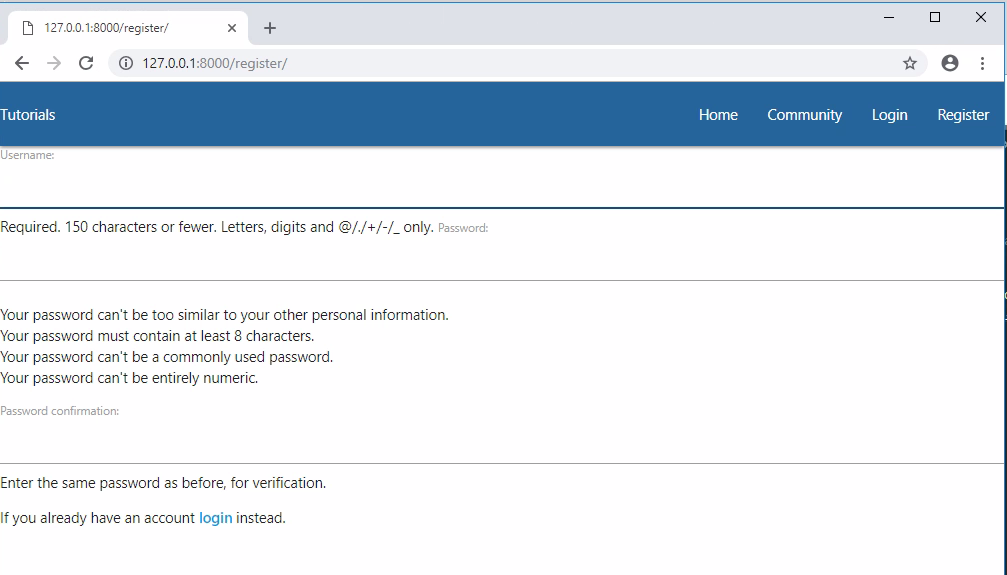
urlpatterns = [

path("", views.homepage, name="homepage"),

path("register/", views.register, name="register"),

]

Are we done? I think we're done! Let's click on register in the navbar!



Okay, we're off to a decent start. I wish the form was not right up to the left edge like that though. Let's fix that by putting the body content in a container div.

mysite/main/templates/main/header.html

...

<div class="container">

{% block content %}

{% endblock %}

</div>

...

Then maybe a break or two in the register.html

{% extends 'main/header.html' %}

{% block content %}

<br>

<form method="POST">

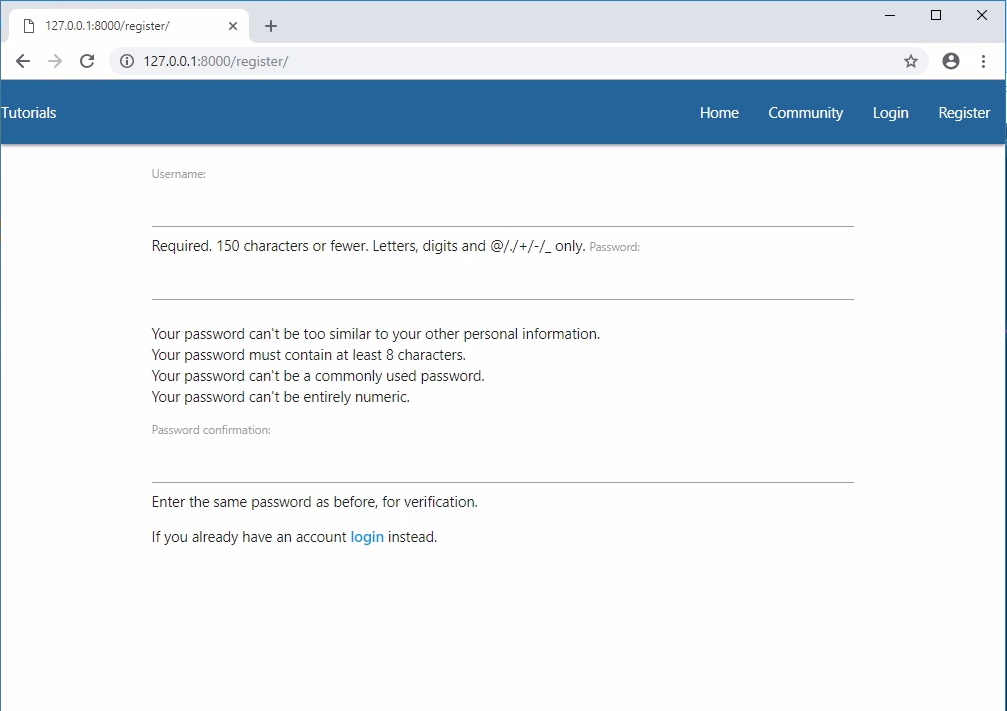
{% csrf\_token %}

{{form}}

</form>

If you already have an account <a href="/login" target="blank"><strong>login</strong></a> instead.

{% endblock %}



Great, but the form doesn't actually do anything, so let's change that next!

Inside our form tags, let's add a submit button:

<button style="background-color:#F4EB16; color:blue" class="btn btn-outline-info" type="submit">Sign Up</button>

So now our full mysite/main/templates/main/register page is:

{% extends 'main/header.html' %}

{% block content %}

<div class="container">

<form method="POST">

{% csrf\_token %}

{{form.as\_p}}

<button style="background-color:#F4EB16; color:blue" class="btn btn-outline-info" type="submit">Sign Up</button>

</form>

If you already have an account <a href="/login" target="blank"><strong>login</strong></a> instead.

</div>

{% endblock %}

Next, forms have various attributes that we can use:

* as\_p - paragraph tags
* as\_table - as a table
* as\_ul - as an unordered list

You might not have noticed, but our form is a little confusing at the first passowrd, it's hard to tell what goes there at a quick glance. let's instead set our form as\_p to remedy this. Full page:

{% extends 'main/header.html' %}

{% block content %}

<br>

<form method="POST">

{% csrf\_token %}

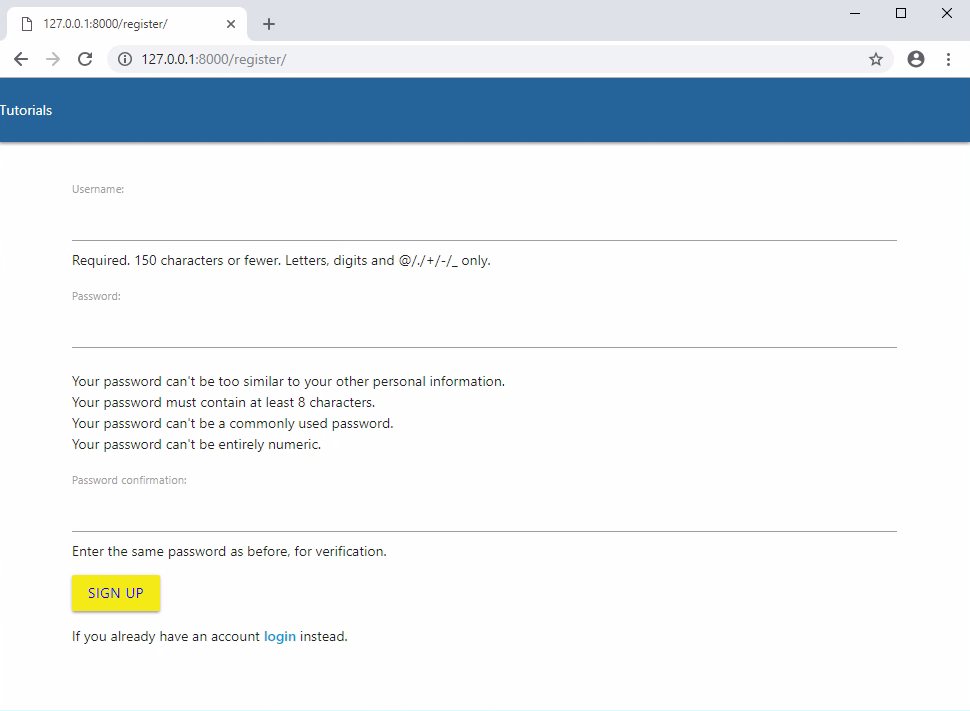
{{form.as\_p}}

<button style="background-color:#F4EB16; color:blue" class="btn btn-outline-info" type="submit">Sign Up</button>

</form>

If you already have an account <a href="/login" target="blank"><strong>login</strong></a> instead.

{% endblock %}



Now that our form has a submit button, we can fill out the form, try to sign up... but nothing happens? Well, right now our registration page has no handling for when there's a POST request, so it's actually just reloading the registration page. Where might we need to edit in order to handle for this type of request?

We can handle for specific requests and other types of logic from within our views.py, under the register function that we've started.

The very first thing we should do is check to see if the request made to this page has been a POST:

if request.method == "POST":

If it has, then we're going to map the submitted form to the UserCreationForm:

form = UserCreationForm(request.POST)

If the form is valid, then we just save this new User object (which is part of the form, so we save the form)

user = form.save()

This will save the user, but, in general, it's nice if the user just registered to also log them in so they don't need to re-type things again. To do this, we can just use Django's login function which we can grab from django.contrib.auth, which also gives use logout, and authenticate...all of which we'll need, so let's grab all three with:

from django.contrib.auth import logout, authenticate, login

So now we can log in the user with:

username = form.cleaned\_data.get('username')

login(request, user)

Now we just want to send the user to the homepage. While we could just...render...the homepage now, we also need to change the URL, and, since we're actually sending them to the homepage, we want to redirect them there now. Add redirect to your imports from django.shortcuts:

from django.shortcuts import render, redirect

So now we can just do:

return redirect("main:homepage")

This will look in urls.py for the app\_name called main (recall we did this with: app\_name = 'main' ), then it finds the url with the name of "homepage." So now our full register function is:

def register(request):

if request.method == "POST":

form = UserCreationForm(request.POST)

if form.is\_valid():

user = form.save()

username = form.cleaned\_data.get('username')

login(request, user)

return redirect("main:homepage")

form = UserCreationForm

return render(request = request,

template\_name = "main/register.html",

context={"form":form})

In some cases, the form wont be valid for whatever reason, however, so we also need to handle for that:

def register(request):

if request.method == "POST":

form = UserCreationForm(request.POST)

if form.is\_valid():

user = form.save()

username = form.cleaned\_data.get('username')

login(request, user)

return redirect("main:homepage")

else:

for msg in form.error\_messages:

print(form.error\_messages[msg])

return render(request = request,

template\_name = "main/register.html",

context={"form":form})

form = UserCreationForm

return render(request = request,

template\_name = "main/register.html",

context={"form":form})

Now, we can register another user, then they're sent back to the homepage!

Full views.py just in case you're missing something:

from django.shortcuts import render, redirect

from .models import Tutorial

from django.contrib.auth.forms import UserCreationForm, AuthenticationForm

from django.contrib.auth import logout, authenticate, login

# Create your views here.

def homepage(request):

return render(request = request,

template\_name='main/home.html',

context = {"tutorials":Tutorial.objects.all})

def register(request):

if request.method == "POST":

form = UserCreationForm(request.POST)

if form.is\_valid():

user = form.save()

username = form.cleaned\_data.get('username')

login(request, user)

return redirect("main:homepage")

else:

for msg in form.error\_messages:

print(form.error\_messages[msg])

return render(request = request,

template\_name = "main/register.html",

context={"form":form})

form = UserCreationForm

return render(request = request,

template\_name = "main/register.html",

context={"form":form})

So now we've found ourselves at a point where we need something more than a print statement for information. There's nothing to inform the user once they've logged in, and there's nothing to tell us that we've done something wrong if we have. This is where messaging comes in, which

## Messages - Django Tutorial

Welcome to part 7 of the web development with Python and Django tutorial series, where we'll be covering messaging and continuing to cover handling for users within our web app by popping up messages to the user, changing the navbar depending on if they are logged in or not, as well as showing the use for includes in Django templating.

The Django web frameworks comes with a messaging system that allows us to store messages that we can check for on each page load. If there are some messages, we can display them to the user.

For these messages, we could show them however we see fit. With materialize.css, there are things called toasts, which use javascript to popup a little snippet of a message to your user. For more information: **[Materialize Toasts](https://materializecss.com/toasts.html" \t "blank)**.

In our case right now, we'd like to be able to quickly inform the user of errors with their form, or alert them to the success of logging in. This is the sort of thing that Django messaging is for, and we can use the toasts to display the messages if they happen to exist.

To begin, let's add the cases for messages. Inside our mysite/main/views.py, make the following import:

from django.contrib import messages

Next, after a user has been created, we can do something like:

messages.success(request, f"New account created: {username}")

Then, for our errors, we could handle them with:

for msg in form.error\_messages:

messages.error(request, f"{msg}: {form.error\_messages[msg]}")

Full register function:

def register(request):

if request.method == "POST":

form = UserCreationForm(request.POST)

if form.is\_valid():

user = form.save()

username = form.cleaned\_data.get('username')

messages.success(request, f"New account created: {username}")

login(request, user)

return redirect("main:homepage")

else:

for msg in form.error\_messages:

messages.error(request, f"{msg}: {form.error\_messages[msg]}")

return render(request = request,

template\_name = "main/register.html",

context={"form":form})

form = UserCreationForm

return render(request = request,

template\_name = "main/register.html",

context={"form":form})

Once that's done, we're ready to show the messages. We don't need to pass messages through the context, though that's exactly what we'd do if there wasn't something already in place like Django's messaging! Instead, we can just go straight to our templates to handle for them.

We could modify the register.html page to handle for messages, but messages are something we'd like to use all over our website, not just on this specific page, so instead we'll handle this on the header.html page.

Just before the container div that contains our block content, let's add the following:

{% if messages %}

{% for message in messages %}

<script>M.toast({html: "{{message}}", classes: 'blue rounded', displayLength:2000});</script>

{% endfor %}

{% endif %}

Now, create another user, but this time purposefully mess up the password fields. You should now see the error message pop up. Go ahead and register a new user legitimately, and you should see a message about creating a new account successfully!

Next, we could get a bit more fancy with this by setting custom messages depending on the message tag itself.

{% if messages %}

{% for message in messages %}

{% if message.tags == 'success'%}

<script>M.toast({html: "{{message}}", classes: 'green rounded', displayLength:2000});</script>

{% elif message.tags == 'info'%}

<script>M.toast({html: "{{message}}", classes: 'blue rounded', displayLength:2000});</script>

{% elif message.tags == 'warning'%}

<script>M.toast({html: "{{message}}", classes: 'orange rounded', displayLength:10000});</script>

{% elif message.tags == 'error'%}

<script>M.toast({html: "{{message}}", classes: 'red rounded', displayLength:10000});</script>

{% endif %}

{% endfor %}

{% endif %}

Next, we should dynamically handle for the user being logged in. Once someone has registered and logged in, we shouldnt show the register and login buttons. Instead maybe an account and logout button.

Looking above the messages part of the header.html file, let's look specifically at the ul tags for the navbar:

<ul id="nav-mobile" class="right hide-on-med-and-down">

<li><a href="/">Home</a></li>

<li><a href="https://discord.gg/sentdex">Community</a></li>

<li><a href="/login">Login</a></li>

<li><a href="/register">Register</a></li>

</ul>

Here, we can do a check to see if the user is authenticated. If they are, then instead we should show a logout button and maybe a user account. If not, then we show login/register.

<li><a href="/">Home</a></li>

<li><a href="https://discord.gg/sentdex">Community</a></li>

{% if user.is\_authenticated %}

<li><a href="/account">{{user.username|title}}</a></li>

<li><a href="/logout">Logout</a></li>

{% else %}

<li><a href="/login">Login</a></li>

<li><a href="/register">Register</a></li>

{% endif %}

At this point, we've got some cool functionality for our header, but, in all honesty, things are starting to get messy for this header page. As we continue to add functionality like this, it's only going to become more and more confusing and time consuming to add things or change them. This is probably a good time to incorporate includes. These includes are similar to extends, where we can just place one line of code to call in an include, which may contain many more lines of code. Let's see an example. First, let's change our navbar list items to be an include.

Inside of our mysite/main/template/maindirectory, let's make another directory called includes. Includes are just snippets of HTML usually, so I would like to organize them into their own directory.

Inside the new directory, let's create a new HTML file called navbaritems.html. Edit that, and let's copy and paste in the list items and the logic for the navbar.

mysite/main/templates/main/includes/navbaritems.html

<li><a href="/">Home</a></li>

<li><a href="https://discord.gg/sentdex">Community</a></li>

{% if user.is\_authenticated %}

<li><a href="/account">{{user.username|title}}</a></li>

<li><a href="/logout">Logout</a></li>

{% else %}

<li><a href="/login">Login</a></li>

<li><a href="/register">Register</a></li>

{% endif %}

Then, we can head over to our header.html file in the main templates dir and replace the list items there with:

{% include 'main/includes/navbaritems.html' %}

We can now do the same thing with messages. Again, it's useful for us to know "oh okay, the nav options are here" ...but otherwise we probably wont be modifying those very often. Same with messages. We'd like to see where the message code is, but there's really no need for us to have it cluttering up that top level HTML file.

Now, inside that includes directory, make another file called messaging.html and put the following in there:

{% if messages %}

{% for message in messages %}

{% if message.tags == 'success'%}

<script>M.toast({html: "{{message}}", classes: 'green rounded', displayLength:2000});</script>

{% elif message.tags == 'info'%}

<script>M.toast({html: "{{message}}", classes: 'blue rounded', displayLength:2000});</script>

{% elif message.tags == 'warning'%}

<script>M.toast({html: "{{message}}", classes: 'orange rounded', displayLength:10000});</script>

{% elif message.tags == 'error'%}

<script>M.toast({html: "{{message}}", classes: 'red rounded', displayLength:10000});</script>

{% endif %}

{% endfor %}

{% endif %}

Then remove those lines from the main template and replace with:

{% include 'main/includes/messaging.html' %}

Now our header.html is just:

<head>

{% load static %}

<!-- Prism CSS -->

<link href="{% static "tinymce/css/prism.css" %}" rel="stylesheet">

<!-- Compiled and minified CSS -->

<link rel="stylesheet" href="{% static "main/css/materialize.css" %}">

<!-- Compiled and minified JavaScript -->

<script src="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/js/materialize.min.js"></script>

</head>

<body>

<nav>

<div class="nav-wrapper"

<a href="/" class="brand-logo">Tutorials</a>

<ul id="nav-mobile" class="right hide-on-med-and-down">

{% include 'main/includes/navbaritems.html' %}

</ul>

</div>

</nav>

{% include 'main/includes/messaging.html' %}

<div class="container">

{% block content %}

{% endblock %}

</div>

</body>

<!-- Prism JS -->

<script src="{% static "tinymce/js/prism.js" %}"></script>

Now the next thing we need to handle for is allowing a user to actually login or logout. Right now, the only way to log in is by registering, and there's no great way to log out! Let's fix both in the next tutorial.

## User Login and Logout - Django Tutorial

Welcome to part 8 of the web development with Python and Django tutorial series. Here, we're going to continue working on our user handling and we will be bringing in the ability for a user to login and log out.

Let's start with logout, since that's super easy. We don't need to design anything for it and Django already has a logout function!

Okay, let's start with the URL, it should be /logout, so let's head into main/mysite/urls.py and add a path for that:

from django.urls import path

from . import views

app\_name = 'main' # here for namespacing of urls.

urlpatterns = [

path("", views.homepage, name="homepage"),

path("register/", views.register, name="register"),

path("logout", views.logout\_request, name="logout"),

]

Now that we have the URL for it, let's create the view, which we've already decided will be a function called logout\_request

Here's a great example of where you might decide to call your function "logout"...but we're trying to use logout from Django. We'd definitely like to not make that mistake, so make sure to not call it the same as the function we're going to be trying to use. Same thing when we go to make our login function...we'll have to call it something other than that!

mysite/main/views.py

...

def logout\_request(request):

logout(request)

messages.info(request, "Logged out successfully!")

return redirect("main:homepage")

...

Now, refresh our website, and click logout. Works!

Okay, now seems like a good time to do the login page. To begin, let's start in views.py since we're already here anyway. We'll start with:

from django.contrib.auth.forms import UserCreationForm, AuthenticationForm

...

def login\_request(request):

form = AuthenticationForm()

return render(request = request,

template\_name = "main/login.html",

context={"form":form})

Next, let's work on the html page: mysite/main/templates/main/login.html

{% extends 'main/header.html' %}

{% block content %}

<div class="container">

<form method="POST">

{% csrf\_token %}

{{form.as\_p}}

<button style="background-color:#F4EB16; color:blue" class="btn btn-outline-info" type="submit">Login</button>

</form>

Don't have an account? <a href="/register" target="blank"><strong>register here</strong></a>!

</div>

{% endblock %}

Finally, we just need to add a link in urls.py:

path("login", views.login\_request, name="login"),

Okay, now if we go to http://127.0.0.1:8000/login, we can see a login page. Awesome, except that, just like our registration page, nothing will happen unless we handle for the POST request, which is where the meat of this page will actually be:

mysite/main/views.py

def login\_request(request):

if request.method == 'POST':

form = AuthenticationForm(request=request, data=request.POST)

if form.is\_valid():

username = form.cleaned\_data.get('username')

password = form.cleaned\_data.get('password')

user = authenticate(username=username, password=password)

if user is not None:

login(request, user)

messages.info(request, f"You are now logged in as {username}")

return redirect('/')

else:

messages.error(request, "Invalid username or password.")

else:

messages.error(request, "Invalid username or password.")

form = AuthenticationForm()

return render(request = request,

template\_name = "main/login.html",

context={"form":form})

With that, go ahead and try the login functionality and you should find that you're all set.

Finally, let's change one last thing about our user registration. At the moment, we're not collecting the user's email, which could turn out to be fairly problematic later for things like forgotten passwords. How can we change this? We could either completely build our own form, or we could just modify the Django one a bit. I am going to opt to just extend the Django form. To do this, create a new file: mysite/main/forms.py:

forms.py

from django import forms

from django.contrib.auth.forms import UserCreationForm

from django.contrib.auth.models import User

class NewUserForm(UserCreationForm):

email = forms.EmailField(required=True)

class Meta:

model = User

fields = ("username", "email", "password1", "password2")

def save(self, commit=True):

user = super(NewUserForm, self).save(commit=False)

user.email = self.cleaned\_data["email"]

if commit:

user.save()

return user

Then, inside of our views.py, we instead need to import our NewUserForm

views.py

from .forms import NewUserForm

Then replace the two instances of UserCreationForm with NewUserForm in the register function. Now make sure you're logged out and register a new user. This time, there will be a password field.

We didn't need to update any models because we're using the Django User model, which already had an email field.

Next, if we were to actually add full-length tutorials, we'd be in some trouble, because our home page would be quickly filled up. We'll be talking about how to handle for this by using foreign keys to point to tutorial series and categories.

## Foreign Keys with Models - Django Tutorial

Welcome to part 9 of the web development with Python and Django tutorial series. In this tutorial, we're going to work on the organization of tutorials through the use of foreign keys which will allow us to tie certain elements in a database table to another database table's data.

Consider our current problem. We have some tutorials, but they're very short. Once we start adding the full length of tutorials, our home page will be nearly impossible to navigate. So then maybe we just show cards with the tutorial's title? The problem here is we still have over a thousand tutorials. Way too much!

... ok ok, why don't we order by series? Well, we still have over 50 series, which is still a lot of clutter.

So, we organize tutorials by series, and those series by category like "data analysis" or "fundamentals."

Ideally though, however, we'd just want to pick the series for a tutorial, and not need to pick both the series AND category every time, plus this would take up unnecessary database space.

Thus, what we instead do is create 2 new models: series and category. Then, tutorials will have a foreign key that point to the series they belong to, and series will point to a specific category, and this is how we have relationships between tables. Let's see how this works.

To begin, let's start by creating the TutorialCategory model.

mysite/main/models.py

class TutorialCategory(models.Model):

tutorial\_category = models.CharField(max\_length=200)

category\_summary = models.CharField(max\_length=200)

category\_slug = models.CharField(max\_length=200, default=1)

class Meta:

# Gives the proper plural name for admin

verbose\_name\_plural = "Categories"

def \_\_str\_\_(self):

return self.tutorial\_category

The category\_slug is the URL that we want to point to this category. For example if the category is "Data Analysis," then what's the URL that will point to this category, should a user click on the "Data Analysis" card.

Next, we'll make another model TutorialSeries, which will have a foreign key that points to the TutorialCategory model

class TutorialSeries(models.Model):

tutorial\_series = models.CharField(max\_length=200)

tutorial\_category = models.ForeignKey(TutorialCategory, default=1, verbose\_name="Category", on\_delete=models.SET\_DEFAULT)

series\_summary = models.CharField(max\_length=200)

class Meta:

# otherwise we get "Tutorial Seriess in admin"

verbose\_name\_plural = "Series"

def \_\_str\_\_(self):

return self.tutorial\_series

The only new thing here that you may not understand is the on\_delete bit. Basically, we need to know what to do with the referenced objects when the main one is deleted. When we delete a category, we don't really want to delete the tutorials from that category, nor visa versa, so instead we're opting to SET\_DEFAULT here. If the category gets deleted, then the tutorials that have that category will have their categories set to their default values rather than deleted.

Now, finally, the tutorial itself:

class Tutorial(models.Model):

tutorial\_title = models.CharField(max\_length=200)

tutorial\_content = models.TextField()

tutorial\_published = models.DateTimeField('date published')

#https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ForeignKey.on\_delete

tutorial\_series = models.ForeignKey(TutorialSeries, default=1, verbose\_name="Series", on\_delete=models.SET\_DEFAULT)

tutorial\_slug = models.CharField(max\_length=200, default=1)

def \_\_str\_\_(self):

return self.tutorial\_title

Okay, that's a lot of changes! We've definitely changed and added models, so it's time for a makemigrations and migrate!

python manage.py makemigrations

Migrations for 'main':

main\migrations\0005\_auto\_20190116\_1349.py

- Create model TutorialCategory

- Create model TutorialSeries

- Add field tutorial\_slug to tutorial

- Alter field tutorial\_published on tutorial

- Add field tutorial\_series to tutorial

python manage.py migrate

Operations to perform:

Apply all migrations: admin, auth, contenttypes, main, sessions

Running migrations:

Applying main.0005\_auto\_20190116\_1349... OK

Now that we've done all this, we also need to make modifications to our Admin page to see categories, series, and to be able to pair tutorials to series.

In mysite/main/admin.py

from .models import Tutorial, TutorialSeries, TutorialCategory

...

class TutorialAdmin(admin.ModelAdmin):

fieldsets = [

("Title/date", {'fields': ["tutorial\_title", "tutorial\_published"]}),

("URL", {'fields': ["tutorial\_slug"]}),

("Series", {'fields': ["tutorial\_series"]}),

("Content", {"fields": ["tutorial\_content"]})

]

formfield\_overrides = {

models.TextField: {'widget': TinyMCE(attrs={'cols': 80, 'rows': 30})},

}

admin.site.register(TutorialSeries)

admin.site.register(TutorialCategory)

admin.site.register(Tutorial,TutorialAdmin)

Now that we've done that, it's time to go into the admin, create 2 or 3 categories, a few tutorial series, and a few tutorials per series. You're free to just put gobbly goop (yes that's a technical term) in there, just to see how this all works. In the next tutorial, we'll start off assuming you've got a few categories, series, and tutorials per series, and show how we can go about organizing these on our website.

Working with Foreign Keys - Django Tutorial

Welcome to part 10 of the web development with Django tutorial series. In this tutorial, we're going to show an example of how we can interact with our various tables linked together by foreign keys.

If you have not created your tutorials, series, and categories, I've hosted my version 9 with all of those added here: [Part 9 with added tutorials, series, and categories](https://pythonprogramming.net/static/downloads/django-tutorials/django-9-with-series-and-topics-n-such.zip).

On PythonProgramming.net, every series has always been contained all within "one slash." Back in the early days, files were served exactly where they sat on the server, so you organized them usually in directories. So you might have something like pythonprogramming.net/data-analysis/pandas-intro, because you had to. Nowadays, especially with a model like what we're using, we don't need this. I am not a fan of long URLs, so I opted historically to not do this and instead do pythonprogramming.net/pandas-intro.

That said, this means every single tutorial resides under ONE slash. With a framework like Django in mind, I might not have set out to do this, because it means we have to handle for this all within a single view function pretty much, other than for pages where we know the exact "slug." This will all hopefully make more sense as we go, so now I am just going to get started.

For now, I am going to refer to a case where you just have one slash example: pythonprogramming.net/data-analysis/pandas-intro as a "single" slug. At the moment, we have two cases for a "single slug:"

* Category (data analysis, fundamentals...etc)
* Specific tutorial

So, we begin filling our function with:

from .models import Tutorial, TutorialCategory, TutorialSeries

from django.http import HttpResponse

...

def single\_slug(request, single\_slug):

categories = [c.category\_slug for c in TutorialCategory.objects.all()]

if single\_slug in categories:

return HttpResponse(f"{single\_slug} is a category")

tutorials = [t.tutorial\_slug for t in Tutorial.objects.all()]

if single\_slug in tutorials:

return HttpResponse(f"{single\_slug} is a Tutorial")

return HttpResponse(f"'{single\_slug}' does not correspond to anything we know of!")

Next, let's go ahead and modify the homepage function as well to instead give us the categories to iterate over, rather than the tutorials:

def homepage(request):

return render(request=request,

template\_name='main/categories.html',

context={"categories": TutorialCategory.objects.all})

Finally, let's add that mysite/main/templates/categories.html:

{% extends 'main/header.html' %}

{% block content %}

<div class="row">

{% for cat in categories %}

<div class="col s12 m6 l4">

<a href="{{cat.category\_slug}}", style="color:#000">

<div class="card hoverable">

<div class="card-content">

<div class="card-title">{{cat.tutorial\_category}}</div>

<!--<p style="font-size:70%">Published {{tut.tutorial\_published}}</p>-->

<p>{{cat.category\_summary}}</p>

</div>

</div>

</a>

</div>

{% endfor %}

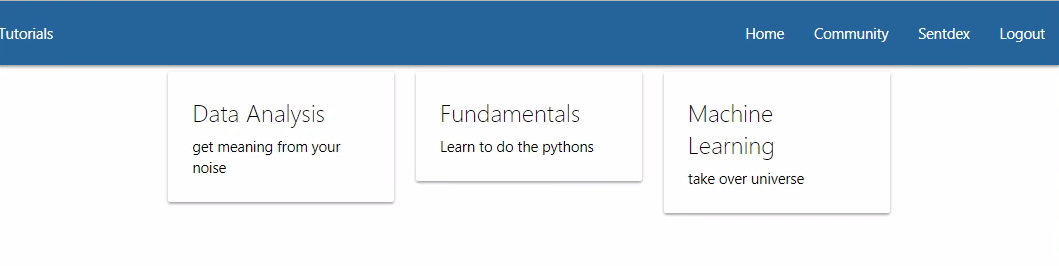
</div>

{% endblock %}

Finally, we just need to add in our controller bit in the urls.py. This is where we've unfortunately got this "single slug" every where thing going on, because we can actually have variables within our URL handling:

path("<single\_slug>", views.single\_slug, name="single\_slug"),

With that added to our views, go ahead and refresh the homepage and you should instead see:



Now click on one of the categories. You should get a message like: machine-learning is a category.

Okay, let's populate this page correctly now. If it's a category, we want to display the tutorial series that fall under that category rather than just serving this message!

Once we know the category, we want to then get all of the tutorial series objects that match that category. We will do this with a filter:

matching\_series = TutorialSeries.objects.filter(tutorial\_category\_\_category\_slug=single\_slug)

Next, when someone clicks on a tutorial series, what should happen? For now, we'd like to send them to part 1. Maybe later, we might rather have the functionality in place to send the user back to wherever they left off if they have started. Regardless, this will link to some tutorial specifically. To do this, I am going to just use a dictionary to map the series to part 1:

for m in matching\_series.all():

part\_one = Tutorial.objects.filter(tutorial\_series\_\_tutorial\_series=m.tutorial\_series).earliest("tutorial\_published")

series\_urls[m] = part\_one.tutorial\_slug

Then we'd render. Full code for this views.py function now:

z

def single\_slug(request, single\_slug):

# first check to see if the url is in categories.

categories = [c.category\_slug for c in TutorialCategory.objects.all()]

if single\_slug in categories:

matching\_series = TutorialSeries.objects.filter(tutorial\_category\_\_category\_slug=single\_slug)

series\_urls = {}

for m in matching\_series.all():

part\_one = Tutorial.objects.filter(tutorial\_series\_\_tutorial\_series=m.tutorial\_series).earliest("tutorial\_published")

series\_urls[m] = part\_one.tutorial\_slug

return render(request=request,

template\_name='main/category.html',

context={"tutorial\_series": matching\_series, "part\_ones": series\_urls})

Now we need to make this category.html page:

mysite/main/templates/main/category.html

{% extends 'main/header.html' %}

{% block content %}

<div class="row">

{% for tut, partone in part\_ones.items %}

<div class="col s12 m6 l4">

<a href="{{partone}}", style="color:#000">

<div class="card hoverable">

<div class="card-content">

<div class="card-title">{{tut.tutorial\_series}}</div>

<!--<p style="font-size:70%">Published {{tut.tutorial\_published}}</p>-->

<p>{{tut.series\_summary}}</p>

</div>

</div>

</a>

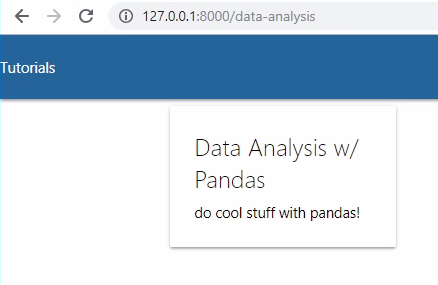
</div>

{% endfor %}

</div>

{% endblock %}

Now, if we click on a category, we get something like:



Now, if we click on a tutorial, we get a message like pandas-intro is a Tutorial.

So we need to handle for a specific tutorial, which I am going to save for the next tutorial along with the tutorial sidebar.

## Dynamic sidebar - Django Tutorial

Welcome to part 11 of the Django tutorial series. In this part, we're going to finally point to specific tutorials, along with adding the side bar to help users navigate through that specific tutorial.

Inside our views.py, and the single\_slug function, our tutorial code now becomes:

tutorials = [t.tutorial\_slug for t in Tutorial.objects.all()]

if single\_slug in tutorials:

this\_tutorial = Tutorial.objects.get(tutorial\_slug=single\_slug)

return render(request = request,

template\_name='main/tutorial.html',

context = {"tutorial":this\_tutorial})

Now we need to make that tutorial page.

{% extends 'main/header.html' %}

{% block content %}

<div class="row">

<div class="col s12, m8, l8">

<h3>{{tutorial.tutorial\_title}}</h3>

<p style="font-size:70%">Published {{tutorial.tutorial\_published}}</p>

{{tutorial.tutorial\_content|safe}}

</div>

</div>

{% endblock %}

With this, we can click on a tutorial now and that works now.

Next, we might want to have a side-bar that helps users navigate through the parts of the tutorial series like pythonprogramming.net currently has.

To do this, head back to views.py

tutorials = [t.tutorial\_slug for t in Tutorial.objects.all()]

if single\_slug in tutorials:

this\_tutorial = Tutorial.objects.get(tutorial\_slug=single\_slug)

tutorials\_from\_series = Tutorial.objects.filter(tutorial\_series\_\_tutorial\_series=this\_tutorial.tutorial\_series).order\_by('tutorial\_published')

this\_tutorial\_idx = list(tutorials\_from\_series).index(this\_tutorial)

return render(request=request,

template\_name='main/tutorial.html',

context={"tutorial: "this\_tutorial,

"sidebar": tutorials\_from\_series,

"this\_tut\_idx": this\_tutorial\_idx})

Now we can use this newly passed data in our tutorial.html template. We can make a side bar with:

<div class="col s12 m4 l4">

<ul class="collapsible popout">

{% for tutorial in sidebar %}

{% if forloop.counter0 == this\_tut\_idx %}

<li class="active">

<div class="collapsible-header">{{tutorial.tutorial\_title}}<br>(currently viewing)</div>

</li>

{% else %}

<li>

<div class="collapsible-header">{{tutorial.tutorial\_title}}</div>

<div class="collapsible-body">

<p><a href="/{{tutorial.tutorial\_slug}}"><button class="btn waves-effect waves-light right-align" style="background-color:yellow; color:black">Go</button></a></p>

</div>

</li>

{% endif %}

{% endfor %}

</ul>

</div>

Full code for tutorial.html:

{% extends 'main/header.html' %}

{% block content %}

<div class="row">

<div class="col s12, m8, l8">

<h3>{{tutorial.tutorial\_title}}</h3>

<p style="font-size:70%">Published {{tutorial.tutorial\_published}}</p>

{{tutorial.tutorial\_content|safe}}

</div>

<div class="col s12 m4 l4">

<ul class="collapsible popout">

{% for tutorial in sidebar %}

{% if forloop.counter0 == this\_tut\_idx %}

<li class="active">

<div class="collapsible-header">{{tutorial.tutorial\_title}}<br>(currently viewing)</div>

</li>

{% else %}

<li>

<div class="collapsible-header">{{tutorial.tutorial\_title}}</div>

<div class="collapsible-body">

<p><a href="/{{tutorial.tutorial\_slug}}"><button class="btn waves-effect waves-light right-align" style="background-color:yellow; color:black">Go</button></a></p>

</div>

</li>

{% endif %}

{% endfor %}

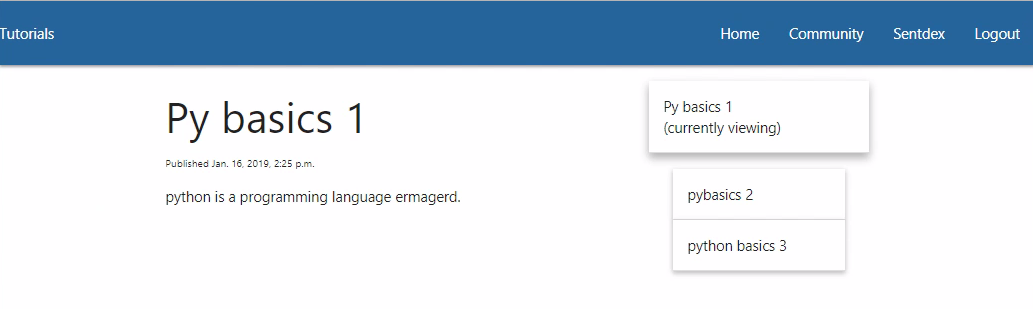
</ul>

</div>

</div>

{% endblock %}

Now, viewing any of the tutorials should give you something like:



If we click on the side bar though, it appears to not quite function right. This is because we need to initialize the collapsible javascript. That said, as time goes on, we're going to likely need to initialize a few, or many, of the javascript things. Instead, we can just do them all with:

<script>

M.AutoInit();

</script>

So let's just add that to our header file like:

header.html

<head>

{% load static %}

<!-- Prism CSS -->

<link href="{% static "tinymce/css/prism.css" %}" rel="stylesheet">

<!-- Compiled and minified CSS -->

<link rel="stylesheet" href="{% static "main/css/materialize.css" %}">

<!-- Compiled and minified JavaScript -->

<script src="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/js/materialize.min.js"></script>

</head>

<body>

<nav>

<div class="nav-wrapper"

<a href="/" class="brand-logo">Tutorials</a>

<ul id="nav-mobile" class="right hide-on-med-and-down">

{% include 'main/includes/navbaritems.html' %}

</ul>

</div>

</nav>

{% include 'main/includes/messaging.html' %}

<div class="container">

{% block content %}

{% endblock %}

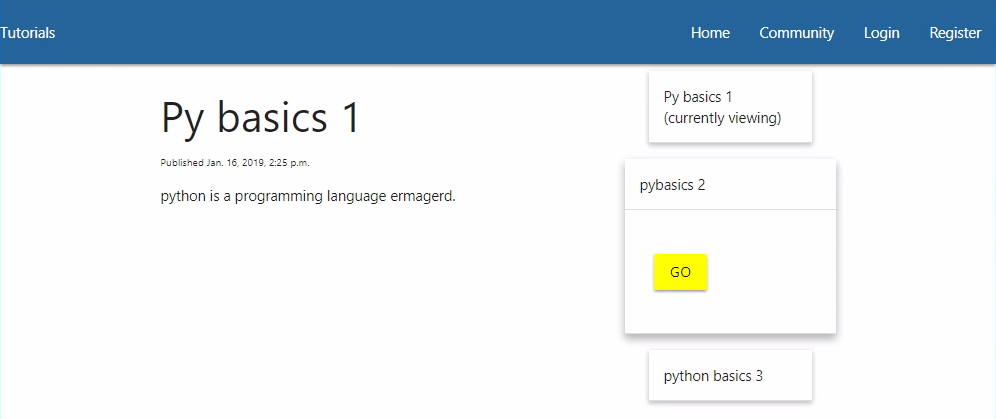
</div>

</body>

<script src="{% static "tinymce/js/prism.js" %}"></script>

<script>M.AutoInit();</script>

With that, we should have no trouble clicking on and interacting with the nav bar.



Ok, at this point, we've covered the bulk of what you need to know about how to use Django. I may continue to develop this Django version of my site further and incorporate a bunch more, and I may cover that process. That said, much of it will be done using concepts you've already learned up to this point.

So, the last thing I want to show is deployment to an actual server where you can access your website from anywhere in the world, which is what we'll be doing in the next tutorial. See you there!

## Deploying to a Server - Django Tutorial

Welcome to part 12 of the Django tutorials. In this tutorial, we're going to focus on the deployment of a Django website. There are many steps involved with setting up just about any web server including steps for added security, improved performance, and more. We could dedicate an entire series to this process alone. If there's interest for that, I can do that, but, for now, my goal is to condense this enough to just one tutorial.

We're going to be using [Linode](https://www.linode.com/sentdex) as our host here. I have been using Linode for almost a decade, this website has been hosted on them for almost as long. They are also a sponsor for this series, so thank you Linode! If you want to support me while using a high quality host, check out the [Linode $20 Credit](https://linode.com/sentdex) to get started.

To begin, we need a server to host our website on. You could do this from home, but, most likely, you'll want something with a bit better reliability. When I hosted things from home, I saw more like 90% reliability. Things like power power outages and times of heavy traffic would cause trouble. Then, looking for servers, there are places like [Heroku](https://www.heroku.com/" \t "blank) or [PythonAnywhere](https://www.pythonanywhere.com/" \t "blank), where you can host your websites for free, or for cheap, with the added benefit of having very quick and easy ways to deploy something like a Django app.

The main issue I have with these types of websites is when it comes to scaling and further customization. Moving off of these services can often be a huge pain. I have used both services, and at one time I was one of the top resource users at PythonAnywhere. Now, I am sure lots have changed with both services since I last used them, things are always improving, but I think it's often best to have your own server, which you have root/sudoer access to. If you're just looking for a quick way to share your web application with others, or so you can access it publicly away from home, then check out those two services, or something like them. If you're looking to actually deploy something to be used by many users, which you will need to scale out, then you may want to dig deeper.

So, we need some server, which will almost certainly be running a linux operating system, such as Ubuntu. You can either get a truly dedicated server, or something called a VPS (virtual private server). For VPS providers, I have made use of Google, Digital Ocean, Linode, AWS, Paperspace, and a bunch of others. Various hosts have various benefits. When just starting out, I'd recommend a simpler-to-use host like Google, Linode, or Digital Ocean. I personally find AWS very tedious to use, and I know to some extent what I am doing. I've been with Linode for the longest, almost a decade now, and Linode is what I use to host this website, so that's what I'll be using here. You can use my referral code for Linode to get $20 to start with, though you will still need to sign up with a valid credit card so they know you're not just constantly creating new accounts $20 at a time:

[Linode $20 Credit](https://linode.com/sentdex" \o "Linode $20 credit)

The smallest $5/month server is more than enough for our needs here, so the referral code should last you a few months.

Once you have that server, we need Django and then we need a web server to actually handle requests and such. Up to this point, we've been using the built-in Django testing webserver, but this is not suitable for production both in terms of security and performance/scale. Instead, you're looking at something like Apache or Nginx. Initial setup on various VPS providers is going to vary, but, on [Linode](https://www.linode.com/sentdex" \t "blank), we'll start by creating and account/logging in going to the [cloud manager dashboard](https://cloud.linode.com/dashboard" \t "blank), then clicking the "create" button, and then "linode." The term "linode" is used for a typical VPS here.

Next, we choose an image. I am going to use Ubuntu 18.04 LTS here for the long term support and upgrade-ability down the road. Next, pick a region. Pick something either close to you or close to your average user. I'll go with Dallas.

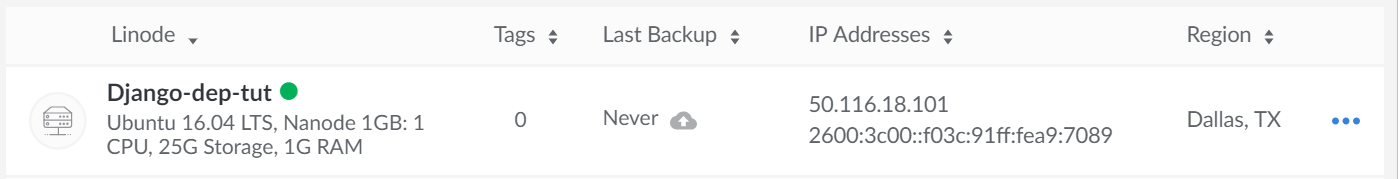
Next, pick the plan. I am going to go with the Nanode type, with 1gb of ram, 1 cpu (1 core, basically), and 25gb of storage. This should be enough for testing.

Now we will give our linode a label if we want, then set a root user password.

The last thing you may want to consider is setting backups. I can say this: I have never regretted setting up backups. You can disable them later. They are often not worth the cost compared to doing automated backups yourself, but you have to do that first. Til then, let your host do backups if it's an option!

Set up the server, and this will take a few moments. Once done, you're ready to connect to your server. To connect to our server, we're going to use SSH. On Mac and Linux, this is as simple as starting by opening your terminal. On Windows, you will need to instead open your powershell, or use a third-party program like [PuTTY](https://www.putty.org/" \t "blank) (which is still useful, because you can also save various connections as profiles). For now, I would just recommend you use powershell though if you're on Windows.

In your terminal/powershell, you will need to type: ssh root@YOUR.VPS.IP.ADDR. To get your IP address, just head to [cloud.linode.com](https://cloud.linode.com/linodes" \t "blank) and look for the IP address of your new server. It should look something like this:



So my IP address is 50.116.18.101. To connect then I type:

ssh root@50.116.18.101

If this is your first ever time connecting to this server, you will see a message like:

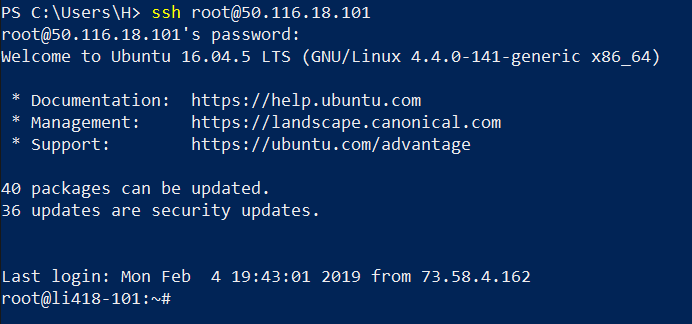
The authenticity of host '50.116.18.101 (2300:2c00::f05c:92ff:fe84:196d)' can't be established.

ECDSA key fingerprint is SHA256:MeZ8pafsfSF!2fff6XRJbP/HelzorT.

Are you sure you want to continue connecting (yes/no)?

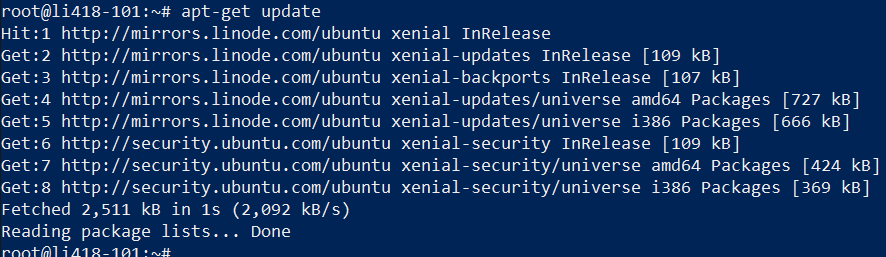
Type yes

On initial login, you should see something like:



Now, you're connected! So for many of you, this might be an entirely new, unfamiliar, and uncomfortable experience. You will be interacting entirely with this machine through the shell like this, all in text...no desktop...etc. For many of you as well, not only is it a new way of interacting with a machine, it's probably a foreign operating system (linux). It's just plain going to be a learning process. Don't feel like you need to learn everything here at once. I will explain the commands that we're doing, but the objective here is to set everything up. If you want to learn Linux and the shell, you may want to find a tutorial dedicated to that specifically.

We're ready to start issuing commands. The first one is an update for apt, which is your advanced package tool. This is what we use to install, update, and remove packages. Like pip for Python. First, let's run an update:

apt-get update 

With current versions of Ubuntu, you should have python 3 already. Type in python3 to confirm that you're running python 3. In this case, we have Python 3.5. That's fine for now. If you want to upgrade Python 3, have at it, but I am trying to stuff all this into one tutorial.

What we will need is pip for python 3:

apt-get install -y python3-pip

Then let's update pip:

python3 -m pip install --upgrade pip

Now we can install django and the tinymce app that we used so far:

python3 -m pip install django django-tinymce4-lite

Now let's make our web dir:

mkdir /var/www

Then change directory into that:

cd /var/www

Now, let's make a quick demo django project to see how things work:

django-admin startproject mysite

Change directory into that project:

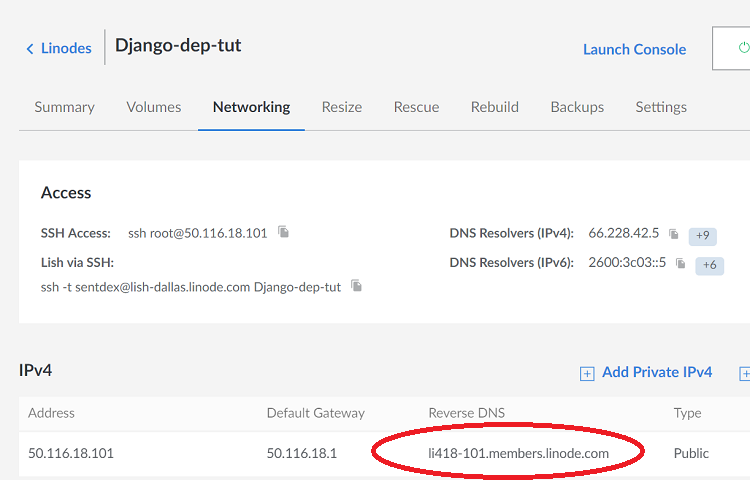
cd mysite

Next, we're going to edit the settings.py file, using the editor called nano.

nano mysite/settings.py

The biggest thing is to turn off DEBUG, setting this to False.

Next, we want to set the hostname. This might be your domain name, but, again, I am not going to get into pointing a domain name here. To do it, you'd make a domain name, point that domain name to linode's nameservers, then setup that domain to point to your linode on linode.com. That's the 1-sentence run down, should you want to do it! Linode, however, will give you a temporary domain name as well though. To find it, go to your [Linodes section](https://cloud.linode.com/linodes" \t "blank) and click on your new linode. From here, go to the networking tab, and there you will find your hostname:



So mine is li418-101.members.linode.com

Back into our settings.py, I will paste that into my ALLOWED\_HOSTS list:

ALLOWED\_HOSTS = ["li418-101.members.linode.com"]

Now, control+x to exit, y to save, enter to keep the filename.

Next, we will make a quick app to test to make sure the basic Django installation works. The one we saw before when we just made a project and ran it was based on DEBUG being set to true. We'd still like to know if things are working though, so a quick app will do!

python3 manage.py startapp it\_works

Then we will point our main/urls.py to this app:

nano mysite/urls.py

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path('', include('it\_works.urls')),

path('admin/', admin.site.urls),

]

Exit and save (ctrl+x, y, enter). Now we have to make that urls.py:

nano it\_works/urls.py

from django.urls import path

from . import views

app\_name = 'it\_works'

urlpatterns = [

path("", views.homepage, name="homepage"),

]

nano it\_works/views.py

from django.http import HttpResponse

def homepage(request):

return HttpResponse("It works! (served from Django)")

Okay, the Django bits are all setup, and now we need to install/setup our web server WSGI. The web server is the thing that helps us communicate with outside traffic, handling requests and such. WSGI, standing for Web Server Gateway Interface is software that serves as the intermediary between our web server and your Django code.

apt-get install -y apache2 libapache2-mod-wsgi-py3

Now let's setup the apache config file for our website:

nano /etc/apache2/sites-available/mysite.conf

Filling that with:

<VirtualHost \*:80>

ServerName YOURHOSTNAME

ErrorLog ${APACHE\_LOG\_DIR}/mysite-error.log

CustomLog ${APACHE\_LOG\_DIR}/mysite-access.log combined

WSGIDaemonProcess mysite processes=2 threads=25 python-path=/var/www/mysite

WSGIProcessGroup mysite

WSGIScriptAlias / /var/www/mysite/mysite/wsgi.py

Alias /robots.txt /var/www/mysite/static/robots.txt

Alias /favicon.ico /var/www/mysite/static/favicon.ico

Alias /static/ /var/www/mysite/static/

Alias /static/ /var/www/mysite/media/

<Directory /var/www/mysite/mysite>

<Files wsgi.py>

Require all granted

</Files>

</Directory>

<Directory /var/www/mysite/static>

Require all granted

</Directory>

<Directory /var/www/mysite/media>

Require all granted

</Directory>

</VirtualHost>

Note that you must change YOURHOSTNAME to your hostname.

Short explanation of what we're doing here:  
  
- `ServerName hostname` - replace `hostname` with your current hostname. That's the line that "tells" Apache to use this configuration when given domain/hostname is used in a request (eg. when you run http://hostname in a browser)  
- `ErrorLog` and `CustomLog` are default log configuration lines for Apache, we just want to save logs for that `vhost` in separate files  
- `WSGIDaemonProcess`, `WSGIProcessGroup` and `WSGIScriptAlias` sets our `WSGI` mod - in our configuration we want to have 2 processes with 25 threads each and `/var/www/mysite` as a current working directory (that's a path where we have all files and folders fof our website), then we are instructing Apache to use `/var/www/mysite/mysite/wsgi.py`, file created for us by Django, to serve website under `/` (root path) - treat this `wsgi.py` file like entry point for your project  
- `Alias` lines set associations between different files and folders we need in a root of our domain to actual files and folders in a filesystem structure, so for example it says: if browser asks for `http://hostname/robots.txt`, it's here: `/var/www/mysite/static/robots.txt` - return it's content. We want to set those aliases for static files, that are actually saved on a disk only, all dynamic pages generated by Django are going to be served from Python script (using `wsgi.py` entry point)  
- last lines are here to set permissions for Apache to open files and folders from given locations

Save and exit.

rm /etc/apache2/sites-available/000-default.conf

nano /etc/apache2/sites-available/000-default.conf

<VirtualHost \*:80>

ServerName \_

Redirect 404 /

</VirtualHost>

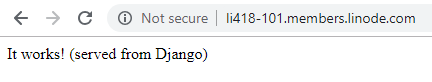
Save and exit.

The above just ensures that, if our hostname doesn't match, we return a 404.

Now enable our new site and reload apache:

a2ensite mysite  
systemctl reload apache2

Alright, head to your hostname in a browser and view your work! You should see something like:



Cool, but we've got our project that we've been working on so far, so let's bring that in! Since I am already hosting our project files, I am just going to download from there. You can also use SCP to move your own projects. You can use SCP on terminal in mac/linux, or with powershell in Windows. You can also download a GUI-based SCP program called WinSCP, which you may find more noob-friendly since you can just drag and drop your files, and, like PuTTY, you can save profiles.

For this tutorial's sake, I am just going to grab my zipped file that I have hosted.

cd /var/www/  
wget https://pythonprogramming.net/static/downloads/django-tutorials/django-11.zip

wget is used to download files. Now we want to unzip these:

apt-get install unzip

Then, let's remove the old mysite that we made:

rm -r mysite/

rm is remove, the -r is recursive (so remove everything in there).

Next, let's unzip our tutorial code:

unzip django-11.zip

Okay, great! Now we just need to turn off DEBUG and set the hostname again, then reload apache.

nano mysite/mysite/settings.py

Set DEBUG = False and add your hostname.

Then scroll to the end of the file, you should find STATIC\_URL = '/static/', add right below it:

STATIC\_ROOT = '/var/www/mysite/static/'

MEDIA\_URL = '/media/'

MEDIA\_ROOT = '/var/www/mysite/media/'

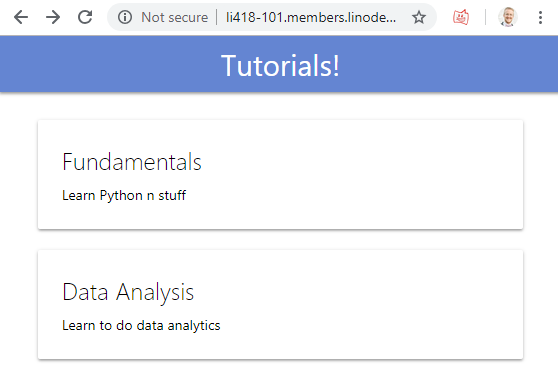
Exit that, and now we need to run:

python3 manage.py collectstatic

This will copy all of your static files to one spot for production.

service apache2 reload

Check to see if things are working!



Awesome!

One thing some people pointed out up to this point however is that... suddenly the paths for our login, register, and logout don't work!

This is because of our trailing slash. Let's remove those.

nano mysite/main/urls.py

from django.urls import path

from . import views

app\_name = "main"

urlpatterns = [

path("", views.homepage, name="homepage"),

path("register", views.register, name="register"),

path("logout", views.logout\_request, name="logout"),

path("login", views.login\_request, name="login"),

path("<single\_slug>", views.single\_slug, name="single\_slug"),

]

Exit and save. Reload apache. Try the methods again.

Finally, we can try to login, but we find that we can't because our database is read-only for our user. In this case, using apache, our "user" is www-data. We need to give this user permission to edit these files for us. To do that:

cd /var/www/

chown www-data mysite/

Then:

chown www-data mysite/db.sqlite3

Let's run a

service apache2 reload

Now we can interact with our database. I think we're all set!