Basic Django Tutorial

30 January 2017

What is Django/Python?

So, before we start, I am going to explain the basic structure of a web application from the perspective of a backend developer in Michigan Biological Software. We receive requests from a website to a specific URL, we figure what data the frontend wants, get the data from the database, and then respond with that data. These tasks are handled with Django, which is a framework that specializes in making these jobs simple. To use Django, we write Python code that uses the framework to execute these tasks.

Starting Up

To begin our tutorial project, we just need to ensure that we have a version of Django and Python installed (these should have been installed during previous tutorials). Next, create a folder where the project will be and type into your terminal:

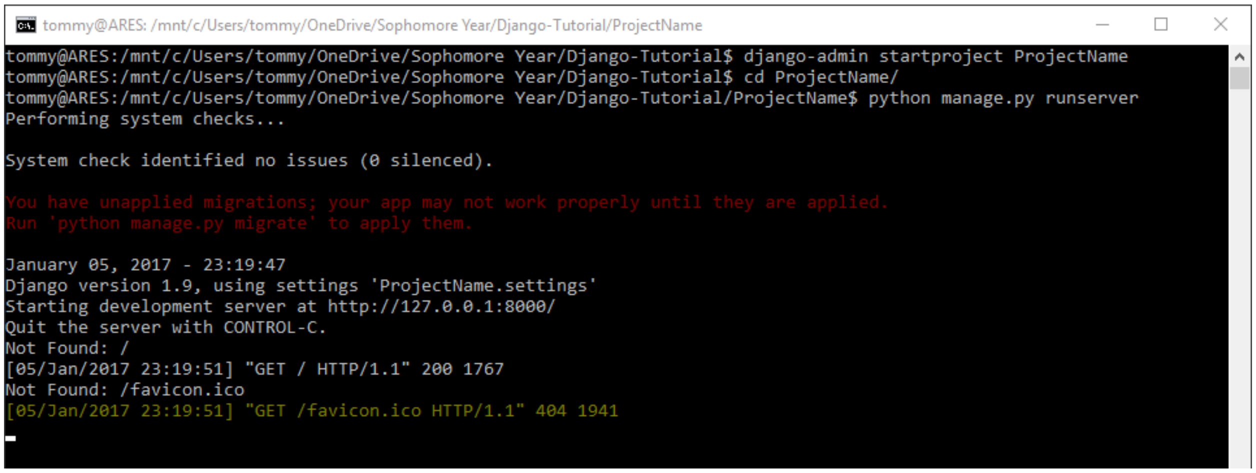
```
django-admin startproject Tutorial
```

This will create a folder, inside of which is a manage.py file. This file, written in Python, is where you issue commands to start the server, update the database, etc. You will never need to edit the manage.py file. There will also be a folder called Tutorial next to manage.py and this contains a few settings and basic items we will need to change.

You can immediately test that all this is working by making sure your terminal is in the folder with manage.py and typing (to start up your server):

```
python manage.py runserver
```

and then going to localhost:8000 in one of your browser windows. It should have something welcoming you to your first Django website.


tommy@ARES: /mnt/c/Users/tommy/OneDrive/Sophomore Year/Django-Tutorial/ProjectName

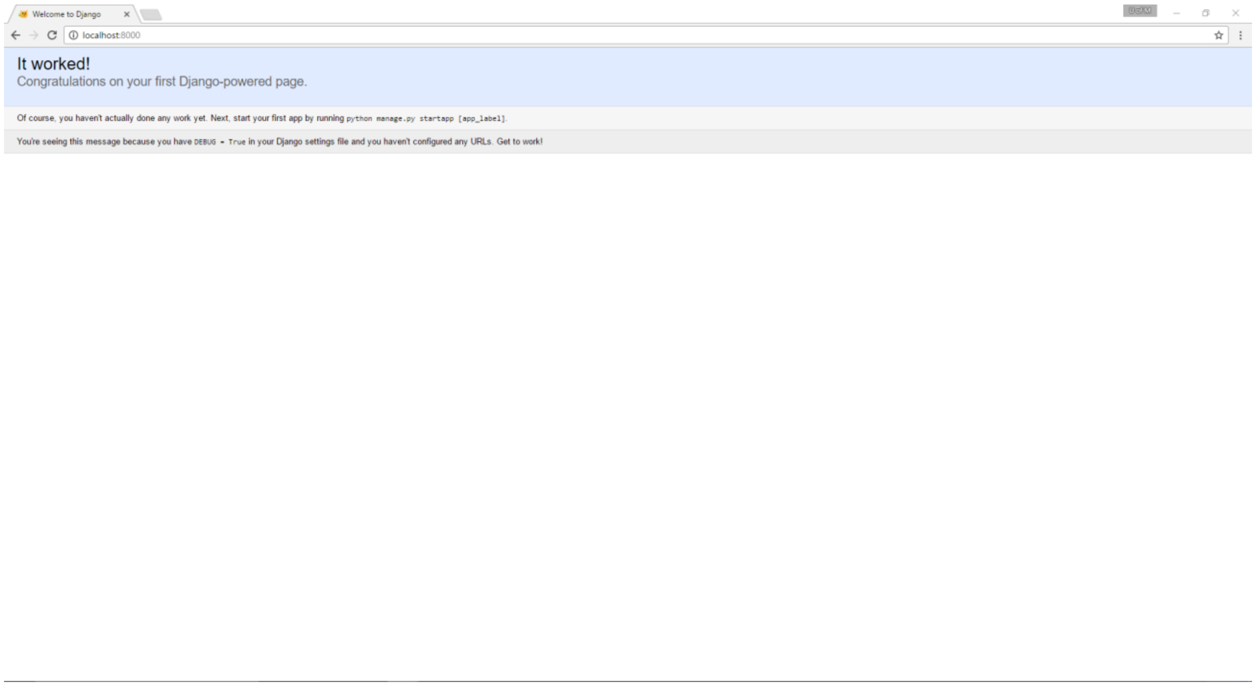
```
tommy@ARES:/mnt/c/Users/tommy/OneDrive/Sophomore Year/Django-Tutorial$ django-admin startproject ProjectName
tommy@ARES:/mnt/c/Users/tommy/OneDrive/Sophomore Year/Django-Tutorial$ cd ProjectName/
tommy@ARES:/mnt/c/Users/tommy/OneDrive/Sophomore Year/Django-Tutorial/ProjectName$ python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may not work properly until they are applied.
Run 'python manage.py migrate' to apply them.

January 05, 2017 - 23:19:47
Django version 1.9, using settings 'ProjectName.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
Not Found: /
[05/Jan/2017 23:19:51] "GET / HTTP/1.1" 200 1767
Not Found: /favicon.ico
[05/Jan/2017 23:19:51] "GET /favicon.ico HTTP/1.1" 404 1941
```

Terminal Output (Styling will look different on Mac and on Powershell)



It worked!
Congratulations on your first Django-powered page.

Of course, you haven't actually done any work yet. Next, start your first app by running `python manage.py startapp [app_label]`.

You're seeing this message because you have `DEBUG = True` in your Django settings file and you haven't configured any URLs. Get to work!

Website on Chrome for the Basic Project

Now, using that same terminal window type:

```
python manage.py startapp TutorialApp
```

This creates a folder that contains the files you will be working with 99% of the time. So, I think it will be beneficial to go over what each of these files do:

urls.py – Routes website requests to specific URLs (ie to google.com/maps) to specific functions in views.py

views.py – Takes the information from a website request, gets the data from the database, and responds to the frontend.

models.py – Contains the structure of the data in the database.

(Side note: I find the best way to visualize the way a database stores data as a bunch of very large Excel spreadsheets, where each row is an item and each column is a specific variable).

Next, we are going to make a few changes to make sure that everything we have done so far works. First, go to the folder Tutorial/Tutorial (should be next to a settings.py file) and add the following code:

```
from django.conf.urls import include, url
from django.contrib import admin
urlpatterns = [
    url(r'^App/', include('TutorialApp.urls')),
    url(r'^admin/', admin.site.urls),
]
```

This makes it so that whenever the project gets a request that looks like “webite.com/App/”, it directs it to the TutorialApp to handle it. The admin part will not really matter for this tutorial, it just allows the administrators to look at the data in the database to make sure everything is working well. Next, we are going to make a change to the Tutorial/Tutorial/settings.py file. Change the INSTALLED_APPS array to also include ‘TutorialApp’ so that the project knows that the app we made even exists. The penultimate change will be to the Tutorial/TutorialApp/urls.py file. We are going to add:

```
from django.conf.urls import url
from . import views
urlpatterns = [
	url(r'^$', views.index, name='index'),
]
```

This makes it so that when the project directs the website.com/App/ URL to this app, if is just website.com/App/, the app will direct this specific request to the function index in the views.py file. So, to finally handle this request, go into Tutorial/TutorialApp/views.py file and add:

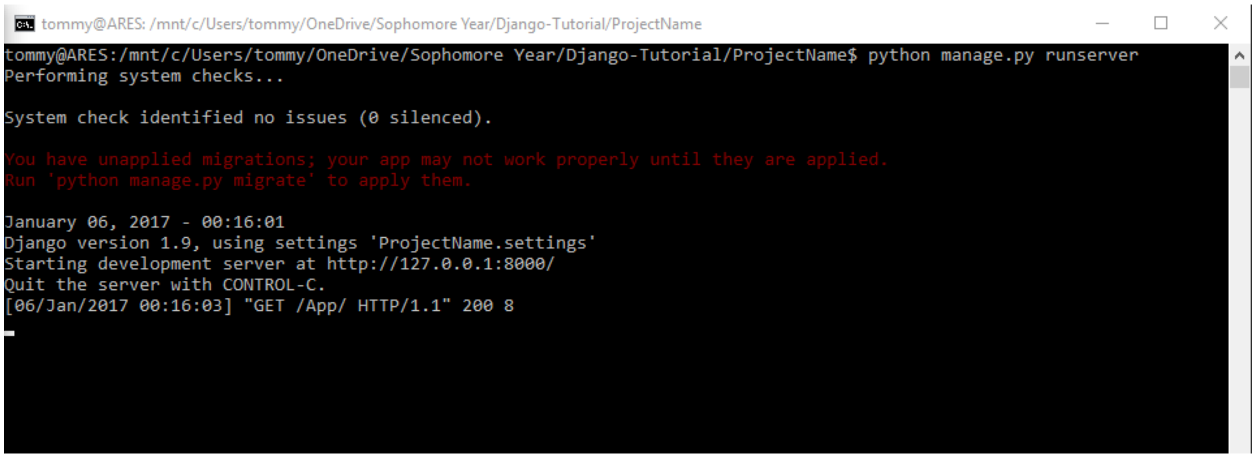
```
from django.http import HttpResponse
def index(request):
	return HttpResponse("It works")
```

The basics of this is that when the function index is called with basic information of the request in the variable request, it simply responds with “It works” (which is not even proper HTML, but it will still work).

To recap, a request goes to the Tutorial/Tutorial/urls.py file, notice that it looks like website.com/App/ and send it to Tutorial/TutorialApp/urls.py. This file will notice the same pattern and send the request to Tutorial/TutorialApp/views.py function index, which will simply respond “It works” to the browser. So now, if you type into your terminal:

```
python manage.py runserver
```

and then go to localhost:8000/App/, “It works” should be the only thing that appears.


tommy@ARES: /mnt/c/Users/tommy/OneDrive/Sophomore Year/Django-Tutorial/ProjectName

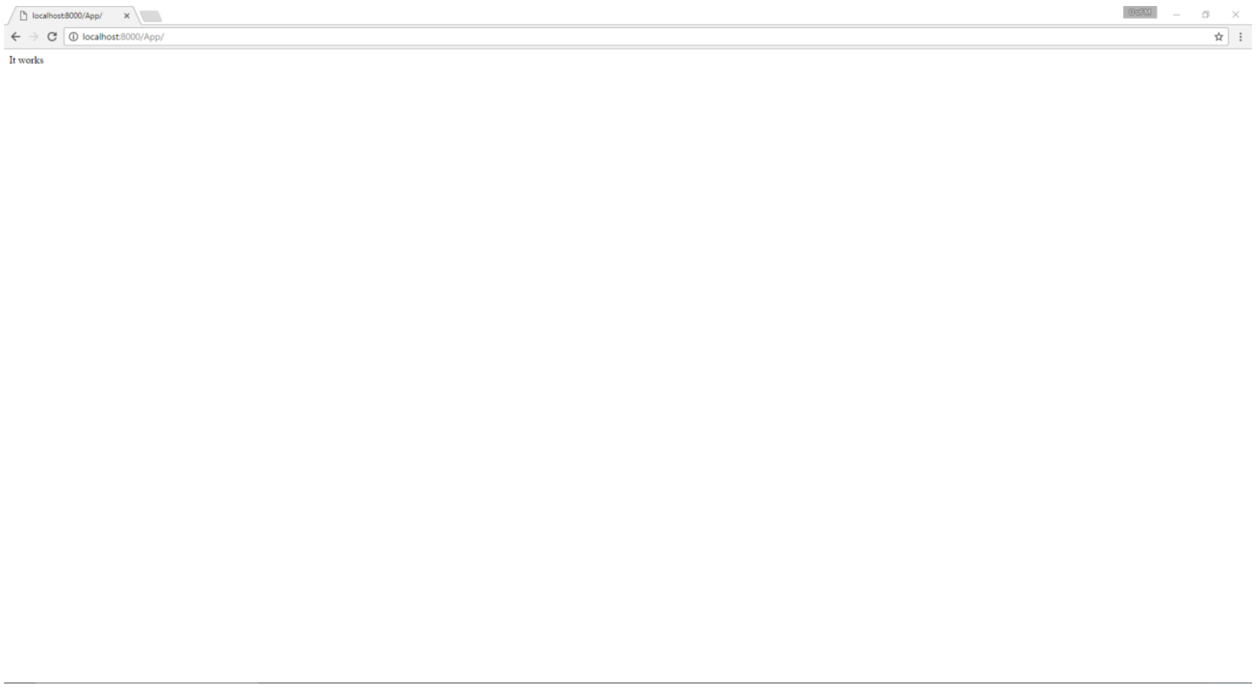
```
tommy@ARES:/mnt/c/Users/tommy/OneDrive/Sophomore Year/Django-Tutorial/ProjectName$ python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may not work properly until they are applied.
Run 'python manage.py migrate' to apply them.

January 06, 2017 - 00:16:01
Django version 1.9, using settings 'ProjectName.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
[06/Jan/2017 00:16:03] "GET /App/ HTTP/1.1" 200 8
```

Starting Server in Terminal



It works

HTTPRequest in Browser

Making Our App Not Make Me Sad

While it is technically possible to make our website using only HttpResponse(html), it would be such bad style that it might as well be wearing jorts (yeah, this is a really bad attempt at a joke in a tutorial likely read by no one, but it made me laugh so I don't care). So, to fix this problem, we are going to move our HTML to another page. Inside of Tutorial/TutorialApp/, create a folder called "templates" (must be exactly this), create a file called "index.html" and put whatever HTML you want in there. Next, change your Tutorial/TutorialApp/views.py file, change it to:

```
from django.shortcuts import render
from django.http import HttpResponse
def index(request):
     context = { }
     return render(request, 'index.html', context)
```

So now whenever index is called, it will return index.html (don't worry about context for now, we will get to it in just a second).

Now, just restart your server and reload the /App/ website and now you should be seeing your HTML webpage.

Making Our Webpages Dynamic (aka Making Our Server Do Something)

Up until now, we have been working with static webpages, which honestly could have been done with just writing out HTML and then launching the HTML file in your browser. But a server can do a lot more than that – it can grab variables from the program or a database and put it in the webpage. To show the basics of this, replace the

```
context = { }
```

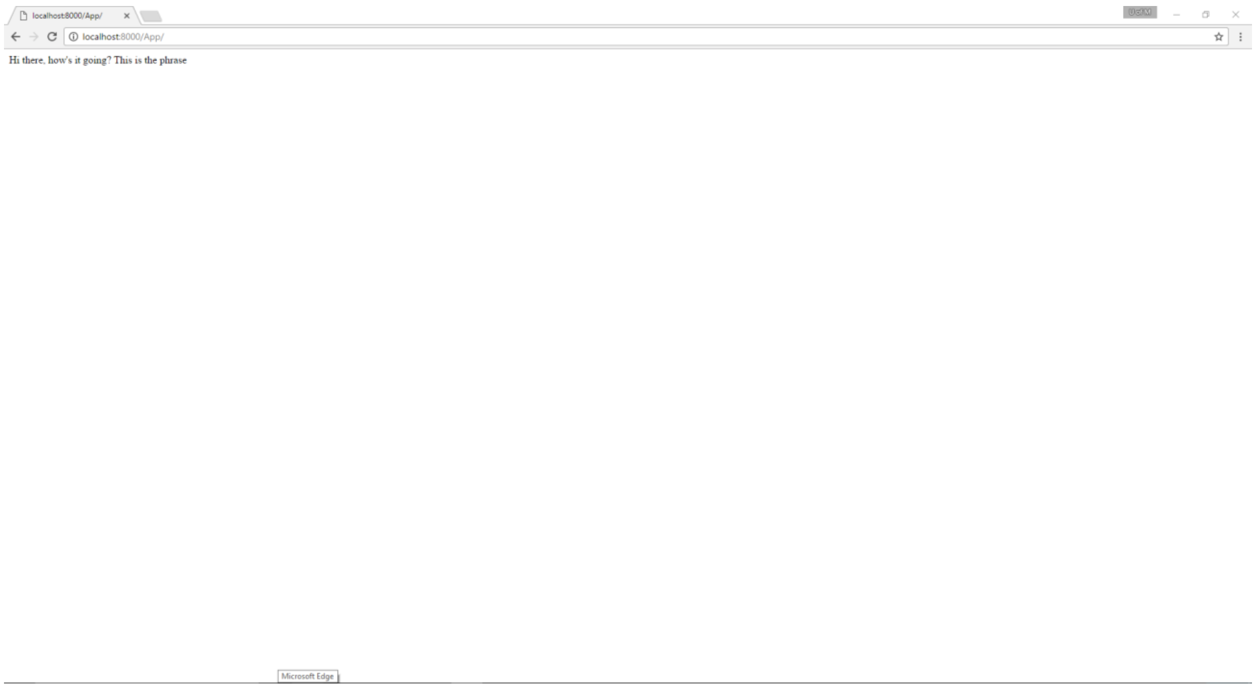
in your Tutorial/TutorialApp/views.py file with

```
context = {"phrase": "Waffles are better than pancakes"}
```

This gets the data in the context and then passes this data to the rendering, put we need to add this data in our HTML file somewhere. So, wherever you want to put this utterly amazing and completely true phrase, add

```
{{ phrase }}
```

This grabs the phrase and shoves it in the HTML file before sending it off to the frontend. Restart the server and reload the webpage and now you should see your phrase in your webpage.



Hi there, how's it going? This is the phrase

Templated Phrase on Website

Storing Data on the Server

The next logical step is to store data on the server so we can retrieve it later. We are going to first work on the backend of this and then make our frontend request the data. To do this, we are going to make a few edits to our Tutorial/TutorialApp/models.py file. Change it to

```
class Post(models.Model):
      text = models.TextField()
      upload_time = models.DateTimeField(auto_now_add = True)
```

If we are to use my spreadsheet analogy, consider each row of a spreadsheet a post. Implicitly, each post has an id number and then we specified it to have some sort of text and an upload time, the latter of which is added automatically when we save it. Now that we have defined what each post is made of, we have to update our database to this new model. In the terminal where you are running the server, issue these commands

```
python manage.py makemigrations

python manage.py migrate
```

These are the commands to update the database. Next, we will update our HTML so that it can send a request to the server. Add the following code to your HTML

```
<form method="POST" action="/App/post/">
	<textarea name="text"></textarea>
	{% csrf_token %}
	<input type="submit" value="Post" />
</form>
```

This make a POST request to the /App/post/ URL whenever the Post button is pressed. You don't need to worry about that {% csrf_token %}, it is just there for security reasons (if you really want to know, you can ask me or go online). So now that we have the website making requests to /App/post, we must update our Tutorial/TutorialApp/urls.py file to also include

```
url(r'^post/$', views.post, name='post'),
```

And because this route sends requests to the post function, we should probably make a post function. Go to Tutorial/TutorialApp/views.py and add the following code to it

```
def post(request):
	post = Post()
	post.text = request.POST['text']
	post.save()
	return redirect('/App/')
```

This creates a new Post object every time post is called, gets the data from the request and puts it in our object, saves it, and then redirects the user to just /App/. We need to add a bit more to make this code work.

```
from .models import *
from django.http import HttpResponse
```

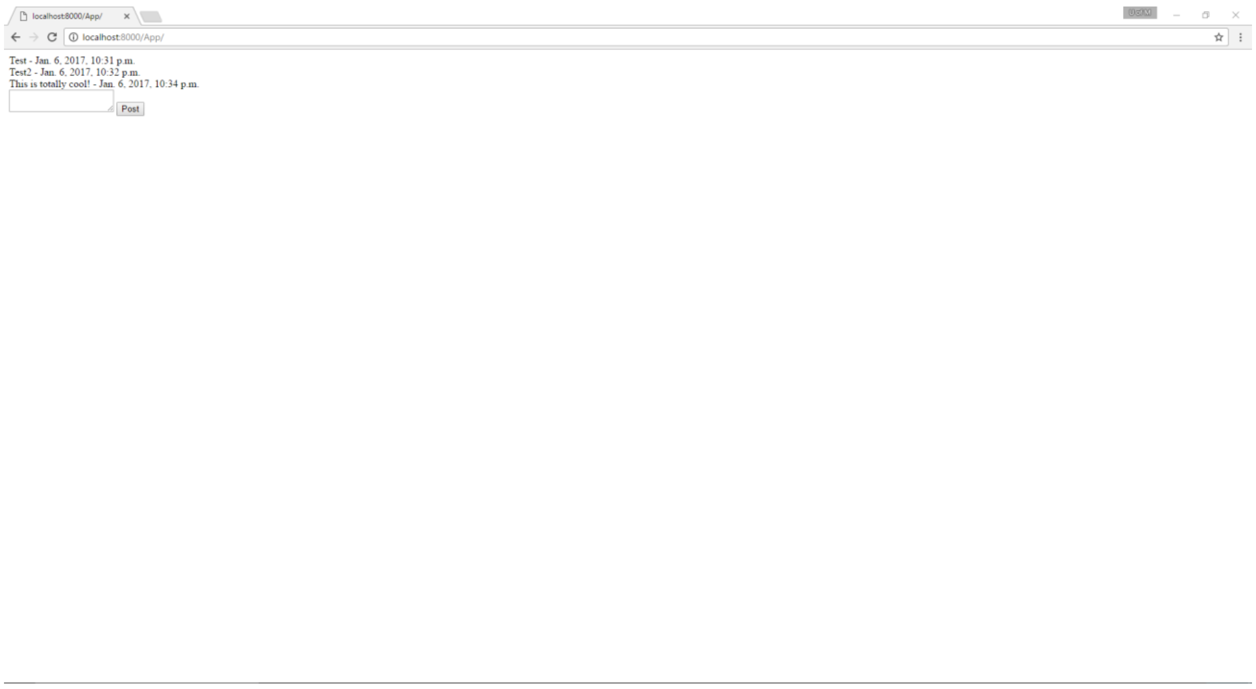
This just allows us to use the functions we just added. We also need to update the index function to show the data that we are storing with each post. Just replace your context variable with

```
posts = Post.objects.all()
context = {
	"posts": posts
}
```

This grabs every post in the database and allows it to be used on the webpage. Finally, we just need to show this data in the HTML file. Go to that index.html file and add

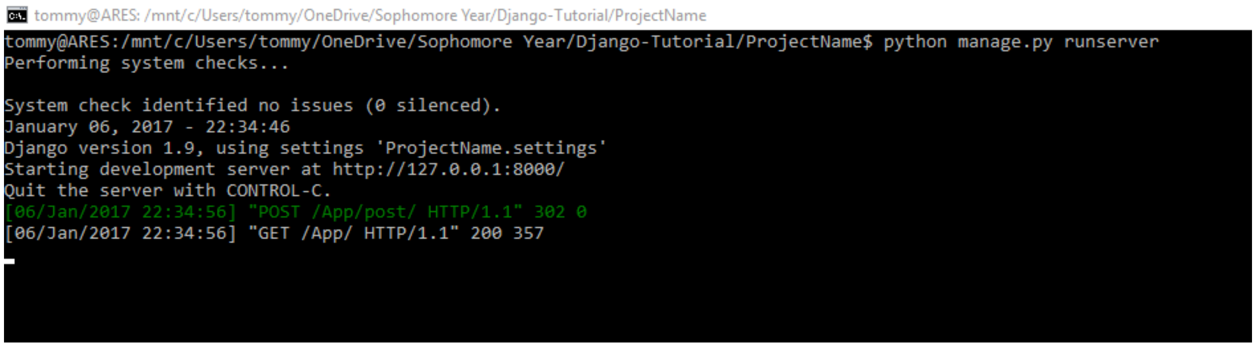
```
{% for post in posts %}
{{ post.text }} - {{ post.upload_time }}<br/>
{% endfor %}
```

This goes over each post that was grabbed from the database and puts the data onto the webpage.



Test - Jan. 6, 2017, 10:31 p.m.
Test2 - Jan. 6, 2017, 10:32 p.m.
This is totally cool! - Jan. 6, 2017, 10:34 p.m.

Final Website



```
tommy@ARES: /mnt/c/Users/tommy/OneDrive/Sophomore Year/Django-Tutorial/ProjectName
tommy@ARES:/mnt/c/Users/tommy/OneDrive/Sophomore Year/Django-Tutorial/ProjectName$ python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).
January 06, 2017 - 22:34:46
Django version 1.9, using settings 'ProjectName.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
[06/Jan/2017 22:34:56] "POST /App/post/ HTTP/1.1" 302 0
[06/Jan/2017 22:34:56] "GET /App/ HTTP/1.1" 200 357
```

Terminal Output (Green text indicates the POST request)

Recap

This goes over all the basics of backend development. There are plenty more topics to cover, like keys to other data (aka a way to link one Excel row to another), Django's built in user system, the Django REST API Framework, and others, but this is what you really need to get started. If you have any more questions, feel free to email me at olivertc@umich.edu or ask me at one of our MiBioSoft meetings.