Faculty of Computer Science, Dalhousie University *8-Mar-2017*

CSCI 2133 — Rapid Programming Techniques for Innovation

Lab 5: Python/Django 5 The Best App is a Finished App

Lab Instructor: Colin Conrad
Location: Shiftkey Labs
Time: Wednesday, 17:30–19:00
Notes copyright: Colin Conrad

Python/Django 5: The Best App is a Finished App

Welcome to Part 5 of the Shiftkey Labs Django Tutorial, the last session of this series. Last week we managed to get out application off the ground, complete with `views`, `urls`, and `templates`. This week we will apply the Twitter API code from Week 1, and configure Django's `forms` feature to handle user requests. By the end of this week, you will have created an app similar to the one at `femto.cs.dal.ca\profileGrabber`.

Step 1: Refresh Git and Investigate

Once again, we should start by updating your tutorial repository using `git pull`. The week5 folder contains a copy of our profiler app as of the end of week 4. We will use this folder as our starting point for this week. If you run the application by using `python manage.py runserver` you can visit the site in your web browser by typing `http://127.0.0.1:8000/profileGrabber/`. You will be able to see our beautiful CSS interface, but the app is not yet functional.

The week5 folder contains a number of important files that you learned about last week. As you recall, the `views.py` file contains the interface logic that renders html contained the `templates` folder. This week, we will be adding new material to the views file to make it work like it should.

Step 2: Preparing the Tweet Profiler Script

The first thing we have to do is bring our old tweet_profiler script into the django API. You can do this by retrieving the material you already created. My script looks like this:

```
#!/usr/bin/env python
# encoding: utf-8

import tweepy #https://github.com/tweepy/tweepy
import time

#Twitter API credentials
consumer_key = ""
consumer_secret = ""
access_key = ""
acces_secret = ""
```

```
# this is used to collect the twitter names
from collections import defaultdict

def get_profile(screen_name):
    auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
    auth.set_access_token(access_key, access_secret)
    api = tweepy.API(auth)
    try:
        user_profile = api.get_user(screen_name)
    except:
        user_profile = "broken"

    return user_profile
```

Write a script like the one above and save it to the `profileGrabber` directory. Do not forget to include your unique API keys in this script, or we will not be able to retrieve profiles. You can test the script by running it in the python shell as per the first week by assing a `print profile.name` statement at the end of the script and running the script. Once you have it working, you can incorporate this function in the `views.py` file.

Return to the `views.py` file and incorporate the Twitter API script. Add a `from . import tweet_profiler` statement to the `views.py` file. This will tell python to import the code from the `tweet_profiler.py` script and use it in the views. Your views file can now read Twitter profiles! This is how you utilize the power of Django in your application. You can likewise import custom scripts that process and handle data using python's libraries, giving you considerable flexibility with data processing and other advanced features.

In our case, we will want to ensure that the variables used in the `views.py` file and rendered in the HTML templates reflect data that are retrieved from the Twitter API. We can do that by making the `profile` variable into the Twitter request. Include the following in the `discover` definition:

```
from . import tweet_profiler

...

def discover(reqest):
    profile = tweet_profiler.get_profile("cd_conrad")
    influencer = "NO"

context = {
    'profile': profile,
    'influencer': influencer,
}

return render(request, 'profileGrabber/discover.html', context)
```

Save the file. If successful, you will see information from my Twitter profile!

Step 3: Create the Handle Form

So far, we have created an application that renders when it receives an HTTP `GET` If we look at the original application however, the `discover` view had two features that utilized `POST` requests. The first feature was the feature that displayed Twitter profile data from the submit form. The second was the feature that added the profile to the database when the + button was pushed. Let's incorporate these.

Django handles web forms in a special forms library. It is a best practice to include your forms in a file called `forms.py`. As with `views.py` the application's forms should be located in the application local folder. Create `profileGrabber/forms.py` and include the following:

```
from django import forms

class HandleForm(forms.Form):
    twitter_handle = forms.CharField(label='Twitter Handle', max_length=100)

class PushForm(forms.Form):
    push_source = forms.CharField(label='Source', max_length=16)
    push_handle = forms.CharField(label='Handle', max_length=100)
```

Save the file. The HandleForm class is designed to process a twitter handle, which is passed through the text field in the HTML document. The PushForm class can push a source and form to the database. As with any front-end features however, Django actually handles forms through the `views.py` file. The Django `request` objects allow us to identify the request using the `request.method`. This way, we can set views logic that is conditional on the method type. We also Let's edit the `views.py` file and edit the file to incorporate the following:

```
...

from .forms import HandleForm

...

def discover(request):
    profile = tweet_profiler.get_profile("cd_conrad")
    influencer = "NO"

    if request.method == 'POST':
        # create a form instance and populate it with data from the request:
        render_handle = HandleForm(request.POST)
        # check whether it's valid:
        if render_handle.is_valid():
            profile = tweet_profiler.get_profile
                    (render_handle.cleaned_data['twitter_handle'])
        else:
            render_handle = HandleForm()
    context = {
        'profile': profile,
        'influencer': influencer,
    }

    return render(request, 'profileGrabber/discover.html', context)
```

Save the file. You should now have the ability to add any given Twitter handle to the text field and submit it to the Twitter api. Give it a try on a familiar such as "shiftkeylabs" and see what happens.

Step 4: Prepare the Push Form

The handle form is working, however the push form is not working yet. The view is currently structured to process GET and POST requests differently. However, we will actually have two forms on this view, so we will need to use a second `elif` statement to define the logic for the Push Form. Given that the purpose of this form is to push data to the database, .

```
if request.method == 'POST':
        # create a form instance and populate it with data from the request:
        render_handle = HandleForm(request.POST)
        render_push = PushForm(request.POST)
        # check whether it's valid:
        if render_handle.is_valid():
            profile = tweet_profiler.get_profile
                                (render_handle.cleaned_data['twitter_handle'])
            render_push = PushForm()
        elif render_push.is_valid():
            profile = tweet_profiler.get_profile
                                (render_push.cleaned_data['push_handle'])
            try:
                s = SavedUser(
                    source = render_push.cleaned_data['push_source'],
                    handle = profile.screen_name,
                    name = profile.name,
                    followers = profile.followers_count,
                    following = profile.friends_count,
                    location = profile.location,
                    influencer = influencer,
                    description = profile.description,
                    )
                s.save()
            except:
                s = ""
            render_handle = HandleForm()
        else:
            render_handle = HandleForm()
            render_push = PushForm()
```

These data features bridge the API data and the Django database. Save your work and open the `discover.html` file. Click on the + button to try adding the profile to the database. Open the `/admin` interface to view whether it has been added to the database. If you cannot enter the `admin` utility remember you can use `createsuperuser` to create a valid user and password. You should find the added profiles there!

Step 5: Add Application Logic

The application is almost complete! Though the forms and `profile` variable is processing correctly, t he `views.py` logic currently does not process the `influencer` feature the way it should. Twitter influencers refer to people who influence other Twitter users and their behaviours. Influencers can be determined using any number of algorithms, especially those identified using machine learning. We will do something much simpler however, and get our app to identify users who have over 500 followers.

This is trivial. Add a conditional statement that investigates the number of Twitter followers and changes influencer to "YES" if there are more than 500 followers. The code should go just before the context:

```
...

if profile.followers_count > 500:
        influencer = "YES"
    context = {
...
```

Save the `views.py` file. Congratulations! You have completed a Twitter application that uses Django and the Twitter API to process Twitter profiles! This demonstrates the basics of how to build web applications using APIs and Python. Hopefully this will help you in your future careers.

Step 6: Next Steps

Though the app is close to where you would like it to be, there are a few things that can be done to make it better. I will list them below:

– Change the `urls` master file to render the `profileGrabber/urls.py` in the `/` directory
– Add a view for `collection.html` similar to that in the original app.
– Create custom logic for the `influencer` function. Perhaps this is derived from machine learning?

The sky is the limit, really. There is a lot you can do with this application. You challenge this week is not just to configure the collection view to handle Tweets as you did in previous weeks (it should be a given that you include this), but to add a more robust algorithm for the influencer function. The algorithm can consider numbers of user tweets, ratios of followers to friends, or anything else you think might be a good measure of whether someone is an influencer. If you need some ideas, visit `https://blog.hootsuite.com/influencer-marketing/`. Good luck!