Faculty of Computer Science, Dalhousie University *1-Mar-2017*

CSCI 2133 — Rapid Programming Techniques for Innovation

Lab 4: Python/Django 4 There is a Template for Everyone

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

Python/Django 4: There is a Template for Everyone

Welcome to Part 4 of the Shiftkey Labs Django Tutorial. Last week we started using Django and learned about models, built the lovely Justin Beiber app, and got started on our profiler. This week we will learn more about views and forms, and will work on our original application through the entire session. We will also start using the Django Templates feature, which will make it easier to manage our application. For more information about these features, check out `docs.djangoproject.com`.

Step 1: Refresh Git and Investigate

The first thing we should do is update your tutorial repository using `git pull`. The week4 folder contains a copy of our profiler app as of the end of the last tutorial. We will use this folder as our starting point. 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/`. We have a "hello world!", which is exactly what `views.py` asks Django to do.

You probably noticed that Django gave us a warning about our migrations. This warning appears because though we have models, they are not yet made into a database. Migrations are Django's mechanism for moving the `models.py` code into a database. Let's try investigating the problem through our handy admin panel. Start by making the migrations with `python manage.py makemigrations` and `python manage.py migrate`. You can then use `python manage.py createsuperuser` to access the administrator backend, if so desired. This will bring us to where we were last week. We need to fix this to work with out models and make it accept user data-entered data.

Step 2: Preparing Views

Currently we have the ability to save Twitter profiles in our database, but not to render them in the browser. Let's change that. Open the `profileGrabber/views.py` file and prepare the views for the relevant web pages. As you recall, our app has three views: `index` which displays the homepage, `discover` which allows users to look at Twitter profiles, and `collections` which displays the collected profiles. We thus need to define our `views.py` file to reflect these views. Use the following code:

```
from django.shortcuts import render, get_object_or_404

from .models import SavedUser

def index(request):
```

```
    context = {}
    return render(request, 'profileGrabber/index.html', context)

def discover(request):
    profile = "cd_conrad"
    influencer = "NO"
    context = {
        'profile': profile,
        'influencer': influencer,
    }

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

def collection(request):
    saved_users = SavedUser.objects.all()

    context = {
        'saved_users': saved_users,
    }

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

Unlike the previous `profileGrabber/views.py` file, this views file uses Django's render shortcut. Render makes it easier to work with the views logic, and automatically renders python variables and objects as HTML code when defined in the `context`. This is a handy feature for quickly producing views with complex logic.

Try running the application using `python manage.py runserver`. Django will return an error. Now that we are using the models in the views, there is an additional step, which tells Django how the two pieces of code are connected. Create a `profileGrabber/apps.py` file and copy the following code:

```
from __future__ import unicode_literals

from django.apps import AppConfig


class ProfilegrabberConfig(AppConfig):
    name = 'profileGrabber'
```

This code tells Django that this is an app that uses models. Now, connect this change to your core Django logic by changing the settings. Open `profiler/settings.py` and change the following in the settings file:

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

This will resolve the server error. Try running server again. You will see the debug interface, and that Django is missing a template file. At least it's not a server error.

Step 3: Configure the Templates

Return to the `profileGrabber` folder. As we saw earlier, we have a folder containing our html views, but it has not been formatted to meet Django's template files. I have configured the views to request template files, but there is nothing yet available for Django to retrieve.

To fix this, create a new folder called `templates` and a folder within that called `profileGrabber`. By default, Django is configured to look in the `<app>/templates` folder for the html templates that are rendered in the views logic. Copy your html files into this folder such that you have something like:

```
profiler
-profileGrabber
--templates
---discover.html
---index.html
...
```

Try running the Django server. You will see that the html we defined in `templates/index.html` renders and we have something that looks like our app! This is because we defined `'index.html'` in the render function of our views. Success! However, if we click on 'Discover' we realize that the Discover file is not being recognized properly. Return to `profileGrabber/urls.py` to define the new urls.

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    url('discover.html', views.discover, name='discover'),
]
```

Now if we open the `discover.html` file, the view will render properly. For real this time... success!

Step 4: Connect the Templates to the Views

Our front-end may be loading, but we still need to connect the views logic. Let's edit the `discover.py` file to fix this. In Django, whenever we want to get a template to talk to a view, we encase the desired object or variable in brackets. For instance, if we wanted to render the value of `profile`, we would do so by denoting profile somewhere in the template document. Let's try this in our document by making the following changes:

```
<div class="row">
  <div class="col m3">
    <h4>{{ profile.name }}</h4>
    <span style="font-weight:bold;">{{ profile.screen_name }}</span>
    <span>{{ profile.location }}</span>
  </div>
  <div class="col m3">
```

```
    <h4>Influencer: {{ influencer }}</h4>
    <!-- Modal Trigger -->
    <a class="waves-effect waves-light btn green" href="#modal1">What's this?</a>

    <!-- Modal Structure -->
    <div id="modal1" class="modal">
      <div class="modal-content">
        <h4>Influencers</h4>
        <p>Influencers are people who start trends on
                Twitter. Typically these users have a large number of

followers, and a large number of people actually read what they

post. We can create any number of algorithms to determine influencers,

but we kept it simple for this tutorial: people with over 500

followers.</p>

      </div>
      <div class="modal-footer">
        <a href="#!" class=" modal-action modal-close waves-effect waves-green btn-flat">
      </div>
    </div>
  </div>
  <div class="col m3">
    <h4>Followers: {{ profile.followers_count }}</h4>
    <span style="font-weight:bold;">Following:</span>
    <span>{{ profile.friends_count }}</span>
  </div>
  <div class="col m3">
    <form action="" method="post">
      {% csrf_token %}
      <input type="hidden" name="push_source" value="Web">
      <input type="hidden" name="push_handle" value="{{ profile.screen_name }}">
      <button class="btn-floating btn-large waves-effect waves-light red" type="submit" n
          <i class="material-icons">add</i>
      </button>
    </form>
  </div>
</div>
<div class="row">
  <div class="col m12">
    <blockquote>
      <p>{{ profile.description }}</p>
      <footer>{{ profile.screen_name }}<cite title="Source Title"> Profile Summary </cite
    </blockquote>
  </div>
</div>
```

Save the file and open it in your browser. It currently renders blank items. This is a good thing because we haven't

yet configured the API to retrieve items, but this is a task for next week.

Step 5: Configure Static

The app looks nice, but it is not currently rendering images and static files. Where templates and views are fundamentally dynamic (see the subsequent step), it is often desirable to keep things like photos and stylesheets in a single static folder that doesn't change between template folders. Fortunately, Django has this feature, and it is super-easy to use.

Move the `assets` folder to `profileGrabber` and rename it `static`.

Step 5: Configure Reusable Templates

Perhaps the greatest advantage of using a web framework is that it can decrease the amount of code you have to write. Imagine if you had to create the same code over and over again, even when large parts of it is identical between views. Django is designed to use reusable templates to store things like headers and footers, which makes your site more consistent and a lot less work.

This flexibility of course comes at the expense of keeping track of everything. Currently, our application is not rendering images and static files. It is often desirable to keep things like photos and stylesheets in a single static folder that doesn't change between template folders. Fortunately, Django has this feature, and it is super-easy to use. Move the `assets` folder to and rename it `static`. Django will automatically recognize verb,/static/, folder when we incorporate them into our templates.

For simplicity, I provided you with a simple template file. Rename `template.txt` to `template.html` and take a look in your editor.

This code does a number of things that will make our lives easier. The `{\% load static \%}` tag uses Django-reserved words (`load` and `static`) to track the static folder and load the files therein. This will allow the `twitter.svg` file to render properly in our homepage. The header content of our template is subsequently defined by the html/css in the file. It should look familiar.

The `{% block content %} {% endblock %}` tags define where the template stops and where the child of the template begins. In our case, we would like to define our `discover.html` and `index.html` files to incorporate this template. Open the discover template and remove the duplicate code, and replace it with the following:

```
{% extends "profileGrabber/template.html" %}

{% block content %}

...[the non-header content]


{% endblock %}
```

This will make the view depend on the ,template.html,. Do the same for `index.html` and save. Your site should render perfectly! Congratulations! You now know how the Django forms and templates work and have come to the end of this week's tutorial.

If you are ambitious, or are in CSCI 2133, I have a another challenge for you. I wonder whether you can create a `collections.html` view and template that renders the tweets that you prepared in your database last week? The view should render html that you drafted in week 2, and should use the master `template.html` file that we just created. Good luck!