CMPUT 404 Lab 4

26-27th January 2016

OVERVIEW

Build a simple Django website. Understand the fundamentals of Django's MVC architecture using the built in models and views framework. Understand how to write basic unit tests.

STEPS

1. Create a virtual environment and install Django 1.8

```
cd path/to/your/project
virtualenv venv
source venv/bin/activate
pip install Django==1.8
```

2. Initialize a new Django project, called **mysite**

```
django-admin startproject mysite
cd mysite
python manage.py runserver # Lab Question 1
```

Question 1:

After starting a brand new django application and running the runserver command, what does the browser show you?

3. Create the a new application within **mysite** called **polls**.

```
python manage.py startapp polls
```

4. Modify the polls/views.py file to look like the following.

```
from django.http import HttpResponse
def index(request):
    return HttpResponse("Hello, world. You're at the polls index.")
```

5. Create a file at polls/urls.py with the following code.

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

6. Within the **mysite/urls.py**, include the following code.

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

7. Run the Django project with the runserver command.

```
python manage.py runserver
```

Question 2:

After creating the first view within polls, what does the browser show you when navigating to / and to /polls respectively?

8. Time to create our first models. Open up the **mysite/settings.py** file and ensure that the default database is set to sqlite3.

```
# Database
# https://docs.djangoproject.com/en/1.8/ref/settings/#databases
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

9. Within the **polls/models.py**, include the following code.

```
from django.db import models
from django.utils.encoding import python_2_unicode_compatible

@python_2_unicode_compatible
class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')
      def __str__(self):
        return self.question_text

@python_2_unicode_compatible
class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
      def __str__(self):
        return self.choice_text
```

10. Edit the **mysite/settings.py** file again, change the *INSTALLED_APPS* setting includes '*polls*':

```
INSTALLED_APPS = [
    'polls',
    'django.contrib.admin',
      ...
]
```

11. Run the migrations for the Polls app.

```
python manage.py makemigrations polls
```

12. Run the *migrate* command to add the model to the database.

```
python manage.py migrate
```

Question 3:

What is a Django migration and why do we need them?

13. Create a superuser to access the model from the django admin backend.

```
python manage.py createsuperuser
```

14. Make the polls app modifiable in the admin by editing the **polls/admin.py** file to be the following:

```
from django.contrib import admin
from .models import Choice, Question
admin.site.register(Choice)
admin.site.register(Question)
```

15. Start the development server and go to /admin/ on your local domain.

```
python manage.py runserver
```

Question 4:

What do you see after you log into the Django Administration site? From a high level, how do you get custom models to appear in the Django admin?

16. Add some additional views to the **polls/views.py** file. Include the following methods:

```
def detail(request, question_id):
    return HttpResponse("You're looking at question %s." % question_id)
def results(request, question_id):
    response = "You're looking at the results of question %s."
    return HttpResponse(response % question_id)
def vote(request, question_id):
    return HttpResponse("You're voting on question %s." % question_id)
```

17. With the above views added, add those views to the **polls/urls.py** file:

```
from django.conf.urls import url
from . import views
urlpatterns = [
    # ex: /polls/
    url(r'^$', views.index, name='index'),
    # ex: /polls/5/
    url(r'^(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
    # ex: /polls/5/results/
    url(r'^(?P<question_id>[0-9]+)/results/$', views.results, name='results'),
    # ex: /polls/5/vote/
    url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
]
```

Question 5:

What do you see when you go to */polls/38/* in your browser? What about */polls/38/results* and */polls/38/vote*? What happens when you don't put a number, and instead use a string? How would you modify the urls.py file to allow arbitrary alphabetic characters?

18. Modify the **polls/views.py** file, specifically the **index** view to be the following:

```
from django.http import HttpResponse
from .models import Question
def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    output = ', '.join([p.question_text for p in latest_question_list])
    return HttpResponse(output)
# Leave the rest of the views (detail, results, vote) unchanged
```

19. Create an empty directory named **templates** within the **polls**. Then, create another directory named **polls** within the **templates** directory. Lastly, create a file called **index.html** within the second polls directory.

```
mkdir -p polls/templates/polls
touch polls/templates/polls/index.html
```

20. Within the newly created empty **polls/templates/polls/index.html** file, write the following:

```
{% if latest_question_list %}
    <ul>
    {% for question in latest_question_list %}
        <li><a href="/polls/{{ question.id }}/">{{ question.question_text
}}</a></li>
    {% endfor %}
    </ul>
{% else %}
    <p>No polls are available.</p>
{% endif %}
```

21. Update the **polls/views.py** file, such that the **index** view now uses this new template.

```
from django.shortcuts import render
from .models import Question
def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    context = {'latest_question_list': latest_question_list}
    return render(request, 'polls/index.html', context)
```

22. Add a new template file for the poll details view.

```
touch polls/templates/polls/detail.html
```

23. For the newly created detail template in **polls/templates/polls/detail.html**, update the content with the template tag for our question:

```
<h1>{{ question.question_text }}</h1>
<ul>
{% for choice in question.choice_set.all %}
    <li>{{ choice.choice_text }}</li>
{% endfor %}
</ul>
```

24. Update the **polls/views.py** file, such that the **details** view now uses a template, and can throw a 404 error if there is no question matching the url.

```
from django.shortcuts import get_object_or_404, render
from .models import Question
# ...
def detail(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    return render(request, 'polls/detail.html', {'question': question})
```

25. Remove the hardcoded urls that we specified in the **polls/templates/polls/index.html** file and replace it with a template tag, referencing our url.

```
<li><a href="{% url 'polls:detail' question.id %}">{{ question.question_text
}}</a></li>
```

Question 6:

Why is it a bad idea to hardcode urls into the templates?

26. Update the **polls/templates/polls/detail.html** file to match the following:

```
<h1>{{ question.question_text }}</h1>
{% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}
<form action="{% url 'polls:vote' question.id %}" method="post">
{% csrf_token %}
{% for choice in question.choice_set.all %}
    <input type="radio" name="choice" id="choice{{ forloop.counter }}"
value="{{ choice.id }}" />
    <label for="choice{{ forloop.counter }}">{{ choice.choice_text
}}</label><br />
{% endfor %}
<input type="submit" value="Vote" />
</form>
```

27. Update the **polls/views.py** file to handle the above form submission, allowing for voting:

```
from django.shortcuts import get_object_or_404, render
from django.http import HttpResponseRedirect, HttpResponse
from django.core.urlresolvers import reverse

from .models import Choice, Question
# ...
def vote(request, question_id):
    p = get_object_or_404(Question, pk=question_id)
    try:
        selected_choice = p.choice_set.get(pk=request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
        # Redisplay the question voting form.
        return render(request, 'polls/detail.html', {
            'question': p,
            'error_message': "You didn't select a choice.",
        })
    else:
```

```
        selected_choice.votes += 1
        selected_choice.save()
        # Always return an HttpResponseRedirect after successfully dealing
        # with POST data. This prevents data from being posted twice if a
        # user hits the Back button.
        return HttpResponseRedirect(reverse('polls:results', args=(p.id,)))
```

28. Create a template file for the poll results at **polls/templates/polls/results.html** with the following:

```
<h1>{{ question.question_text }}</h1>
<ul>
{% for choice in question.choice_set.all %}
    <li>{{ choice.choice_text }} -- {{ choice.votes }} vote{{
choice.votes|pluralize }}</li>
{% endfor %}
</ul>
<a href="{% url 'polls:detail' question.id %}">Vote again?</a>
```

29. Also, modify the **polls/views.py** file to pass the required data through the context to the results view:

```
from django.shortcuts import get_object_or_404, render
def results(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    return render(request, 'polls/results.html', {'question': question})
```

30. Run your application. Using the admin interface, create a question, then create multiple choices for your question. Navigate back to the /polls page and attempt to use your application. The following steps will refactor the above code to use Django's generic views.

31. Open up the **polls/urls.py** file and modify it such that it looks like the following:

```
from django.conf.urls import url
from . import views
urlpatterns = [
    url(r'^$', views.IndexView.as_view(), name='index'),
    url(r'^(?P<pk>[0-9]+)/$', views.DetailView.as_view(), name='detail'),
    url(r'^(?P<pk>[0-9]+)/results/$', views.ResultsView.as_view(),
name='results'),
    url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
]
```

32. Modify the **polls/views.py** to change the **index, results, detail** view to match the following:

```
from django.shortcuts import get_object_or_404, render
from django.http import HttpResponseRedirect
from django.core.urlresolvers import reverse
from django.views import generic
from .models import Choice, Question
class IndexView(generic.ListView):
    template_name = 'polls/index.html'
    context_object_name = 'latest_question_list'
    def get_queryset(self):
        """Return the last five published questions."""
        return Question.objects.order_by('-pub_date')[:5]

class DetailView(generic.DetailView):
    model = Question
    template_name = 'polls/detail.html'

class ResultsView(generic.DetailView):
    model = Question
    template_name = 'polls/results.html'

def vote(request, question_id):
    ... # same as above
```

Question 7:

What are the benefits of using Django's generic views over writing views 'the hard way'? When should you use a generic view and when shouldn't you use a generic view?

33. Let's add a method to our Question model. Modify our **polls/models.py** file such that there's a **was_published_recently()** method:

```
import datetime
from django.utils import timezone
# ...

class Question(models.Model):
    # ...
    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)
```

34. Lets test this newly created method out using Django's shell feature.

```
python manage.py shell --plain
```

35. Observe the output of the following code:

```
>>> import datetime
>>> from django.utils import timezone
>>> from polls.models import Question
>>> # create a Question that was published 60 days ago
>>> past_question = Question(pub_date=timezone.now() -
datetime.timedelta(days=60))
>>> past_question.was_published_recently()
False
>>> # create a Question instance with pub_date 30 days in the future
>>> future_question = Question(pub_date=timezone.now() +
datetime.timedelta(days=30))
>>> # was it published recently?
>>> future_question.was_published_recently()
True
```

36. Obviously that is a bug. Let's write a unit test to catch this issue. Open up the **polls/tests.py** and modify the file contents:

```
import datetime

from django.utils import timezone
from django.test import TestCase

from .models import Question

class QuestionMethodTests(TestCase):
    def test_was_published_recently_with_future_question(self):
        """
        was_published_recently() should return False for questions whose
        pub_date is in the future.
        """
        time = timezone.now() + datetime.timedelta(days=30)
        future_question = Question(pub_date=time)
        self.assertEqual(future_question.was_published_recently(), False)
```

37. Lets run the test for the application now. The test should fail, as the code in our model have not changed.

```
python manage.py test polls
# or if testing all applications
python manage.py test
```

38. Lets fix this bug. Modify the **polls/models.py** file and change the **was_published_recently()** method:

```
def was_published_recently(self):
    now = timezone.now()
    return now - datetime.timedelta(days=1) <= self.pub_date <= now
```

39. Run the tests again. The one test should pass.

Question 8:

Why is it a good idea to write unit tests for our code?

SUMMARY

For additional information, check out the existing Django documentation at https://docs.djangoproject.com/en/1.8/ as well as the Django tutorial https://docs.djangoproject.com/en/1.8/intro/.