Django Beginner's Tutorial

Parts 5 and 6

`https://docs.djangoproject.com/en/2.2/intro/tutorial05/`
`https://docs.djangoproject.com/en/2.2/intro/tutorial06/`

- Automated testing
- Exposing and fixing a bug in the polls app
- Testing views
- (Very) brief introduction to CSS
- Static files

1

What are automated tests?

- Tests are simple routines that check the operation of your code
- Testing operates at different levels
- With automated tests, the testing work is done by the system
- Create a set of tests once, and as you make changes to your app, you can check that your code still works as you originally intended

2

Why create tests?

- Tests save you time
- Tests don't just identify problems, they help prevent them
- Tests make your code more attractive (to others)
- Tests help teams work together

3

Test-driven development

- Write a test before you actually write the code!
- Seems counter-intuitive, *but*:
  - Similar to what many people do already
  - Describe a problem, create some code to solve it
  - TDD formalises the problem in a Python test case
- Easier to write tests as you go along rather than to add them later

4

Exposing a bug

- The polls app actually has a bug already!
- `Question.was_published_recently()` returns True if the Question was published within the last day (which is correct) but also if the Question's `pub_date` field is in the future (which isn't)
- Create a question whose publication date is in the future:

```
>>> import datetime
>>> from django.utils import timezone
>>> from polls.models import Question
>>> future_question = Question(pub_date=timezone.now() +
                       datetime.timedelta(days=30))
>>> # was it published recently?
>>> future_question.was_published_recently()
True
```

5

Writing our first test

- Turn the shell commands into an automated test
- Add the following to `tests.py`:

```
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.assertIs(future_question.was_published_recently(),
                      False)
```

6

Running tests

- In the terminal, we can run our test:

```
python manage.py test polls
```

- We'll see the following output

```
Creating test database for alias 'default'...
F
======================================================================
FAIL: test_was_published_recently_with_future_question
(polls.tests.QuestionMethodTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/path/to/mysite/polls/tests.py", line 16, in
  test_was_published_recently_with_future_question
  self.assertIs(future_question.was_published_recently(),
  False)
AssertionError: True is not False
----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
Destroying test database for alias 'default'...:
```

7

What actually happened

- `python manage.py test polls` looked for tests in the polls application
- it found a subclass of the `django.test.TestCase` class
- it created a special database for the purpose of testing
- it looked for test methods - ones whose names begin with test
- in `test_was_published_recently_with_future_question` it created a `Question` instance whose `pub_date` field is 30 days in the future
- using the `assertIs()` method, it discovered that its `was_published_recently()` returns `True`, though we wanted it to return `False`
- The test informs us which test failed and even the line on which the failure occurred

8

Fixing the bug

- Add the following code to `models.py`:

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

- Run the test again:

```
Creating test database for alias 'default'...
.
----------------------------------------------------
Ran 1 test in 0.001s

OK
Destroying test database for alias 'default'...
```

9

More comprehensive tests

- Add the following code to `tests.py`:

```
def test_was_published_recently_old_question(self):
    # was_published_recently() should return False for
    # questions whose pub_date is older than 1 day.
    time = timezone.now() - datetime.timedelta(days=30)
    old_question = Question(pub_date=time)
    self.assertIs(old_question.was_published_recently(),
                  False)

def test_was_published_recently_recent_question(self):
    # was_published_recently() should return True for
    # questions whose pub_date is within the last day.
    time = timezone.now() - datetime.timedelta(hours=1)
    recent_question = Question(pub_date=time)
    self.assertIs(recent_question.
                  was_published_recently(), True)
```

10

Testing views

- Django provides a test `Client` to simulate a user interacting with the code at the view level
- We can use `Client` in `tests.py` or even in the shell

```
>>> from django.test.utils import setup_test_environment
>>> setup_test_environment()
```

- `setup_test_environment()` installs a template renderer which will allow us to examine some additional attributes on responses such as `response.context`
  - Unlike previous tests, *does not* setup test database
- Next we need to import the test client class

```
>>> from django.test import Client
>>> # create an instance of the client for our use
>>> client = Client()
```

11

Using the Django test client from the shell

```
>>> # get a response from '/'
>>> response = client.get('/')
Not Found: /
>>> # we should expect a 404 from that address
>>> response.status_code
404
>>> # on the other hand we should expect to find
>>> # something at '/polls/'
>>> # we'll use 'reverse()' rather than a hardcoded URL
>>> from django.urls import reverse
>>> response = client.get(reverse('polls:index'))
>>> response.status_code
200
>>> response.content
b'\n    <ul>\n    \n
<li><a href="/polls/1/">What&#39;s up?</a></li>...
>>> response.context['latest_question_list']
<QuerySet [<Question: What's up?>]>...
```

12

Improving the view

- The list of polls shows polls that aren't published yet – we will modify this
- Recall the `index` view from `views.py`:

```
def index(request):
    latest_questions = Question.objects.order_by
                       ('-pub_date')[:3]
    context = {'latest_question_list': latest_questions}
    return render(request, 'polls/index.html', context)
```

- We need to ensure that when we output the list of questions we also check the date using `timezone.now()`

```
from django.utils import timezone
```


Improving the view (cont)

- Add a filter to `Question.objects` as follows:

```
def index(request):
    latest_questions = Question.objects.filter
                       (pub_date__lte=timezone.now()).
                        order_by('-pub_date')[:3]
    context = {'latest_question_list': latest_questions}
    return render(request, 'polls/index.html', context)
```

- We now need to add appropriate tests for the new view
- Add the following to tests.py:

```
from django.urls import reverse
...
```

Testing the new view

```
def create_question(question_text, days):
  # creates a question given given question_text and
  # published the given number of days offset to now
  # (negative for past, positive for future)
  time = timezone.now() + datetime.timedelta(days=days)
  return Question.objects.create(
            question_text=question_text, pub_date=time)

class QuestionViewTests(TestCase):
  def test_index_view_with_no_questions(self):
    # If no questions exist, an appropriate message
    # should be displayed
    response = self.client.get(reverse('polls:index'))
    self.assertEqual(response.status_code, 200)
    self.assertContains(response, "No polls are
                                    available.")
    self.assertQuerysetEqual(response.context
                  ['latest_question_list'], [])
```

15

Testing the new view (2)

```
def test_index_view_with_a_past_question(self):
  # Only past questions published should be displayed
  create_question(question_text="Past question.",
                  days=-30)
  response = self.client.get(reverse('polls:index'))
  self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question.>'])

def test_index_view_with_a_future_question(self):
  # Future questions should not be displayed
  create_question(question_text="Future question.",
                  days=30)
  response = self.client.get(reverse('polls:index'))
  self.assertContains(response, "No polls are
                                  available.")
  self.assertQuerysetEqual(
        response.context['latest_question_list'], [])
```

16

Testing the new view (3)

```
def test_index_view_with_future_question_
    and_past_question(self):
  # Even if both past and future questions exist,
  # only past questions should be displayed
  create_question(question_text="Past question.",
                  days=-30)
  create_question(question_text="Future question.",
                  days=30)
  response = self.client.get(reverse('polls:index'))
  self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question.>'])
```

17

Testing the new view (4)

```
def test_index_view_with_two_past_questions(self):
  # The questions index page may display multiple
  # questions.
  create_question(question_text="Past question 1.",
                  days=-30)
  create_question(question_text="Past question 2.",
                  days=-5)
  response = self.client.get(reverse('polls:index'))
  self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question 2.>',
             '<Question: Past question 1.>'])
```

18

Testing the detail view

- Even though future questions don't appear in the index, users can still reach them if they know / guess the URL

```
def detail(request, question_id):
    question = get_object_or_404(Question,
                                 pk=question_id)
    return render(request, 'polls/detail.html',
                  {'question': question})
```

- We need to add a similar constraint in the detail view:

```
def detail(request, question_id):
    question = get_object_or_404(Question.
         objects.filter(pub_date__lte=timezone.now()),
         pk=question_id)
    return render(request, 'polls/detail.html',
                  {'question': question})
```

19

Testing the new detail view

- We will add some tests to check that a Question whose `pub_date` is in the past can be displayed, and that one with a `pub_date` in the future is not

```
class QuestionIndexDetailTests(TestCase):
  def test_detail_view_with_a_future_question(self):
    # The detail view of a question with a pub_date in
    # the future should return a 404 not found
    future_question = create_question(
      question_text='Future question.', days=5)
    url = reverse('polls:detail',
          args=(future_question.id,))
    response = self.client.get(url)
    self.assertEqual(response.status_code, 404)
```

20

Testing the new detail view (2)

```
def test_detail_view_with_a_past_question(self):
    # The detail view of a question with a pub_date in
    # the past should display the question's text
    past_question = create_question(
                question_text='Past Question.', days=-5)
    url = reverse('polls:detail',
                args=(past_question.id,))
    response = self.client.get(url)
    self.assertContains(response,
                past_question.question_text)
```

21

Ideas for more tests

- We ought to add a similar use of the `filter` function to the results view and create a new test class for that view
- Similar to what we have just created (leading to repetition)
- We could also improve our application in other ways, adding tests along the way, e.g.,
  - Ensure that Questions cannot be published without Choices
    - Our views could check for this, and exclude such Questions
    - Our tests would create a Question without Choices and then test that it's not published
  - Perhaps logged-in admin users should be allowed to see unpublished Questions, but not ordinary visitors
    - Again: whatever needs to be added to the software to accomplish this should be accompanied by a test
- Is your code suffering from test bloat?

22

Testing: more is better

- You can't have too many tests
- Sometimes tests need to be updated
  - E.g., if we amend our views so that only Questions with Choices are published
- Redundant tests don't matter
- Tests should be arranged so they are kept manageable. There should be:
  - a separate TestClass for each model or view
  - a separate test method for each set of conditions you want to test
  - intuitive test method names that should describe their function

23

Static files

- These correspond to images, JavaScript or CSS
- We will now add a stylesheet (via a CSS file) and an image
- Inside `static` folder, add subfolder called `polls`
- Create a file called `style.css` inside `polls` folder
- Add the following code to style.css:

```
li a {
    color: green;
}
```

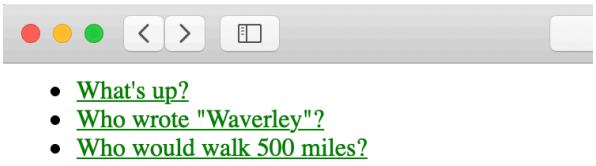
24

Loading our stylesheet

- Add the following code to index.html:

```
{% load static %}

<link rel="stylesheet" type="text/css"
 href="{% static 'polls/style.css' %}" />
```



25

Adding a background image

- Create a subfolder for images. Create a folder called `images` as a subfolder of `static/polls`

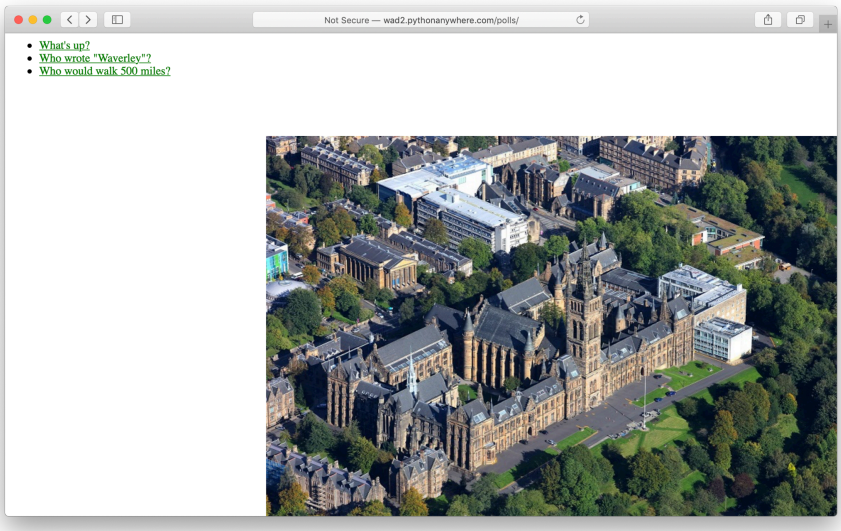
- Put a background image called `background.jpg` in this folder

- Add the following to your stylesheet:

```
body {
    background: white
    url("images/background.jpg")
    no-repeat right bottom;
}
```

26

End-result



27

End of Django lectures!

- These lectures skipped some parts of tutorials
  - Feel free to carry on
  - `https://docs.djangoproject.com/en/2.2/intro/`
- Polls is not assessed
- Don't submit Polls along with/instead of Rango for your assessed exercise
  - To submit the exercise, you'll provide us with the GitHub url FOR YOUR RANGO PROJECT
- New topics from next week – system architectures

28