TDD Django Tutorial

By Harry Percival

@hjwp

http://www.tdd-django-tutorial.com

And look out for my book!

Part 0 - Pre-requisites

- Git and my repo checked out (`git clone https://github.com/hjwp/Test-Driven-Django-Tutorial`)
- Firefox
- Python 2.7
- Django 1.4 or 1.5
- Selenium (latest version)

If you haven't got the pre-requisites, you'll have to follow along with someone else's PC. That's OK, pairing is both fun and edumecational, plus my whole tutorial is available online so you can go through the whole thing in your own time later.

Checking you have the pre-requisites installed

Git:

`cd` to wherever you have my repo checked out

```
git checkout pycon_2013
dir pycon_2013_handout.asciidoc
```

Django:

```
cd /tmp    # or wherever
django-admin.py startproject delete_me
cd delete_me
python manage.py runserver
```

And check you can see the Django "It worked" page

on http://localhost:8000

Selenium

In a script or a Python shell:

```
from selenium import webdriver
browser = webdriver.Firefox()
browser.get('http://www.google.com')
print browser.title
browser.quit()
```

should print "`Google`"

`manage.py tartproject` should give you a folder structure like this:

```
.
`-- mysite
    |-- manage.py
    `-- mysite
        |-- __init__.py
        |-- settings.py
        |-- urls.py
        `-- wsgi.py
```

If your folder structure doesn't look like that — particularly if you don't have the double *mysite/mysite* folder — that means you're on the wrong version of Django. Switch to someone else's PC.

Part 1 - Getting Django set up using a Selenium Functional Test

Note

You should use my repo (which is fairly empty) as the base directory for all your work. That will let you do the occasional `git checkout` to get files we need for later stages. Feel free to start a branch and do your own commits as we go!

Open a new file called *functional_tests.py*

```
from selenium import webdriver

browser = webdriver.Firefox()
browser.get('http://localhost:8000')

assert 'Django' in browser.title
print 'OK!'
browser.quit()
```

Run the file with

```
python functional_tests.py
```

Expected failure: `AssertionError`

Then start a Django project:

```
django-admin.py startproject mysite
```

Now, **in a separate console window**

```
cd mysite
python manage.py runserver
```

Back in your original console window,

```
python functional_tests.py
```

Expected pass: test prints *OK!*

`Startproject` should give you a folder structure like this:

```
.
|-- mysite
|   |-- manage.py
|   `-- mysite
|       |-- __init__.py
|       |-- settings.py
|       |-- urls.py
|       `-- wsgi.py
|-- old_tutorial/
|-- pycon_2013_handout.asciidoc
|-- pycon_2013_handout.html
|-- reference_project/
`-- workshop.rst
```

Tip

If it doesn't work, check the port — is `runserver` using port 8000, or is it on a different one? 8001?

Note

Advanced task

Can you make the dev server run on a different port? And adjust the FT accordingly? What about running just at *http://localhost/*, with no `:`?

Part 2 - `unittest` and looking for our site's home page

We update *functional_tests.py* to use the `unittest` module.

```
import unittest
from selenium import webdriver

class PollsFunctionalTest(unittest.TestCase):

    def setUp(self):
        self.browser = webdriver.Firefox()
        self.browser.implicitly_wait(3)

    def tearDown(self):
        self.browser.quit()

    def test_voting_on_a_poll(self):
        # Elspeth goes to check out a cool new polls site she's heard about
        self.browser.get('http://localhost:8000')

        # It is obviously all about polls:
        self.assertIn('Poll', self.browser.title)

        self.fail('finish this test')

if __name__ == '__main__':
    unittest.main()
```

Note

Some things to ask me about, in case I don't mention them

- why is `unittest` helpful?
- What is `assertIn`?
- `setUp`, `tearDown`
- `if __name__ == '__main__'`
- `self.browser.implicitly_wait()`

Expected failure:

```
AssertionError: 'Polls' not found in u'Welcome to Django'
```

Tip

If you get a message saying "Problem loading page" or "Unable to connect", could it be because the dev server isn't running? Use `python manage.py runserver` to start it up again…

Finish writing the FT as comments:

```
def test_voting_on_a_poll(self):
    # Elspeth goest to check out a cool new polls site she's heard about
    self.browser.get('http://localhost:8000')

    # It is obviously all about polls:
    self.assertIn('Poll', self.browser.title)

    # She clicks on the link to the first Poll, which is titled
    # "How awesome is TDD?"
    self.fail('finish this test')

    # She is taken to a poll 'results' page, which says
    # "No-one has voted on this poll yet"

    # She also sees a form, which offers her several choices.
    # There are three options with radio buttons

    # She decided to select "very awesome", which is answer #1

    # Elspeth clicks 'submit'

    # The page refreshes, and she sees that her choice
    # has updated the results.  They now say
    # "1 vote" and "100 %: very awesome".

    # Elspeth decides to try to vote again

    # The site is not very clever (yet) so it lets her

    # She votes for another choice, and the percentages go 50%-50%

    # She votes again, and they go 66% - 33%

    # Satisfied, she goes back to sleep

[...]
```

Tip

Shortcut to typing all that in: from the top-level of the repo (not in mysite):

```
git checkout PYCON_2013_PART_2_FT -- functional_tests.py
```

Finish up by **moving** *functional_tests.py* into the *mysite* folder

Note

Advanced task

Look up some of the other assertion methods in unittest. Do they all make sense? What might you use *assertItemsEqual* for?

Part 3 - Unit tests, a Django app, urls.py and views.py

Create a polls app and run its unit tests

Run the following command:

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

Your directory tree will now look like this:

```
mysite
|-- functional_test.py
|-- manage.py
|-- mysite
|   |-- __init__.py
|   |-- settings.py
|   |-- urls.py
|   `-- wsgi.py
`-- polls
    |-- __init__.py
    |-- models.py
    |-- tests.py
    `-- views.py
```

Now we deliberately break the unit test at *polls/tests.py*

```
from django.test import TestCase

class SimpleTest(TestCase):
    def test_basic_addition(self):
        """
        Tests that 1 + 1 always equals 2.
        """
        self.assertEqual(1 + 1, 3)
```

To run it: `python manage.py test`

Expected Failure 1:

```
settings.DATABASES is improperly configured.
```

Note	Ask me about: The difference between unit tests and functional tests

Fix in *mysite/settings.py*

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': '',                # Or path to database file if using sqlite3.
```

```
$ python manage.py test # so many tests!
```

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

Expected Failure:

```
ImproperlyConfigured: App with label polls could not be found
```

Note | Ask me about: re-usable apps?

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Uncomment the next line to enable the admin:
    # 'django.contrib.admin',
    # Uncomment the next line to enable admin documentation:
    # 'django.contrib.admindocs',
    'polls',
)
```

Expected failure:

```
AssertionError: 2 != 3
```

Django url mapping in urls.py

Now change *polls/tests.py*, throwing away almost all the old stuff

```
from django.core.urlresolvers import resolve
from django.test import TestCase
from polls.views import home_page

class HomePageTest(TestCase):

    def test_root_url_resolves_to_home_page_view(self):
        found = resolve('/')
        self.assertEqual(found.func, home_page)
```

Expected failure from `manage.py test polls`:

```
ImportError: cannot import name home_page
```

In *polls/views.py*:

```
# Create your views here.
home_page = None
```

Note | ask me about: that being totally ridiculous!

Expected failure:

```
Resolver404: {'path': '', 'tried': []}
```

In *mysite/urls.py*

```
from django.conf.urls import patterns, include, url

# Uncomment the next two lines to enable the admin:
# from django.contrib import admin
# admin.autodiscover()

urlpatterns = patterns('',
    # Examples:
    url(r'^$', 'polls.views.home_page', name='home'),
    # url(r'^polls/', include('polls.foo.urls')),

    # Uncomment the admin/doc line below to enable admin documentation:
    # url(r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Uncomment the next line to enable the admin:
    # url(r'^admin/', include(admin.site.urls)),
)
```

Expected failure:

```
ViewDoesNotExist: Could not import polls.views.home_page. View is not callable.
```

Note	ask me about: dot-notation vs importing views.

So, in *polls/views.py*

```
# Create your views here.

def home_page():
    pass
```

Test should pass!

Note	Advanced task • Would a lambda function work? Are there any other Python objects you could use that would still get the tests to pass? • What happens when you use the empty string ('') as the URL you call in the test? What about two slashes (//)

A minimal view to return static HTML in views.py

We extend the unit tests in *polls/tests.py*, to say we want our view to return some static HTML…

```
from django.core.urlresolvers import resolve
from django.test import TestCase
from django.http import HttpRequest
from polls.views import home_page

class HomePageTest(TestCase):

    def test_root_url_resolves_to_home_page_view(self):
        found = resolve('/')
        self.assertEqual(found.func, home_page)


    def test_home_page_returns_correct_html(self):
        request = HttpRequest()
        response = home_page(request)
        self.assertTrue(response.content.startswith('<html>'))
        self.assertIn('<title>Poll ALL The Things</title>', response.content)
        self.assertTrue(response.content.endswith('</html>'))
```

Don't forget to import `HttpRequest`

Expected failure:

```
TypeError: home_page() takes no arguments (1 given)
```

- Minimal code change:

```
def home_page(request):
    pass
```

- Tests:

```
    self.assertTrue(response.content.startswith('<html>'))
AttributeError: 'NoneType' object has no attribute 'content'
```

- Code

```
from django.http import HttpResponse

def home_page(request):
    return HttpResponse()
```

- Tests again:

```
    self.assertTrue(response.content.startswith('<html>'))
AssertionError: False is not true
```

- Code again:

```
def home_page(request):
    return HttpResponse('<html>')
```

- Tests:

```
AssertionError: '<title>Poll ALL The Things</title>' not found in '<html>'
```

- Code:

```
def home_page(request):
    return HttpResponse('<html><title>Poll ALL The Things</title>')
```

- Tests — almost there?

```
    self.assertTrue(response.content.endswith('</html>'))
AssertionError: False is not true
```

- Come on, one last effort:

```
def home_page(request):
    return HttpResponse('<html><title>Poll ALL The Things</title></html>')
```

- Surely?

```
$ python manage.py test polls
Creating test database for alias 'default'...
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

Now we re-run our functional test, and we expect them to get past the `assertIn` and stop on the `self.fail`

Note	Advanced task Can you rewrite the view as a one-liner? Well done. But don't do that in real life!

Part 4 - Switching to templates

We extend the FT a little:

```
    def test_voting_on_a_poll(self):
        # Elspeth goes to check out a cool new polls site he's heard about
        self.browser.get('http://localhost:8000')

        # It is obviously all about polls:
        self.assertIn('Poll', self.browser.title)
        heading = self.browser.find_element_by_tag_name('h1')
        self.assertEquals(heading.text, 'Current polls')

        # She clicks on the link to the first Poll, which is titled
        # "How awesome is TDD?"
        self.browser.find_element_by_link_text('How awesome is TDD?').click()

        # She is taken to a poll 'results' page, which says
        # "No-one has voted on this poll yet"
        self.fail('finish this test')
```

Expected failure is:

```
NoSuchElementException: Message: u'Unable to locate element: {"method":"tag
name","selector":"h1"}' ; Stacktrace: [...]
```

Note | Ask me about: `find_element_by_tag_name` vs `find_elements_by_tag_name`

Refactoring

Note | Ask me about: “Don’t test constants”

We start with passing tests:

```
python manage.py test polls
[...]
OK
```

- make a new directory at polls/templates

Then open a file at *polls/templates/home.html*, to which we’ll transfer our HTML:

```
<html>
    <title>Poll ALL The Things</title>
</html>
```

Now change *polls/views.py*:

```
from django.shortcuts import render

def home_page(request):
```

```
    return render(request, 'home.html')
```

Oops, an unexpected failure:

```
    self.assertTrue(response.content.endswith('</html>'))
AssertionError: False is not true
```

Add a `print` statement to test to debug:

```
    def test_home_page_returns_correct_html(self):
        request = HttpRequest()
        response = home_page(request)
        self.assertTrue(response.content.startswith('<html>'))
        self.assertIn('<title>Poll ALL The Things</title>', response.content)
        print repr(response.content)
        self.assertTrue(response.content.endswith('</html>'))
```

And fix, in your own way…. Then we change the test:

```
[...]
from django.template.loader import render_to_string
[...]

    def test_home_page_renders_correct_template(self):
        request = HttpRequest()
        response = home_page(request)
        expected_html = render_to_string('home.html')
        self.assertEqual(response.content, expected_html)
```

Note	Ask me about the Django Test Client NOTE: Ask me what Kent Beck said — "do I really expect you to always code like this?"

Adding the h1 to our home page:

```
<html>
    <head>
        <title>Poll ALL The Things</title>
    </head>
    <body>
        <h1>Current polls</h1>
    </body>
</html>
```

Expected failure:

```
NoSuchElementException: Message: u'Unable to locate element:
{"method":"link text","selector":"How awesome is TDD?"}' ; Stacktrace:
[...]
```

Hopefully we'll have a break at this point!

Advanced	How would you test that we are returning valid (standards-compliant) HTML?

Part 5 - The Django admin site

Add a new test method to *functional_tests.py*:

```
def test_can_create_a_new_poll_via_admin_site(self):
    # Mo the administrator goes to the admin page
    self.browser.get('http://localhost:8000/admin/')

# He sees the familiar 'Django administration' heading
body = self.browser.find_element_by_tag_name('body')
self.assertIn('Django administration', body.text)
self.fail('Finish this test')
```

Note	Ask me about — DONTifying tests

Expected failure:

```
AssertionError: 'Django administration' not found in u"Page not found
(404)\nRequest Method: GET\nRequest URL:
http://localhost:8000/admin/\nUsing the URLconf defined in mysite.urls,
Django tried these URL patterns, in this order:\n^$ [name='home']\nThe
current URL, admin/, didn't match any of these.\[..]
```

Switch on the admin involves uncommenting 3 lines in 2 files:

mysite/settings.py:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Uncomment the next line to enable the admin:
    'django.contrib.admin',
    # Uncomment the next line to enable admin documentation:
    # 'django.contrib.admindocs',
    'polls',
)
```

mysite/urls.py:

```
# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    # Examples:
    url(r'^$', 'polls.views.home_page', name='home'),

    # Uncomment the next line to enable the admin:
    url(r'^admin/', include(admin.site.urls)),
)
```

Expected failure (at the top of a long traceback):

```
AssertionError: 'Django administration' not found in u'ImproperlyConfigured
at /admin/\nsettings.DATABASES is improperly configured. Please supply the
NAME value.\nRequest Method: GET\ [...]
```

Add a database name in *settings.py*:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'db.sqlite', # Or path to database file if using sqlite3.
```

Expected failure (at the top of a long traceback):

```
AssertionError: 'Django administration' not found in u"DatabaseError at
/admin/\nno such table: django_site\nRequest Method:
```

Run syncdb: `python manage.py syncdb`

Remember the username and password you use — I'm using `admin` and `adm1n.` Should now get to:

```
AssertionError: Finish this test
```

Now the FT should be able to log into the admin site:

```
    def test_can_create_a_new_poll_via_admin_site(self):
        # Mo the administrator goes to the admin page
        self.browser.get('http://localhost:8000/admin/')

        # He sees the familiar 'Django administration' heading
        body = self.browser.find_element_by_tag_name('body')
        self.assertIn('Django administration', body.text)

        # He types in his username and passwords and hits return
        username_field = self.browser.find_element_by_name('username')
        username_field.send_keys('admin')

        password_field = self.browser.find_element_by_name('password')
        password_field.send_keys('adm1n')
        password_field.send_keys(Keys.RETURN)

        # His username and password are accepted, and he is taken to
        # the Site Administration page
        body = self.browser.find_element_by_tag_name('body')
        self.assertIn('Site administration', body.text)

        self.fail('Use the admin site to create a poll')
```

Expected failure: `AssertionError: Use the admin site to create a poll`

Advanced	What other methods could we have used, apart from `find_element_by_name`, to find the username and password fields? What about clicking instead of pressing RETURN?

Part 6: A model for Polls

Extend the FT:

```
    [...]
    # His username and password are accepted, and he is taken to
    # the Site Administration page
    body = self.browser.find_element_by_tag_name('body')
    self.assertIn('Site administration', body.text)

    # He sees a section named "Polls" with a model called "Polls" in it
    polls_links = self.browser.find_elements_by_link_text('Polls')
    self.assertEquals(len(polls_links), 2)
    self.fail('Use the admin site to create a poll')
```

Expected failure:

```
    self.assertEquals(len(polls_links), 2)
AssertionError: 0 != 2
```

Unit test for our Poll model:

```
from django.core.urlresolvers import resolve
from django.http import HttpRequest
from django.template.loader import render_to_string
from django.test import TestCase
from django.utils import timezone
from polls.models import Poll
from polls.views import home_page

class PollModelTest(TestCase):

    def test_creating_a_new_poll_and_saving_it_to_the_database(self):
        # start by creating a new Poll object with its "question" set
        poll = Poll()
        poll.question = "What's up?"
        poll.pub_date = timezone.now()

        # check we can save it to the database
        poll.save()

        # now check we can find it in the database again
        all_polls_in_database = Poll.objects.all()
        self.assertEquals(len(all_polls_in_database), 1)
        only_poll_in_database = all_polls_in_database[0]
        self.assertEquals(only_poll_in_database, poll)

        # and check that it's saved its two attributes: question and pub_date
        self.assertEquals(only_poll_in_database.question, "What's up?")
        self.assertEquals(only_poll_in_database.pub_date, poll.pub_date)


class HomePageTest(TestCase):

    def test_root_url_resolves_to_home_page_view(self):
```

```
        [...]
```

Don't miss the 2 extra imports (I did!)

- Expected failure:

```
ImportError: cannot import name Poll
```

- Now edit *polls/models.py*:

```
from django.db import models

Poll = None
```

- Expected failure:

```
TypeError: 'NoneType' object is not callable
    ImportError: cannot import name Poll
```

- *models.py*:

```
from django.db import models

class Poll(object):
    pass
```

- failure:

```
AttributeError: 'Poll' object has no attribute 'save'
```

- inherit:

```
class Poll(models.Model):
    pass
```

- failure - note it's quite late!

```
AttributeError: 'Poll' object has no attribute 'question'
```

- add question attribute

```
class Poll(models.Model):
    question = models.CharField(max_length=200)
```

- new failure:

```
AttributeError: 'Poll' object has no attribute 'pub_date'
```

- new field - deliberately wrong:

```
class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.CharField(max_length=200)
```

- sure enough, tests help us:

```
AssertionError: u'2013-03-03 12:40:29.241235+00:00' !=
datetime.datetime(2013, 3, 3, 12, 40, 29, 241235, tzinfo=<UTC>)
```

- fix

```
pub_date = models.DateTimeField()
```

- and it should now work!

```
$ python manage.py test polls
Creating test database for alias 'default'...
...
----------------------------------------------------------------------
Ran 3 tests in 0.008s

OK
```

Do the FTs pass? No, still need to *register* Polls in the admin site, using a new file at *polls/admin.py*

```
from django.contrib import admin
from polls.models import Poll

admin.site.register(Poll)
```

And now we should get our self.fail:

```
AssertionError: Use the admin site to create a poll
```

Note	Advanced task Give pub_date a verbose name of *Date published*. See the official tutorial for the implementation… but can you find a way to unit test it? Hint: the model `._meta` attribute might work… Is there another way?

Part 7: LiveServerTestCase and test fixtures

Start by extending the FT to actually create a new poll via the admin site:

```
    # He clicks the 'Add poll' link
    new_poll_link = self.browser.find_element_by_link_text('Add poll')
    new_poll_link.click()

    # He types in an interesting question for the Poll
    question_field = self.browser.find_element_by_name('question')
    question_field.send_keys("How awesome is Test-Driven Development?")

    # He sets the date and time of publication - it'll be a new year's
    # poll!
    date_field = self.browser.find_element_by_name('pub_date_0')
    date_field.send_keys('01/01/12')
    time_field = self.browser.find_element_by_name('pub_date_1')
    time_field.send_keys('00:00')

    # Mo clicks the save button
    save_button = self.browser.find_element_by_css_selector("input[value='Save']")
    save_button.click()

    # He is returned to the "Polls" listing, where he can see his
    # new poll, listed as a clickable link
    new_poll_links = self.browser.find_elements_by_link_text(
            "How awesome is Test-Driven Development?"
    )
    self.assertEquals(len(new_poll_links), 1)
```

First expected fail -

```
    self.assertEquals(len(new_poll_links), 1)
AssertionError: 0 != 1
```

__unicode__

Fix by changing the string representation of a poll:

in *polls/tests.py*, add to `PollModelTest`:

```
    def test_string_representation(self):
        poll = Poll()
        poll.question = "Why?"
        self.assertEqual(unicode(poll), "Why?")
```

Expected fail:

```
AssertionError: u'Poll object' != 'Why?'
```

models.py:

```
class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField()

    def __unicode__(self):
        return self.question
```

Unit tests should now pass

LiveServerTestCase and the test database

Functional tests should pass once… but fail the second time:

```
AssertionError: '0 polls' not found in u'Django administration\nWelcome, admin.
Change password / Log out\nHome \u203a Polls \u203a Polls\nSelect poll to
change\nAdd poll\nAction:\n---------\nDelete selected polls\nGo 0 of 1
selected\nPoll\nHow awesome is Test-Driven Development?\n1 poll'
```

change *functional_tests.py* to being tests inside a new Django app called *fts*:

```
python manage.py startapp fts
mv functional_tests.py fts/tests.py
```

then edit *fts/tests.py* to inherit from `LiveServerTestCase`:

```
from django.test import LiveServerTestCase
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

class PollsFunctionalTest(LiveServerTestCase):

    def setUp(self):
        self.browser = webdriver.Firefox()
        self.browser.implicitly_wait(3)

    def tearDown(self):
        self.browser.quit()

    def test_voting_on_a_poll(self):
        # Elspeth goes to check out a cool new polls site she's heard about
        self.browser.get(self.live_server_url)

        [...]

    def test_can_create_a_new_poll_via_admin_site(self):
        # Mo the administrator goes to the admin page
        self.browser.get(self.live_server_url + '/admin/')
        [...]
```

- make sure to use `self.live_server_url` in both test methods
- also delete the `if __name__ == __main__` block

Add `fts` to *settings.py*:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Uncomment the next line to enable the admin:
    'django.contrib.admin',
    # Uncomment the next line to enable admin documentation:
    # 'django.contrib.admindocs',
    'polls',
    'fts',
)
```

Now run

```
$ python manage.py test fts
```

Should see one `self.fail` (can DONTify this test now) and one:

```
    self.assertIn('Site administration', body.text)
AssertionError: 'Site administration' not found in u'Django
administration\nPlease enter the correct username and password for a staff
account. Note that both fields may be case-sensitive.\nUsername:\nPassword:\n '
```

Test fixture setup

- make a new directory at *polls/fixtures*

```
python manage.py dumpdata auth.user > polls/fixtures/admin_user.json
```

Add to *fts/tests.py*:

```
class PollsFunctionalTest(LiveServerTestCase):

    fixtures = ['admin_user.json']

    def setUp(self):
        [...]
```

FT should now pass, no matter how many times you run them!

By the end, your folder structure should look like this:

```
.
|-- fts
|   |-- __init__.py
|   |-- models.py
|   |-- tests.py
|   `-- views.py
|-- manage.py
|-- mysite
|   |-- __init__.py
|   |-- settings.py
|   |-- urls.py
|   `-- wsgi.py
`-- polls
    |-- admin.py
    |-- fixtures
    |   `-- admin_user.json
    |-- __init__.py
    |-- models.py
    |-- templates
    |   `-- home.html
    |-- tests.py
    `-- views.py
```

Part 8 - Add the Choice model

Add a bit to the FT (*fts/tests.py*), just before we save the new poll

```
    # He sets the date and time of publication - it'll be a new year's
    # poll!
    date_field = self.browser.find_element_by_name('pub_date_0')
    date_field.send_keys('01/01/12')
    time_field = self.browser.find_element_by_name('pub_date_1')
    time_field.send_keys('00:00')

    # He sees he can enter choices for the Poll.  He adds three
    choice_1 = self.browser.find_element_by_name('choice_set-0-choice')
    choice_1.send_keys('Very awesome')
    choice_2 = self.browser.find_element_by_name('choice_set-1-choice')
    choice_2.send_keys('Quite awesome')
    choice_3 = self.browser.find_element_by_name('choice_set-2-choice')
    choice_3.send_keys('Moderately awesome')

    # Mo clicks the save button
    save_button = self.browser.find_element_by_css_selector("input[value='Save']")
```

Expected failure for `manage.py test fts`:

```
NoSuchElementException: Message: u'Unable to locate element:
{"method":"name","selector":"choice_set-0-choice"}' ; Stacktrace: [...]
```

Now in the unit tests - *polls/tests.py*

```
[...]
from django.utils import timezone
from polls.models import Choice, Poll
from polls.views import home_page

class PollModelTest(TestCase):
    [...]


class ChoiceModelTest(TestCase):

    def test_creating_some_choices_for_a_poll(self):
        # start by creating a new Poll object
        poll = Poll()
        poll.question="What's up?"
        poll.pub_date = timezone.now()
        poll.save()

        # now create a Choice object
        choice = Choice()

        # link it with our Poll
        choice.poll = poll

        # give it some text
        choice.choice = "doin' fine..."
```

```
        # and let's say it's had some votes
        choice.votes = 3

        # save it
        choice.save()

        # try retrieving it from the database, using the poll object's reverse
        # lookup
        poll_choices = poll.choice_set.all()
        self.assertEquals(poll_choices.count(), 1)

        # finally, check its attributes have been saved
        choice_from_db = poll_choices[0]
        self.assertEquals(choice_from_db.id, choice.id)
        self.assertEquals(choice_from_db.choice, "doin' fine...")
        self.assertEquals(choice_from_db.votes, 3)
```

- Expected failure:

```
ImportError: cannot import name Choice
```

- *polls/models.py*:

```
class Choice(object):
    pass
```

```
AttributeError: 'Choice' object has no attribute 'save'
```

- *models.py*

```
class Choice(models.Model):
    pass
```

```
AttributeError: 'Poll' object has no attribute 'choice_set'
```

- *models.py*

```
class Choice(models.Model):
    poll = models.ForeignKey(Poll):

    self.assertEquals(choice_from_db.choice, "doin' fine...")
AttributeError: 'Choice' object has no attribute 'choice'
```

- *models.py*

```
class Choice(models.Model):
    poll = models.ForeignKey(Poll)
    choice = models.CharField(max_length=200):
```

```
AttributeError: 'Choice' object has no attribute 'votes'
```

- *models.py*

```
class Choice(models.Model):
    poll = models.ForeignKey(Poll)
```

```
    choice = models.CharField(max_length=200)
    votes = models.IntegerField()
```

Now, in *polls/admin.py*

```
from django.contrib import admin
from polls.models import Choice, Poll

class ChoiceInline(admin.StackedInline):
    model = Choice
    extra = 3

class PollAdmin(admin.ModelAdmin):
    inlines = [ChoiceInline]

admin.site.register(Poll, PollAdmin)
```

Run the FT - still fails:

```
    self.assertEquals(len(new_poll_links), 1)
AssertionError: 0 != 1
```

Inspect manually… Need to add a default -- in *polls/tests.py*:

```
class ChoiceModelTest(TestCase):

    def test_creating_some_choices_for_a_poll(self):
        [...]

    def test_choice_defaults(self):
        choice = Choice()
        self.assertEquals(choice.votes, 0)
```

polls/models.py:

```
class Choice(models.Model):
    poll = models.ForeignKey(Poll)
    choice = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

FT should now pass

Note | ask me about: `TemplateDoesNotExist: 500.html` and `settings.DEBUG`

Note | Advanced task: Figure out how to fix the `TemplateDoesNotExist: 500.html` issue

Part 9 - The Page pattern

Start by refactoring the admin ft:

```
from datetime import datetime
from django.test import LiveServerTestCase
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

class AdminPage(object):

    def __init__(self, test, browser):
        self.test = test
        self.browser = browser

    def login(self):
        # Mo the administrator goes to the admin page
        self.browser.get(self.test.live_server_url + '/admin/')

        # He sees the familiar 'Django administration' heading
        body = self.browser.find_element_by_tag_name('body')
        self.test.assertIn('Django administration', body.text)

        # He types in his username and passwords and hits return
        username_field = self.browser.find_element_by_name('username')
        username_field.send_keys('admin')

        password_field = self.browser.find_element_by_name('password')
        password_field.send_keys('adm1n')
        password_field.send_keys(Keys.RETURN)

        # His username and password are accepted, and he is taken to
        # the Site Administration page
        body = self.browser.find_element_by_tag_name('body')
        self.test.assertIn('Site administration', body.text)


    def logout(self):
        self.browser.find_element_by_link_text('Log out').click()


    def add_poll(self, question, pub_date, choices):
        self.browser.get(self.test.live_server_url + '/admin/')
        # He sees a section named "Polls" with a model called "Polls" in it
        polls_links = self.browser.find_elements_by_link_text('Polls')
        self.test.assertEquals(len(polls_links), 2)
        polls_links[1].click()

        # He clicks the 'Add poll' link
        new_poll_link = self.browser.find_element_by_link_text('Add poll')
        new_poll_link.click()

        # He types in an interesting question for the Poll
        question_field = self.browser.find_element_by_name('question')
        question_field.send_keys(question)
```

```
        # He sets the date and time of publication
        date_field = self.browser.find_element_by_name('pub_date_0')
        date_field.send_keys(pub_date.date().strftime('%x'))
        time_field = self.browser.find_element_by_name('pub_date_1')
        time_field.send_keys(pub_date.time().strftime('%X'))

        # He sees he can enter choices for the Poll.  He adds them
        for no, choice in enumerate(choices):
            choice_input = self.browser.find_element_by_name(
                'choice_set-%d-choice' % (no,)
            )
            choice_input.send_keys(choice)

        # Mo clicks the save button
        save_button = self.browser.find_element_by_css_selector("input[value='Save']")
        save_button.click()

        # He is returned to the "Polls" listing, where he can see his
        # new poll, listed as a clickable link
        new_poll_links = self.browser.find_elements_by_link_text(
                question
        )
        self.test.assertEquals(len(new_poll_links), 1)


class PollsFunctionalTest(LiveServerTestCase):

    [...]

    def test_voting_on_a_poll(self):
        [...]


    def test_can_create_a_new_poll_via_admin_site(self):
        # Mo the administrator goes to the admin page
        # and creates a new poll, with 3 choices
        admin_page = AdminPage(self, self.browser)
        admin_page.login()
        admin_page.add_poll(
            question="How awesome is Test-Driven Development?",
            pub_date=datetime(2012,01,01),
            choices = ['Very awesome', 'Quite awesome', 'Moderately awesome']
        )
        admin_page.logout()
```

Note | Ask me about: “Three strikes then refactor”

Check it works by running `python manage.py test fts`.

Then, use our new AdminPage to pre-populate some polls for our other FT:

```
    def test_voting_on_a_poll(self):
        # Mo the administrator has entered a couple of polls
        admin_page = AdminPage(self, self.browser)
        admin_page.login()
        admin_page.add_poll(
            question="How awesome is TDD?",
```

```
            pub_date = datetime.today(),
            choices=['Very awesome', 'Quite awesome', 'Moderately awesome'],
        )
        admin_page.add_poll(
            question="Which workshop treat do you prefer?",
            pub_date = datetime.today(),
            choices=['Beer', 'Pizza', 'The Acquisition of Knowledge'],
        )
        admin_page.logout()

        # Elspeth goes to check out a cool new polls site she's heard about
        self.browser.get(self.live_server_url)

        # It is obviously all about polls:
        self.assertIn('Poll', self.browser.title)
        heading = self.browser.find_element_by_tag_name('h1')
        self.assertEquals(heading.text, 'Current polls')

        # She clicks on the link to the first Poll, which is titled
        # "How awesome is TDD?"
        self.browser.find_element_by_link_text('How awesome is TDD?').click()

        # She is taken to a poll 'results' page, which says
        # "No-one has voted on this poll yet"
        body = self.browser.find_element_by_tag_name('body')
        self.test.assertIn("No-one has voted on this poll yet", body.text)
        # She also sees a form, which offers her several choices.
        # There are three options with radio buttons
        self.fail('finish this test')
```

Expected fail:

```
NoSuchElementException: Message: u'Unable to locate element:
{"method":"link text","selector":"How awesome is TDD?"}' [...]
```

Note	Advanced task Remove some of the duplicated strings like the poll question, and use some constants instead

Fixing that darned 500 template error!

It's about time we sorted this out!

```
mkdir mysite/templates
echo "Unexpected Error (500) :-/" > mysite/templates/500.html
```

then, in *mysite/settings.py*:

```
import os
[...]
TEMPLATE_DIRS = (
    # Put strings here, like "/home/html/django_templates" or "C:/www/django/templates".
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
    os.path.join(os.path.dirname(__file__), 'templates').replace('\\', '/'),
)
```

Part 10 - Listing polls on the home page template

Note

Skipping ahead to this section

From the top-level folder of the repo

```
git stash # if you want to save what you had so far
git checkout PYCON_2013_PART_10 -- mysite
```

Will blow away everything in *mysite* and replace them with as if you'd skipped to this part

`python manage.py test fts` should give:

```
NoSuchElementException: Message: u'Unable to locate element: {"method":"link
text","selector":"How awesome is TDD?"}'
```

So start by adding check for poll questions to our view unit test. In *polls/tests.py*, change `test_home_page_renders_correct_template` inside `HomePageTest`, to:

```
def test_home_page_renders_home_template_with_current_polls(self):
    # set up some polls
    poll1 = Poll(question='6 times 7', pub_date=timezone.now())
    poll1.save()
    poll2 = Poll(question='life, the universe and everything', pub_date=timezone.now())
    poll2.save()

    request = HttpRequest()
    response = home_page(request)

    # check template rendered correctly
    expected_html = render_to_string('home.html')
    self.assertEqual(response.content, expected_html)

    # check template includes all polls
    self.assertIn(poll1.question, response.content)
    self.assertIn(poll2.question, response.content)
```

Should fail:

```
AssertionError: '6 times 7' not found in '<html>\n    <head>\n
<title>Poll ALL The Things</title>\n    </head>\n    <body>\n
<h1>Current polls</h1>\n    </body>\n</html>\n'
```

Now add them to our template, *polls/templates/home.html*, using special Django template tags — `{% %}` and `{{ }}`

```
<html>
    <head>
        <title>Poll ALL The Things</title>
    </head>
    <body>
        <h1>Current polls</h1>
        <ul>
        {% for poll in current_polls %}
            <li>{{ poll.question }}</li>
        {% endfor %}
        </ul>
    </body>
</html>
```

Tests still fail - v. slightly different error.

Note | ask me about — Django template syntax. obviously

Where would `current_polls` come from? They're actually passed into the render call - we can test that! In *polls/tests.py*:

```
    expected_html = render_to_string('home.html', {'current_polls': [poll1, poll2]})
    self.assertEqual(response.content, expected_html)
```

Now test failure happens earlier :

```
    self.assertEqual(response.content, expected_html)
AssertionError: '<html>\n    <head>\n        <title>Poll ALL The
Things</title>\n    </head>\n    <body>\n        <h1>Current polls</h1>\n
<ul>\n        \n        </ul>\n    </body>\n</html>\n' != u'<html>\n <head>\n
<title>Poll ALL The Things</title>\n    </head>\n    <body>\n <h1>Current
polls</h1>\n        <ul>\n        \n            <li>6 times 7</li>\n        \n
<li>life, the universe and everything</li>\n        \n        </ul>\n
</body>\n</html>\n'
```

Yuk! Let's try using `assertMultiLineEqual`:

```
    # render template with polls
    expected_html = render_to_string('home.html', {'current_polls': [poll1, poll2]})
    self.assertMultiLineEqual(response.content, expected_html)
```

Much better:

```
AssertionError: '<html>\n    <head>\n        <title>Poll ALL The
Things</title>\n    </head>\n   [truncated]... != u'<html>\n    <head>\n
<title>Poll ALL The Things</title>\n    </head>\n  [truncated]...
  <html>
      <head>
          <title>Poll ALL The Things</title>
      </head>
      <body>
          <h1>Current polls</h1>
          <ul>
+             <li>6 times 7</li>
+
+             <li>life, the universe and everything</li>
+
          </ul>
      </body>
  </html>
```

Fix in *polls/views.py*:

```
from django.shortcuts import render
from polls.models import Poll

def home_page(request):
    return render(request, 'home.html', {'current_polls': Poll.objects.all()})
```

Unit tests should now pass - how about FTs? Not quite - but they do get further

```
NoSuchElementException: Message: u'Unable to locate element: {"method":"link
text","selector":"How awesome is TDD?"}' ;
```

Let's make our poll questions into hyperlinks in the template:

```
    {% for poll in current_polls %}
        <li><a>{{ poll.question }}</a></li>
    {% endfor %}
```

FT gets a little further

```
AssertionError: 'No-one has voted on this poll yet' not found in u'Current
polls\nHow awesome is TDD?\nWhich workshop treat do you prefer?'
```

Part 11 - viewing a poll and the Django Test Client

We want individual polls to have their own URL - let's specify that in *polls/templates/home.html*:

```
<html>
    <head>
        <title>Poll ALL The Things</title>
    </head>
    <body>
        <h1>Current polls</h1>
        <ul>
        {% for poll in current_polls %}
            <li><a href="/poll/{{ poll.id }}/">{{ poll.question }}</a></li>
        {% endfor %}
        </ul>
    </body>
</html>
```

Of course that URL doesn't exist yet - try running the FT and you'll get a 500 server error

So let's add a test for our new url, in *polls/tests.py*. This time we use the *Django Test Client*:

```
from polls import views
[...]
class HomePageTest(TestCase):
    [...]

class SinglePollViewTest(TestCase):

    def test_page_shows_poll_title_and_no_votes_message(self):
        # set up two polls, to check the right one is displayed
        poll1 = Poll(question='6 times 7', pub_date=timezone.now())
        poll1.save()
        poll2 = Poll(question='life, the universe and everything',
pub_date=timezone.now())
        poll2.save()

        response = self.client.get('/poll/%d/' % (poll2.id, ))

        # check we've used the poll template
        self.assertTemplateUsed(response, 'poll.html')

        # check we've passed the right poll into the context
        self.assertEquals(response.context['poll'], poll2)

        # check the poll's question appears on the page
        self.assertIn(poll2.question, response.content)

        # check our 'no votes yet' message appears
        self.assertIn('No-one has voted on this poll yet', response.content)
```

gives :

```
TemplateDoesNotExist: 404.html
```

Let's add a minimal 404 template, just like we did for the 500:

```
echo "Page not found (404) :-/" > mysite/templates/404.html
```

Now we get:

```
AssertionError: Template 'poll.html' was not a template used to render the
response. Actual template(s) used: 404.html
```

OK, so let's fix the 404. Here's a possible fix in *mysite/urls.py*:

```
urlpatterns = patterns('',
    url(r'^$', 'polls.views.home_page', name='home'),
    url(r'^poll/(\d+)/$', 'polls.views.poll', name='poll'),

    url(r'^admin/', include(admin.site.urls)),
)
```

which gives

```
ViewDoesNotExist: Could not import polls.views.poll. View does not exist in
module polls.views.
```

Now enter a TDD/code cycle. I will show just the failures:

```
TypeError: poll() takes exactly 1 argument (2 given)
```

```
ValueError: The view polls.views.poll didn't return an HttpResponse object.
```

```
AssertionError: No templates used to render the response
```

(deliberate mistake)

```
AssertionError: Template 'poll.html' was not a template used to render the response.
Actual template(s) used: home.html
```

Then:

```
TemplateDoesNotExist: poll.html
```

So we create it! minimally, at *polls/templates/poll.html*:

```
<html>
</html>
```

And now:

```
    self.assertEquals(response.context['poll'], poll2)
  File "/usr/local/lib/python2.7/dist-packages/django/template/context.py", line 54, in
__getitem__
    raise KeyError(key)
KeyError: 'poll'
```

So we pass poll in our context:

```
def poll(request, poll_id):
    return render(request, 'poll.html', {'poll': None})
```

tests:

```
AssertionError: None != <Poll: life, the universe and everything>
```

So:

```
def poll(request, poll_id):
    poll = Poll.objects.get(id=poll_id)
    return render(request, 'poll.html', {'poll': poll})
```

gives

```
AssertionError: 'life, the universe and everything' not found in
'<html>\n</html>\n'
```

So improve the template:

```
<html>
    <head>
        <title>{{ poll.question }}</title>
    </head>
    <body>
        <h1>{{ poll.question }}</h1>
    </body>
</html>
```

And then:

```
AssertionError: 'No-one has voted on this poll yet' not found in '<html>\n
<head>\n        <title>life, the universe and everything</title>\n    </head>\n
<body>\n        <h1>life, the universe and everything</h1>\n
</body>\n</html>\n'
```

So, for now:

```
    <body>
        <h1>{{ poll.question }}</h1>
        <p>No-one has voted on this poll yet</p>
    </body>
```

FT should now get to the `self.fail`

Advanced

1. Figure out how to use **url includes** to put the poll url into *polls/urls.py* instead of *mysite/urls.py*
2. DRY! We shouldn't have these URL strings hard-coded all over the place. Find out how to remove them from the template
3. Look up how template inheritance works in Django. Make *poll.html* and *home.html* inherit from a common base template.

Part 12 - Voting on a poll

We extend the FT

```
    self.assertIn("No-one has voted on this poll yet", body.text)

    # She also sees a form, which offers her several choices.
    # There are three options with radio buttons
    choice_inputs = self.browser.find_elements_by_css_selector(
            "input[type='radio']"
    )
    self.assertEquals(len(choice_inputs), 3)

    # The buttons have labels to explain them
    choice_labels = self.browser.find_elements_by_tag_name('label')
    choices_text = [c.text for c in choice_labels]
    self.assertEquals(choices_text, [
        'Very awesome',
        'Quite awesome',
        'Moderately awesome',
    ])

    # She decided to select "very awesome", which is answer #1
    chosen = self.browser.find_element_by_css_selector(
            "input[value='1']"
    )
    chosen.click()

    # Elspeth clicks 'submit'
    self.browser.find_element_by_css_selector(
            "input[type='submit']"
    ).click()

    # The page refreshes, and she sees that her choice
    # has updated the results.  They now say
    # "1 vote" and "100%: very awesome".
    body = self.browser.find_element_by_tag_name('body')
    self.ertNotIn("No-one has voted on this poll yet", body.text)
    self.assertIn("1 vote", body.text)
    self.assertIn("100%: Very awesome", body.text)

    # Elspeth decides to try to vote again
    self.fail('second vote')
```

Tip

Shortcut to typing all that in:

```
git checkout PYCON_2013_PART_12_FT -- polls/fts/tests.py
```

Expected fail:

```
    self.assertEquals(len(choice_inputs), 3)
AssertionError: 0 != 3
```

Unit testing template logic

Choice inputs can be a bit tricky. Better have a unit test for them. In *polls/tests.py*, change the test method in `SinglePollViewTest`:

```
class SinglePollViewTest(TestCase):

    def test_template_rendered_with_poll_and_choice_radio_buttons_and_no_votes(
            self
    ):
        # set up two polls, to check the right one is displayed
        poll1 = Poll(question='6 times 7', pub_date=timezone.now())
        poll1.save()
        poll2 = Poll(question='life, the universe and everything',
pub_date=timezone.now())
        poll2.save()

        # add a couple of choices
        choice1 = Choice(poll=poll2, choice="42")
        choice1.save()
        choice2 = Choice(poll=poll2, choice="the Spice")
        choice2.save()

        response = self.client.get('/poll/%d/' % (poll2.id, ))

        # check we've used the poll template
        self.assertTemplateUsed(response, 'poll.html')

        # check we've passed the right poll into the context
        self.assertEquals(response.context['poll'], poll2)

        # check the poll's question appears on the page
        self.assertIn(poll2.question, response.content)

        # check our 'no votes yet' message appears
        self.assertIn('No-one has voted on this poll yet', response.content)

        # check the choices appear as radio buttons, with the
        # correct 'name' and 'value'
        self.assertIn(
            '<input type="radio" name="vote" value="%d" />' % (choice1.id,),
            response.content
        )
        self.assertIn(
            '<input type="radio" name="vote" value="%d" />' % (choice2.id,),
            response.content
        )
        # check there are labels too
        self.assertIn('<label>%s' % (choice1.choice,), response.content)
        self.assertIn('<label>%s' % (choice2.choice,), response.content)
```

Now the tests drive what we add to the template:

```
    <h1>{{ poll.question }}</h1>
    <p>No-one has voted on this poll yet</p>
    <ul>
    {% for choice in poll.choice_set.all %}
        <li><input type="radio" name="vote" value="{{ choice.id }}"/></li>
    {% endfor %}
    </ul>
```

Gets past the `<input>` tests:

```
AssertionError: '<label>42' not found in '<html>\n    <head>\n
<title>life, the universe and everything</title>\n    </head>\n    <body>\n
<h1>life, the universe and everything</h1>\n        <p>No-one has voted on this
poll yet</p>\n        \n            <input type="radio" name="vote" value="1"
/>\n        \n            <input type="radio" name="vote" value="2" />\n
\n    </body>\n</html>\n'
```

Let's add a `<label>` or two:

```
    {% for choice in poll.choice_set.all %}
        <label>{{ choice.choice }}
            <input type="radio" name="vote" value="{{ choice.id }}" />
        </label>
    {% endfor %}
```

And unit tests should now pass. The FTs want a submit input:

```
NoSuchElementException: Message: u'Unable to locate element: {"method":"css
selector","selector":"input[type=\'submit\']"}' ;
```

Now that we're asking for a submit button, we should probably have a real form that sends a POST to a real URL. Let's do that. Maybe in a new test:

```
    def test_poll_has_vote_form_which_posts_to_correct_url(self):
        poll = Poll.objects.create(question='question', pub_date=timezone.now())

        response = self.client.get('/poll/%d/' % (poll.id, ))

        self.assertIn(
            '<form method="POST" action="/poll/%d/vote">' % (poll.id,),
            response.content
        )
        self.assertIn(
            '<input type="submit"',
            response.content
        )
```

AssertionError: *<form method="POST" action="/poll/1/vote">* not found in *<html>\n <head>\n <title>question</title>\n </head>\n <body>\n <h1>question</h1>\n No-one has voted on this poll yet \n \n </body>\n</html>\n*

So we fix that:

```
    <p>No-one has voted on this poll yet</p>
    <form method="POST" action="/poll/{{ poll.id }}/vote">
        {% for choice in poll.choice_set.all %}
```

And the next fail is about the input, so we add that:

```
        {% endfor %}
        <input type="submit" value="Vote" />
    </form>
```

And now the FT should get to this:

```
AssertionError: '1 vote' not found in u'Page not found (404) :-/'
```

Because we don't yet have a URL and view to submit votes to.

Advanced	Remove any hard-coded references to urls as strings - we should have these defined in one place only. Hint: find the `reverse` function.

A new URL + view for POST submissions

Test the new URL + view with the Django Test Client. In *polls/tests.py*:

```
class SinglePollViewTest(TestCase):
    [...]

class PollVoteViewTest(TestCase):

    def test_can_vote_via_POST(self):
        # set up a poll with choices
        poll = Poll.objects.create(question='who?', pub_date=timezone.now())
        poll.save()
        choice1 = Choice.objects.create(poll=poll, choice='me', votes=1)
        choice2 = Choice.objects.create(poll=poll, choice='you', votes=3)

        # set up our POST data - keys and values are unicode
        post_data = {u'vote': unicode(choice2.id)}

        # make our request to the view
        poll_url = '/poll/%d/vote' % (poll1.id,)
        response = self.client.post(poll_url, data=post_data)

        # check it wasn't a 404
        self.assertNotEqual(response.status_code, 404)

        # retrieve the updated choice from the database
        choice_in_db = Choice.objects.get(pk=choice2.id)

        # check it's votes have gone up by 1
        self.assertEquals(choice_in_db.votes, 4)
```

```
        # "always redirect after a POST". In this case, we go back
        # to the poll page.
        self.assertRedirects(response, "/poll/%d/" % (poll1.id,))
```

Gives:

```
    self.assertNotEqual(response.status_code, 404)
AssertionError: 404 == 404
```

So, in *mysite/urls.py*:

```
    url(r'^poll/(\d+)/vote$', 'polls.views.vote', name='vote'),
```

Gives `ViewDoesNotExist:`

So, in *polls/views.py*, follow normal TDD cycle (I managed 4 steps, can you do more?) until you get to:

```
    self.assertEquals(choice_in_db.votes, 4)
AssertionError: 3 != 4
```

Now use the POST data:

```
from polls.models import Choice, Poll

[...]

def vote(request, poll_id):
    poll = Poll.objects.get(id=poll_id)
    choice = Choice.objects.get(id=request.POST['vote'])
    choice.votes += 1
    choice.save()
    return render(request, 'poll.html', {'poll': poll})
```

Then:

```
AssertionError: Response didn't redirect as expected: Response code was 200
(expected 302)
```

Finally:

```
from django.shortcuts import redirect, render

[...]

    choice.save()
    return redirect('/poll/%d/' % (poll.id,))
```

Unit tests now pass. What about the FT?

```
AssertionError: '1 vote' not found in u'Forbidden (403)\nCSRF verification
failed. Request aborted.\nMore information is available with DEBUG=True.'
```

We need to include a CSRF protection tag in our form:

```
    <form action="/poll/{{ poll.id }}/vote" method="POST">
        {% csrf_token %}
        <ul>
```

And now?

```
AssertionError: 'No-one has voted on this poll yet' unexpectedly found in u'How
awesome is TDD?\nNo-one has voted on this poll yet\nVery awesome\nQuite
awesome\nModerately awesome'
```

Next would be printing the votes… But that's up to you!

Advanced

1. Find out how CSRF protection works
2. Look up the docs for the `redirect` function. What would be a better solution?

THE END.... for now?

Thanks for coming along! I hope you enjoyed it, and I hope you found it useful.

Let me have your feedback! What went well, what could I improve? Let me know right now! Or later at the conference! Or via harry.percival@gmail.com

This doesn't need to be the end of your TDD journey — there's **loads** more content in my tutorial, at http://www.tdd-django-tutorial.com/

You can find me on Twitter via **@hjwp**

Finally, watch out for my book, (book book book!) due later this year on O'Reilly! It should be in the Early Release Program by the time PyCon comes around, so check it out and let me know what you think... I'm not even trying to sell it to you, there'll be a totes free Creative Commons version and everything!