**Extending Our Functional Test Using the unittest Module**

<http://www.obeythetestinggoat.com/book/chapter_02_unittest.html>

Goto: -- C:\Users\Public\devTddMilPub\note

Open: -- R-Click > GIT-bash

Type GIT-bash: -- $ source venvSuperlists/Scripts/activate

Let’s adapt our test, which currently checks for the default Django "it worked" page, and check instead for some of the things we want to see on the real front page of our site.

Time to reveal what kind of web app we’re building: a to-do lists site! In doing so we’re very much following fashion: a few years ago all web tutorials were about building a blog. Then it was forums and polls; nowadays it’s all to-do lists.

The reason is that a to-do list is a nice example. At its most basic it is very simple indeed—​just a list of text strings—​so it’s easy to get a "minimum viable" list app up and running. But it can be extended in all sorts of ways—​different persistence models, adding deadlines, reminders, sharing with other users, and improving the client-side UI. There’s no reason to be limited to just “to-do” lists either; they could be any kind of lists. But the point is that it should allow me to demonstrate all the main aspects of web programming, and how you apply TDD to them.

**Using a Functional Test to Scope Out a Minimum Viable App**

Tests that use Selenium let us drive a real web browser, so they really let us see how the application *functions* from the user’s point of view. That’s why they’re called *functional tests*.

This means that an FT can be a sort of specification for your application. It tends to track what you might call a *User Story*, and follows how the user might work with a particular feature and how the app should respond to them.

Terminology:   
Functional Test == Acceptance Test == End-to-End Test

What I call functional tests, some people prefer to call *acceptance tests*, or *end-to-end tests*. The main point is that these kinds of tests look at how the whole application functions, from the outside. Another term is *black box test*, because the test doesn’t know anything about the internals of the system under test.

FTs should have a human-readable story that we can follow. We make it explicit using comments that accompany the test code. When creating a new FT, we can write the comments first, to capture the key points of the User Story. Being human-readable, you could even share them with nonprogrammers, as a way of discussing the requirements and features of your app.

TDD and agile software development methodologies often go together, and one of the things we often talk about is the minimum viable app; what is the simplest thing we can build that is still useful? Let’s start by building that, so that we can test the water as quickly as possible.

A minimum viable to-do list really only needs to let the user enter some to-do items, and remember them for their next visit.

Open up *functional\_tests.py* and write a story a bit like this one:

functional\_tests.py

from selenium import webdriver

browser = webdriver.Firefox()

# Edith has heard about a cool new online to-do app. She goes

# to check out its homepage

browser.get('http://localhost:8000')

# She notices the page title and header mention to-do lists

assert 'To-Do' in browser.title

# She is invited to enter a to-do item straight away

# She types "Buy peacock feathers" into a text box (Edith's hobby

# is tying fly-fishing lures)

# When she hits enter, the page updates, and now the page lists

# "1: Buy peacock feathers" as an item in a to-do list

# There is still a text box inviting her to add another item. She

# enters "Use peacock feathers to make a fly" (Edith is very methodical)

# The page updates again, and now shows both items on her list

# Edith wonders whether the site will remember her list. Then she sees

# that the site has generated a unique URL for her -- there is some

# explanatory text to that effect.

# She visits that URL - her to-do list is still there.

# Satisfied, she goes back to sleep

browser.quit()

We Have a Word for Comments…​

When I first started at Resolver, I used to virtuously pepper my code with nice descriptive comments. My colleagues said to me: “Harry, we have a word for comments. We call them lies.” I was shocked! But I learned in school that comments are good practice?

They were exaggerating for effect. There is definitely a place for comments that add context and intention. But their point was that it’s pointless to write a comment that just repeats what you’re doing with the code:

# increment wibble by 1

wibble += 1

Not only is it pointless, but there’s a danger that you’ll forget to update the comments when you update the code, and they end up being misleading. The ideal is to strive to make your code so readable, to use such good variable names and function names, and to structure it so well that you no longer need any comments to explain *what* the code is doing. Just a few here and there to explain *why*.

There are other places where comments are very useful. We’ll see that Django uses them a lot in the files it generates for us to use as a way of suggesting helpful bits of its API. And, of course, we use comments to explain the User Story in our functional tests—​by forcing us to make a coherent story out of the test, it makes sure we’re always testing from the point of view of the user.

There is more fun to be had in this area, things like *Behaviour-Driven Development* (see [[appendix\_bdd]](http://www.obeythetestinggoat.com/book/appendix_bdd.html)) and testing DSLs, but they’re topics for other books.

You’ll notice that, apart from writing the test out as comments, I’ve updated the assert to look for the word “To-Do” instead of “Django”. That means we expect the test to fail now. Let’s try running it.

First, start up the server:

$ **python manage.py runserver**

And then, in another shell, run the tests:

$ **python functional\_tests.py**

Traceback (most recent call last):

File "functional\_tests.py", line 10, in <module>

assert 'To-Do' in browser.title

AssertionError

That’s what we call an *expected fail*, which is actually good news—not quite as good as a test that passes, but at least it’s failing for the right reason; we can have some confidence we’ve written the test correctly.

**The Python Standard Library’s unittest Module**

There are a couple of little annoyances we should probably deal with. Firstly, the message "AssertionError" isn’t very helpful—​it would be nice if the test told us what it actually found as the browser title. Also, it’s left a Firefox window hanging around the desktop, so it would be nice if that got cleared up for us automatically.

One option would be to use the second parameter to the assert keyword, something like:

assert 'To-Do' in browser.title, "Browser title was " + browser.title

And we could also use a try/finally to clean up the old Firefox window. But these sorts of problems are quite common in testing, and there are some ready-made solutions for us in the standard library’s unittest module. Let’s use that! In *functional\_tests.py*:

functional\_tests.py

from selenium import webdriver

import unittest

class NewVisitorTest(unittest.TestCase):

def setUp(self):

self.browser = webdriver.Firefox()

def tearDown(self):

self.browser.quit()

def test\_can\_start\_a\_list\_and\_retrieve\_it\_later(self):

# Edith has heard about a cool new online to-do app. She goes

# to check out its homepage

self.browser.get('http://localhost:8000')

# She notices the page title and header mention to-do lists

self.assertIn('To-Do', self.browser.title)

self.fail('Finish the test!')

# She is invited to enter a to-do item straight away

[...rest of comments as before]

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main(warnings='ignore')

You’ll probably notice a few things here:

|  |  |
| --- | --- |
|  | Tests are organised into classes, which inherit from unittest.TestCase. |
|  | The main body of the test is in a method called test\_can\_start\_​a\_list\_and\_retrieve\_it\_later. Any method whose name starts with test is a test method, and will be run by the test runner. You can have more than one test\_ method per class. Nice descriptive names for our test methods are a good idea too. |
|  | setUp and tearDown are special methods which get run before and after each test. I’m using them to start and stop our browser—​note that they’re a bit like a try/except, in that tearDown will run even if there’s an error during the test itself.[[1](http://www.obeythetestinggoat.com/book/chapter_02_unittest.html#_footnote_1)] No more Firefox windows left lying around! |
|  | We use self.assertIn instead of just assert to make our test assertions. unittest provides lots of helper functions like this to make test assertions, like assertEqual, assertTrue, assertFalse, and so on. You can find more in the [unittest documentation](http://docs.python.org/3/library/unittest.html). |
|  | self.fail just fails no matter what, producing the error message given. I’m using it as a reminder to finish the test. |
|  | Finally, we have the if \_\_name\_\_ == '\_\_main\_\_' clause (if you’ve not seen it before, that’s how a Python script checks if it’s been executed from the command line, rather than just imported by another script). We call unittest.main(), which launches the unittest test runner, which will automatically find test classes and methods in the file and run them. |
|  | warnings='ignore' suppresses a superfluous ResourceWarning which was being emitted at the time of writing. It may have disappeared by the time you read this; feel free to try removing it! |
|  | If you’ve read the Django testing documentation, you might have seen something called LiveServerTestCase, and are wondering whether we should use it now. Full points to you for reading the friendly manual! LiveServerTestCase is a bit too complicated for now, but I promise I’ll use it in a later chapter…​ |

Let’s try it!

$ **python functional\_tests.py**

F

======================================================================

FAIL: test\_can\_start\_a\_list\_and\_retrieve\_it\_later (\_\_main\_\_.NewVisitorTest)

---------------------------------------------------------------------

Traceback (most recent call last):

File "functional\_tests.py", line 18, in

test\_can\_start\_a\_list\_and\_retrieve\_it\_later

self.assertIn('To-Do', self.browser.title)

AssertionError: 'To-Do' not found in 'Welcome to Django'

---------------------------------------------------------------------

Ran 1 test in 1.747s

FAILED (failures=1)

That’s a bit nicer, isn’t it? It tidied up our Firefox window, it gives us a nicely formatted report of how many tests were run and how many failed, and the assertIn has given us a helpful error message with useful debugging info. Bonzer!

**Commit**

This is a good point to do a commit; it’s a nicely self-contained change. We’ve expanded our functional test to include comments that describe the task we’re setting ourselves, our minimum viable to-do list. We’ve also rewritten it to use the Python unittest module and its various testing helper functions.

Do a **git status**—that should assure you that the only file that has changed is *functional\_tests.py*. Then do a **git diff**, which shows you the difference between the last commit and what’s currently on disk. That should tell you that *functional\_tests.py* has changed quite substantially:

$ **git diff**

diff --git a/functional\_tests.py b/functional\_tests.py

index d333591..b0f22dc 100644

--- a/functional\_tests.py

+++ b/functional\_tests.py

@@ -1,6 +1,45 @@

from selenium import webdriver

+import unittest

-browser = webdriver.Firefox()

-browser.get('http://localhost:8000')

+class NewVisitorTest(unittest.TestCase):

-assert 'Django' in browser.title

+ def setUp(self):

+ self.browser = webdriver.Firefox()

+

+ def tearDown(self):

+ self.browser.quit()

[...]

Now let’s do a:

$ **git commit -a**

The -a means “automatically add any changes to tracked files” (i.e., any files that we’ve committed before). It won’t add any brand new files (you have to explicitly git add them yourself), but often, as in this case, there aren’t any new files, so it’s a useful shortcut.

When the editor pops up, add a descriptive commit message, like “First FT specced out in comments, and now uses unittest.”

Now we’re in an excellent position to start writing some real code for our lists app. Read on!

Useful TDD Concepts

User Story

A description of how the application will work from the point of view of the user. Used to structure a functional test.

Expected failure

When a test fails in the way that we expected it to.

[1](http://www.obeythetestinggoat.com/book/chapter_02_unittest.html#_footnoteref_1). The only exception is if you have an exception inside setUp, then tearDown doesn’t run.

Last updated 2017-11-21 06:25:12 GMT

>note: Completed this chapter

>VS17: copied code from /note/functional\_test.py into the /vs17/functional\_test.py file from the VS17 project. Then I could run the tests against the DJANGO start in /note/