**Improving Functional Tests: Ensuring Isolation and Removing Voodoo Sleeps**

<http://www.obeythetestinggoat.com/book/chapter_explicit_waits_1.html>

Before we dive in and fix our real problem, let’s take care of a couple of housekeeping items. At the end of the last chapter, we made a note that different test runs were interfering with each other, so we’ll fix that. I’m also not happy with all these time.sleeps peppered through the code; they seem a bit unscientific, so we’ll replace them with something more reliable.

* *Clean up after FT runs*
* *Remove time.sleeps*

Both of these changes will be moving towards testing "best practices", making our tests more deterministic and more reliable.

**Ensuring Test Isolation in Functional Tests**

We ended the last chapter with a classic testing problem: how to ensure *isolation* between tests. Each run of our functional tests was leaving list items lying around in the database, and that would interfere with the test results when you next ran the tests.

When we run *unit* tests, the Django test runner automatically creates a brand new test database (separate from the real one), which it can safely reset before each individual test is run, and then throw away at the end. But our functional tests currently run against the "real" database, *db.sqlite3*.

One way to tackle this would be to "roll our own" solution, and add some code to *functional\_tests.py* which would do the cleaning up. The setUp and tearDown methods are perfect for this sort of thing.

Since Django 1.4 though, there’s a new class called LiveServerTestCase which can do this work for you. It will automatically create a test database (just like in a unit test run), and start up a development server for the functional tests to run against. Although as a tool it has some limitations which we’ll need to work around later, it’s dead useful at this stage, so let’s check it out.

LiveServerTestCase expects to be run by the Django test runner using *manage.py*. As of Django 1.6, the test runner will find any files whose name begins with *test*. To keep things neat and tidy, let’s make a folder for our functional tests, so that it looks a bit like an app. All Django needs is for it to be a valid Python package directory (i.e., one with a *\_\_\_init\_\_\_.py* in it):

$ **mkdir functional\_tests**

$ **touch functional\_tests/\_\_init\_\_.py**

Then we *move* our functional tests, from being a standalone file called *functional\_tests.py*, to being the *tests.py* of the functional\_tests app. We use **git mv** so that Git notices that we’ve moved the file:

$ **git mv functional\_tests.py functional\_tests/tests.py**

$ **git status** # shows the rename to functional\_tests/tests.py and \_\_init\_\_.py

At this point your directory tree should look like this:

.

├── db.sqlite3

├── functional\_tests

│   ├── \_\_init\_\_.py

│   └── tests.py

├── lists

│   ├── admin.py

│   ├── apps.py

│   ├── \_\_init\_\_.py

│   ├── migrations

│   │   ├── 0001\_initial.py

│   │   ├── 0002\_item\_text.py

│   │   ├── \_\_init\_\_.py

│   │   └── \_\_pycache\_\_

│   ├── models.py

│   ├── \_\_pycache\_\_

│   ├── templates

│   │   └── home.html

│   ├── tests.py

│   └── views.py

├── manage.py

├── superlists

│ ├── \_\_init\_\_.py

│ ├── \_\_pycache\_\_

│ ├── settings.py

│ ├── urls.py

│ └── wsgi.py

└── virtualenv

├── [...]

*functional\_tests.py* is gone, and has turned into *functional\_tests/tests.py*. Now, whenever we want to run our functional tests, instead of running python functional\_tests.py, we will use python manage.py test functional\_tests.

|  |  |
| --- | --- |
|  | You could mix your functional tests into the tests for the lists app. I tend to prefer to keep them separate, because functional tests usually have cross-cutting concerns that run across different apps. FTs are meant to see things from the point of view of your users, and your users don’t care about how you’ve split work between different apps! |

Now let’s edit *functional\_tests/tests.py* and change our NewVisitorTest class to make it use LiveServerTestCase:

functional\_tests/tests.py (ch06l001)

from django.test import LiveServerTestCase

from selenium import webdriver

from selenium.webdriver.common.keys import Keys

import time

class NewVisitorTest(LiveServerTestCase):

def setUp(self):

[...]

Next, instead of hardcoding the visit to localhost port 8000, LiveServerTestCase gives us an attribute called live\_server\_url:

functional\_tests/tests.py (ch06l002)

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(self.live\_server\_url)

We can also remove the if \_\_name\_\_ == '\_\_main\_\_' from the end if we want, since we’ll be using the Django test runner to launch the FT.

Now we are able to run our functional tests using the Django test runner, by telling it to run just the tests for our new functional\_tests app:

$ **python manage.py test functional\_tests**

Creating test database for alias 'default'...

F

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

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

(functional\_tests.tests.NewVisitorTest)

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

Traceback (most recent call last):

File "...python-tdd-book/functional\_tests/tests.py", line 65, in

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

self.fail('Finish the test!')

AssertionError: Finish the test!

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

Ran 1 test in 6.578s

FAILED (failures=1)

System check identified no issues (0 silenced).

Destroying test database for alias 'default'...

The FT gets through to the self.fail, just like it did before the refactor. You’ll also notice that if you run the tests a second time, there aren’t any old list items lying around from the previous test—​it has cleaned up after itself. Success! We should commit it as an atomic change:

$ **git status** # functional\_tests.py renamed + modified, new \_\_init\_\_.py

$ **git add functional\_tests**

$ **git diff --staged -M**

$ **git commit** # msg eg "make functional\_tests an app, use LiveServerTestCase"

The -M flag on the git diff is a useful one. It means "detect moves", so it will notice that *functional\_tests.py* and *functional\_tests/tests.py* are the same file, and show you a more sensible diff (try it without the flag!).

**Running Just the Unit Tests**

Now if we run manage.py test, Django will run both the functional and the unit tests:

$ **python manage.py test**

Creating test database for alias 'default'...

......F

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

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

[...]

AssertionError: Finish the test!

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

Ran 7 tests in 6.732s

FAILED (failures=1)

In order to run just the unit tests, we can specify that we want to only run the tests for the lists app:

$ **python manage.py test lists**

Creating test database for alias 'default'...

......

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

Ran 6 tests in 0.009s

OK

System check identified no issues (0 silenced).

Destroying test database for alias 'default'...

Useful Commands Updated

To run the functional tests

**python manage.py test functional\_tests**

To run the unit tests

**python manage.py test lists**

What to do if I say "run the tests", and you’re not sure which ones I mean? Have another look at the flowchart at the end of [[chapter\_philosophy\_and\_refactoring]](http://www.obeythetestinggoat.com/book/chapter_philosophy_and_refactoring.html), and try to figure out where we are. As a rule of thumb, we usually only run the functional tests once all the unit tests are passing, so if in doubt, try both!

**Aside: Upgrading Selenium and Geckodriver**

As I was running through this chapter again today, I found the FTs hung when I tried to run them.

It turns out that Firefox had auto-updated itself overnight, and my versions of Selenium and Geckodriver needed upgrading too. A quick visit to the [geckodriver releases page](https://github.com/mozilla/geckodriver/releases) confirmed there was a new version out. So a few downloads and upgrades were in order:

* A quick pip install --upgrade selenium first.
* Then a quick download of the new geckodriver.
* I saved a backup copy of the old one somewhere, and put the new one in its place somewhere on the PATH.
* And a quick check with geckodriver --version confirms the new one was ready to go.

The FTs were then back to running the way I expected them to.

There was no particular reason that it happened at this point in the book; indeed, it’s quite unlikely that it’ll happen right now for you, but it may happen at some point, and this seemed as good a place as any to talk about it, since we’re doing some housekeeping.

It’s one of the things you have to put up with when using Selenium. Although it is possible to pin your browser and Selenium versions (on a CI server, for example), browser versions don’t stand still out in the real world, and you need to keep up with what your users have.

|  |  |
| --- | --- |
|  | If something strange is going on with your FTs, it’s always worth trying to upgrade Selenium. |

Back to our regular programming now.

**On Implicit and Explicit Waits, and Voodoo time.sleeps**

Let’s talk about the time.sleep in our FT:

functional\_tests/tests.py

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

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

inputbox.send\_keys(Keys.ENTER)

time.sleep(1)

self.check\_for\_row\_in\_list\_table('1: Buy peacock feathers')

This is what’s called an "explicit wait". That’s by contrast with "implicit waits": in certain cases, Selenium tries to wait "automatically" for you when it thinks the page is loading. It even provides a method called implicitly\_wait that lets you control how long it will wait if you ask it for an element that doesn’t seem to be on the page yet.

In fact, in the first edition, I was able to rely entirely on implicit waits. The problem is that implicit waits are always a little flakey, and with the release of Selenium 3, implicit waits became even more unreliable. At the same time, the general opinion from the Selenium team was that implicit waits were just a bad idea, and to be avoided.

So this edition has explicit waits from the very beginning. But the problem is that those time.sleeps have their own issues. Currently we’re waiting for one second, but who’s to say that’s the right amount of time? For most tests we run against our own machine, one second is way too long, and it’s going to really slow down our FT runs. 0.1s would be fine. But the problem is that if you set it that low, every so often you’re going to get a spurious failure because, for whatever reason, the laptop was being a bit slow just then. And even at 1s you can never be quite sure you’re not going to get random failures that don’t indicate a real problem, and false positives in tests are a real annoyance (there’s lots more on this in [an article by Martin Fowler](https://martinfowler.com/articles/nonDeterminism.html)).

|  |  |
| --- | --- |
|  | Unexpected NoSuchElementException and StaleElementException errors are the usual symptoms of forgetting an explicit wait. Try removing the time.sleep and see if you get one. |

So let’s replace our sleeps with a tool that will wait for just as long as is needed, up to a nice long timeout to catch any glitches. We’ll rename check\_for\_row\_in\_list\_table to wait\_for\_row\_in\_list\_table, and add some polling/retry logic to it:

functional\_tests/tests.py (ch06l004)

from selenium.common.exceptions import WebDriverException

MAX\_WAIT = 10

[...]

def wait\_for\_row\_in\_list\_table(self, row\_text):

start\_time = time.time()

while True:

try:

table = self.browser.find\_element\_by\_id('id\_list\_table')

rows = table.find\_elements\_by\_tag\_name('tr')

self.assertIn(row\_text, [row.text for row in rows])

return

except (AssertionError, WebDriverException) as e:

if time.time() - start\_time > MAX\_WAIT:

raise e

time.sleep(0.5)

|  |  |
| --- | --- |
|  | We’ll use a constant called MAX\_WAIT to set the maximum amount of time we’re prepared to wait. 10 seconds should be more than enough to catch any glitches or random slowness. |
|  | Here’s the loop, which will keep going forever, unless we get to one of two possible exit routes. |
|  | Here are our three lines of assertions from the old version of the method. |
|  | If we get through them and our assertion passes, we return from the function and escape the loop. |
|  | But if we catch an exception, we wait a short amount of time and loop around to retry. There are two types of exceptions we want to catch: WebDriverException for when the page hasn’t loaded and Selenium can’t find the table element on the page, and AssertionError for when the table is there, but it’s perhaps a table from before the page reloads, so it doesn’t have our row in yet. |
|  | Here’s our second escape route. If we get to this point, that means our code kept raising exceptions every time we tried it until we exceeded our timeout. So this time, we re-raise the exception and let it bubble up to our test, and most likely end up in our traceback, telling us why the test failed. |

Are you thinking this code is a little ugly, and makes it a bit harder to see exactly what we’re doing? I agree. [Later on](http://www.obeythetestinggoat.com/book/chapter_organising_test_files.html#self.wait-for), we’ll refactor out a general wait\_for helper, to separate the timing and re-raising logic from the test assertions. But we’ll wait until we need it in multiple places.

|  |  |
| --- | --- |
|  | If you’ve used Selenium before, you may know that it has a few [helper functions to do waits](http://www.seleniumhq.org/docs/04_webdriver_advanced.jsp). I’m not a big fan of them. Over the course of the book we’ll build a couple of wait helper tools which I think will make for nice, readable code, but of course you should check out the homegrown Selenium waits in your own time, and see what you think of them. |

Now we can rename our method calls, and remove the voodoo time.sleeps:

functional\_tests/tests.py (ch06l005)

[...]

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

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

inputbox.send\_keys(Keys.ENTER)

self.wait\_for\_row\_in\_list\_table('1: Buy peacock feathers')

# 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)

inputbox = self.browser.find\_element\_by\_id('id\_new\_item')

inputbox.send\_keys('Use peacock feathers to make a fly')

inputbox.send\_keys(Keys.ENTER)

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

self.wait\_for\_row\_in\_list\_table('2: Use peacock feathers to make a fly')

self.wait\_for\_row\_in\_list\_table('1: Buy peacock feathers')

[...]

And rerun the tests:

$ **python manage.py test**

Creating test database for alias 'default'...

......F

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

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

(functional\_tests.tests.NewVisitorTest)

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

Traceback (most recent call last):

File "...python-tdd-book/functional\_tests/tests.py", line 73, in

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

self.fail('Finish the test!')

AssertionError: Finish the test!

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

Ran 7 tests in 4.552s

FAILED (failures=1)

System check identified no issues (0 silenced).

Destroying test database for alias 'default'...

We get to the same place, and notice we’ve shaved a couple of seconds off the execution time too. That might not seem like a lot right now, but it all adds up.

Just to check we’ve done the right thing, let’s deliberately break the test in a couple of ways and see some errors. First let’s check that if we look for some row text that will never appear, we get the right error:

functional\_tests/tests.py (ch06l006)

rows = table.find\_elements\_by\_tag\_name('tr')

self.assertIn('foo', [row.text for row in rows])

return

We see we still get a nice self-explanatory test failure message:

self.assertIn('foo', [row.text for row in rows])

AssertionError: 'foo' not found in ['1: Buy peacock feathers']

Let’s put that back the way it was and break something else:

functional\_tests/tests.py (ch06l007)

try:

table = self.browser.find\_element\_by\_id('id\_nothing')

rows = table.find\_elements\_by\_tag\_name('tr')

self.assertIn(row\_text, [row.text for row in rows])

return

[...]

Sure enough, we get the errors for when the page doesn’t contain the element we’re looking for too:

selenium.common.exceptions.NoSuchElementException: Message: Unable to locate

element: [id="id\_nothing"]

Everything seems to be in order. Let’s put our code back to way it should be, and do one final test run:

$ **python manage.py test**

[...]

AssertionError: Finish the test!

Great. With that little interlude over, let’s crack on with getting our application actually working for multiple lists.

Testing "Best Practices" Applied in this Chapter

Ensuring test isolation and managing global state

Different tests shouldn’t affect one another. This means we need to reset any permanent state at the end of each test. Django’s test runner helps us do this by creating a test database, which it wipes clean in between each test. (See also [[chapter\_purist\_unit\_tests]](http://www.obeythetestinggoat.com/book/chapter_purist_unit_tests.html).)

Avoid "voodoo" sleeps

Whenever we need to wait for something to load, it’s always tempting to throw in a quick-and-dirty time.sleep. But the problem is that the length of time we wait is always a bit of a shot in the dark, either too short and vulnerable to spurious failures, or too long and it’ll slow down our test runs. Prefer a retry loop that polls our app and moves on as soon as possible.

Don’t rely on Selenium’s implicit waits

Selenium does theoretically do some "implicit" waits, but the implementation varies between browsers, and at the time of writing was highly unreliable in the Selenium 3 Firefox driver. "Explicit is better than implict", as the Zen of Python says, so prefer explicit waits.

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