Table of Contents

Improving Functional Tests: Ensuring Isolation and Removing Magic Sleeps

Before we dive in and fix our single-global-list 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.sleep`s 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 us 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 (FTs) left list items lying around in the database, and that interfered with the test results when next running 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 thrown away at the end. But our FTs 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.

But as this is a common problem, Django supplies a test class called `LiveServerTestCase` that addresses this issue. It will automatically create a test database (just like in a unit test run) and start up a development server for the FTs 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*, which will run tests from any files whose name begins with *test_*. To keep things neat and tidy, let's make a folder for our FTs, 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
```

Now we want to *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 keeps track of the fact that this is the same file and should have a single history.

```
$ 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
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations
│   │   ├── 0001_initial.py
│   │   ├── 0002_item_text.py
│   │   └── __init__.py
│   ├── models.py
│   ├── templates
│   │   └── home.html
│   ├── tests.py
│   └── views.py
├── manage.py
└── superlists
    ├── __init__.py
    ├── asgi.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

functional_tests.py is gone, and has turned into *functional_tests/tests.py*. Now, whenever we want to run our FTs, 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 keeping them separate, because FTs 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
[...]


class NewVisitorTest(LiveServerTestCase):
    def setUp(self):
        [...]
```
PYTHON

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_todo_list(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)
```
PYTHON

We can also remove the `if __name__ == '__main__'` from the end if we want, as 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'...
Found 1 test(s).
System check identified no issues (0 silenced).
.
 ----------------------------------------------------------------------
Ran 1 test in 10.519s

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

When I ran this test today, I ran into the Firefox upgrade pop-up. Just a little reminder, in case you happen to see it too, we talked about it in [chapter_01] in a little sidebar.

The FT still passes, reassuring us that our refactor didn't break anything. 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
$ git commit  # msg eg "make functional_tests an app, use LiveServerTestCase"
```

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'...
Found 8 test(s).
System check identified no issues (0 silenced).
........
 ----------------------------------------------------------------------
Ran 8 tests in 10.859s

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

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'...
Found 7 test(s).
System check identified no issues (0 silenced).
.......
 ----------------------------------------------------------------------
Ran 7 tests in 0.009s

OK
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_04_philosophy_and_refactoring], and try to figure out where we are. As a rule of thumb, we usually only run the FTs once all the unit tests are passing, so if in doubt, try both!

On Implicit and Explicit Waits, and Magic time.sleeps

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

functional_tests/tests.py

```
PYTHON
        # 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 in 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 of this book, 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 4, implicit waits were disabled by default. At the same time, the general opinion from the Selenium team is that implicit waits are just a bad idea, and should be avoided (https://www.selenium.dev/documentation/webdriver/waits).

So this edition has explicit waits from the very beginning. But the problem is that those `time.sleep` s 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 one second, there's still a chance of random failures that don't indicate a real problem—and false positives in tests are a real annoyance.[1]

> Unexpected `NoSuchElementException` and `StaleElementException` errors are often a sign that you need an explicit wait.

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)

```
PYTHON
[...]
from selenium.common.exceptions import WebDriverException
import time

MAX_WAIT = 5   1


class NewVisitorTest(LiveServerTestCase):
    def setUp(self):
        [...]
    def tearDown(self):
        [...]

    def wait_for_row_in_list_table(self, row_text):
        start_time = time.time()
        while True:   2
            try:
                table = self.browser.find_element(By.ID, "id_list_table")   3
                rows = table.find_elements(By.TAG_NAME, "tr")
                self.assertIn(row_text, [row.text for row in rows])
                return   4
            except (AssertionError, WebDriverException):   5
                if time.time() - start_time > MAX_WAIT:   6
                    raise   6
                time.sleep(0.5)   5
```

1 We'll use a constant called `MAX_WAIT` to set the maximum amount of time we're prepared to wait. Five seconds should be enough to catch any glitches or random slowness.

2 Here's the loop, which will keep going forever, unless we get to one of two possible exit routes.

3 Here are our three lines of assertions from the old version of the method.

4 If we get through them, and our assertion passes, we return from the function and escape the loop.

5 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.

6 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 reraise 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 ([self.wait-for]), we'll refactor out a general `wait_for` helper, to separate the timing and reraising 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 conduct waits (https://www.selenium.dev/documentation/webdriver/waits/#explicit-waits). I'm not a big fan of them, though not for any objective reason really. Over the course of the book, we'll build a couple of wait helper tools, which I think will make for nice and readable code. But of course you should check out the homegrown Selenium waits in your own time, and see if you prefer them.

Now we can rename our method calls, and remove the magic `time.sleep` s:

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")
        [...]
```

PYTHON

And rerun the tests:

```
$ python manage.py test
Creating test database for alias 'default'...
Found 8 test(s).
System check identified no issues (0 silenced).
........
 ----------------------------------------------------------------------
Ran 8 tests in 4.552s

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

Hooray we're back to passing, and notice we've shaved a few 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 try searching for some text that we know isn't there, and check that we get the expected error:

functional_tests/tests.py (ch06l006)

```
    def wait_for_row_in_list_table(self, row_text):
        [...]
            rows = table.find_elements(By.TAG_NAME, "tr")
            self.assertIn("foo", [row.text for row in rows])
            return
```
PYTHON

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']
```

> Did you get a bit bored waiting five seconds for the test to fail? That's one of the downsides of explicit waits. There's a tricky trade-off between waiting long enough that little glitches don't throw you, versus waiting so long that expected failures are painfully slow to watch. Making `MAX_WAIT` configurable so that it's fast in local dev, but more conservative on continuous integration (CI) servers can be a good idea. See [chapter_25_CI] for an introduction to CI.

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
        [...]
```
PYTHON

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"]; For documentation on this error, [...]
```

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

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

Great. With that little interlude over, let's crack on with getting our application actually working for multiple lists. Don't forget to commit first!

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.

Avoid "magic" 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 too 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 is not always reliable. "Explicit is better than implicit", as the Zen of Python says,[2] so prefer explicit waits.

1. There's lots more on this in an article by Martin Fowler (https://oreil.ly/YdRx-).
2. `python -c "import this"`

Comments

ALSO ON **OBEY THE TESTING GOAT!**

Splitting Our Tests into Multiple Files, and a …

9 years ago • 22 comments

As our first few users start using the site, we've noticed they sometimes make …

Making our App Production-Ready

2 years ago • 8 comments

Static files aren't the same kind of things as the dynamic content that comes from …

Obey the Testing Goat!

8 years ago • 12 comments

I'm still not great at selfies... Here you get two for the price of one tho. The second …

Getting A For Deplo

a year ago • 1

With a PaaS your own se renting a "se

Emoji?

8 Responses

Upvote Funny Love Surprised Angry Sad

1 Comment Kgotso Koete ▾

Join the discussion…

♡ **Share** **Best** **Newest** **Oldest**

Florian Bautry

2 years ago

We don't need to do the upgrade of geckodriver anymore, so this lines can be removes:
"- 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."

1 0 Reply