**Testing a Simple Home Page with Unit Tests**

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

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

Open: -- R-Click > GIT-bash

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

We finished the last chapter with a functional test failing, telling us that it wanted the home page for our site to have “To-Do” in its title. It’s time to start working on our application.

Warning: Things Are About to Get Real

The first two chapters were intentionally nice and light. From now on, we get into some more meaty coding. Here’s a prediction: at some point, things are going to go wrong. You’re going to see different results from what I say you should see. This is a Good Thing, because it will be a genuine character-building Learning Experience™.

One possibility is that I’ve given some ambiguous explanations, and you’ve done something different from what I intended. Step back and have a think about what we’re trying to achieve at this point in the book. Which file are we editing, what do we want the user to be able to do, what are we testing and why? It may be that you’ve edited the wrong file or function, or are running the wrong tests. I reckon you’ll learn more about TDD from these "stop and think" moments than you do from all the bits where the following instructions and copy-pasting goes smoothly.

Or it may be a real bug. Be tenacious, read the error message carefully (see [Reading Tracebacks](http://www.obeythetestinggoat.com/book/chapter_unit_test_first_view.html#read_tracebacks_aside) a little later on in the chapter), and you’ll get to the bottom of it. It’s probably just a missing comma, or trailing slash, or maybe a missing *s* in one of the Selenium find methods. But, as [Zed Shaw put it so well](http://www.obeythetestinggoat.com/book/bibliography.html#lpthw), this kind of debugging is also an vital part of learning, so do stick it out!

You can always drop me an email (or try the [Google Group](https://groups.google.com/forum/#!forum/obey-the-testing-goat-book)) if you get really stuck. Happy debugging

**Our First Django App, and Our First Unit Test**

Django encourages you to structure your code into *apps*: the theory is that one project can have many apps, you can use third-party apps developed by other people, and you might even reuse one of your own apps in a different project…​although I admit I’ve never actually managed it myself! Still, apps are a good way to keep your code organised.

Let’s start an app for our to-do lists:

$ **python manage.py startapp lists**

That will create a folder at *superlists/lists*, next to *superlists/superlists*, and within it a few placeholder files for things like models, views, and, of immediate interest to us, tests:

tddBook/superlists

.

├── db.sqlite3

├── functional\_tests.py

├── lists

│   ├── admin.py

│   ├── apps.py

│   ├── \_\_init\_\_.py

│   ├── migrations

│   │   └── \_\_init\_\_.py

│   ├── models.py

│   ├── tests.py

│   └── views.py

├── manage.py

└── superlists

├── \_\_init\_\_.py

├── \_\_pycache\_\_

├── settings.py

├── urls.py

└── wsgi.py

Pub/tdd/note/

.

├── db.sqlite3

├── functional\_tests.py

├── geckodriver.log (can be removed)

├── manage.py\*

├── superlists/

├── \_\_init\_\_.py

├── \_\_pycache\_\_

├── settings.py

├── urls.py

└── wsgi.py

├── venvSuperlists/

├──(content not shown !)

**Unit Tests, and How They Differ from Functional Tests**

As with so many of the labels we put on things, the line between unit tests and functional tests can become a little blurry at times. The basic distinction, though, is that functional tests test the application from the outside, from the point of view of the user. Unit tests test the application from the inside, from the point of view of the programmer.

The TDD approach I’m following wants our application to be covered by both types of test. Our workflow will look a bit like this:

1. We start by writing a *functional test*, describing the new functionality from the user’s point of view.
2. Once we have a functional test that fails, we start to think about how to write code that can get it to pass (or at least to get past its current failure). We now use one or more *unit tests* to define how we want our code to behave—​the idea is that each line of production code we write should be tested by (at least) one of our unit tests.
3. Once we have a failing unit test, we write the smallest amount of *application code* we can, just enough to get the unit test to pass. We may iterate between steps 2 and 3 a few times, until we think the functional test will get a little further.
4. Now we can rerun our functional tests and see if they pass, or get a little further. That may prompt us to write some new unit tests, and some new code, and so on.

You can see that, all the way through, the functional tests are driving what development we do from a high level, while the unit tests drive what we do at a low level.

Does that seem slightly redundant? Sometimes it can feel that way, but functional tests and unit tests do really have very different objectives, and they will usually end up looking quite different.

|  |  |
| --- | --- |
|  | Functional tests should help you build an application with the right functionality, and guarantee you never accidentally break it. Unit tests should help you to write code that’s clean and bug free. |

Enough theory for now—let’s see how it looks in practice.

**Unit Testing in Django**

Let’s see how to write a unit test for our home page view. Open up the new file at *lists/tests.py*, and you’ll see something like this:

lists/tests.py

from django.test import TestCase

# Create your tests here.

Django has helpfully suggested we use a special version of TestCase, which it provides. It’s an augmented version of the standard unittest.TestCase, with some additional Django-specific features, which we’ll discover over the next few chapters.

You’ve already seen that the TDD cycle involves starting with a test that fails, then writing code to get it to pass. Well, before we can even get that far, we want to know that the unit test we’re writing will definitely be run by our automated test runner, whatever it is. In the case of *functional\_tests.py*, we’re running it directly, but this file made by Django is a bit more like magic. So, just to make sure, let’s make a deliberately silly failing test:

lists/tests.py

from django.test import TestCase

class SmokeTest(TestCase):

def test\_bad\_maths(self):

self.assertEqual(1 + 1, 3)

Now let’s invoke this mysterious Django test runner. Theere is no need to start the Django dev-server. As usual, it’s a *manage.py* command:

$ **python manage.py test**

Creating test database for alias 'default'...

F

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

FAIL: test\_bad\_maths (lists.tests.SmokeTest)

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

Traceback (most recent call last):

File "...python-tdd-book/lists/tests.py", line 6, in test\_bad\_maths

self.assertEqual(1 + 1, 3)

AssertionError: 2 != 3

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

Ran 1 test in 0.001s

FAILED (failures=1)

System check identified no issues (0 silenced).

Destroying test database for alias 'default'...

Excellent. The machinery seems to be working. This is a good point for a commit:

$ **git status** # should show you lists/ is untracked

$ **git add lists**

$ **git diff --staged** # will show you the diff that you're about to commit

$ **git commit -m "Add app for lists, with deliberately failing unit test"**

As you’ve no doubt guessed, the -m flag lets you pass in a commit message at the command line, so you don’t need to use an editor. It’s up to you to pick the way you like to use the Git command line; I’ll just show you the main ones I’ve seen used. The key rule is: *make sure you always review what you’re about to commit before you do it*.

**Django’s MVC, URLs, and View Functions**

Django is structured along a classic *Model-View-Controller* (MVC) pattern. Well, *broadly*. It definitely does have models, but its views are more like a controller, and it’s the templates that are actually the view part, but the general idea is there. If you’re interested, you can look up the finer points of the discussion [in the Django FAQs](https://docs.djangoproject.com/en/1.11/faq/general/).

Irrespective of any of that, as with any web server, Django’s main job is to decide what to do when a user asks for a particular URL on our site. Django’s workflow goes something like this:

1. An HTTP *request* comes in for a particular *URL*.
2. Django uses some rules to decide which *view* function should deal with the request (this is referred to as *resolving* the URL).
3. The view function processes the request and returns an HTTP *response*.

So we want to test two things:

* Can we resolve the URL for the root of the site (“/”) to a particular view function we’ve made?
* Can we make this view function return some HTML which will get the functional test to pass?

Let’s start with the first. Open up *lists/tests.py*, and change our silly test to something like this:

lists/tests.py

from django.urls import resolve

from django.test import TestCase

from lists.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)

What’s going on here?

|  |  |
| --- | --- |
|  | resolve is the function Django uses internally to resolve URLs and find what view function they should map to. We’re checking that resolve, when called with “/”, the root of the site, finds a function called home\_page. |
|  | What function is that? It’s the view function we’re going to write next, which will actually return the HTML we want. You can see from the import that we’re planning to store it in *lists/views.py*. |

So, what do you think will happen when we run the tests?

$ **python manage.py test**

ImportError: cannot import name 'home\_page'

It’s a very predictable and uninteresting error: we tried to import something we haven’t even written yet. But it’s still good news—​for the purposes of TDD, an exception which was predicted counts as an expected failure. Since we have both a failing functional test and a failing unit test, we have the Testing Goat’s full blessing to code away.

**At Last! We Actually Write Some Application Code!**

It is exciting, isn’t it? Be warned, TDD means that long periods of anticipation are only defused very gradually, and by tiny increments. Especially since we’re learning and only just starting out, we only allow ourselves to change (or add) one line of code at a time—​and each time, we make just the minimal change required to address the current test failure.

I’m being deliberately extreme here, but what’s our current test failure? We can’t import home\_page from lists.views? OK, let’s fix that—​and only that. In *lists/views.py*:

lists/views.py

from django.shortcuts import render

# Create your views here.

home\_page = None

*"You must be joking!"* I can hear you say.

I can hear you because it’s what I used to say (with feeling) when my colleagues first demonstrated TDD to me. Well, bear with me, and we’ll talk about whether or not this is all taking it too far in a little while. But for now, let yourself follow along, even if it’s with some exasperation, and see if our tests can help us write the correct code, one tiny step at a time.

We run the tests again:

$ **python manage.py test**

Creating test database for alias 'default'...

E

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

ERROR: test\_root\_url\_resolves\_to\_home\_page\_view (lists.tests.HomePageTest)

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

Traceback (most recent call last):

File "...python-tdd-book/lists/tests.py", line 8, in

test\_root\_url\_resolves\_to\_home\_page\_view

found = resolve('/')

File ".../django/urls/base.py", line 27, in resolve

return get\_resolver(urlconf).resolve(path)

File ".../django/urls/resolvers.py", line 392, in resolve

raise Resolver404({'tried': tried, 'path': new\_path})

django.urls.exceptions.Resolver404: {'tried': [[<RegexURLResolver

<RegexURLPattern list> (admin:admin) ^admin/>]], 'path': ''}

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

Ran 1 test in 0.002s

FAILED (errors=1)

System check identified no issues (0 silenced).

Destroying test database for alias 'default'...

Reading Tracebacks

Let’s spend a moment talking about how to read tracebacks, since it’s something we have to do a lot in TDD. You soon learn to scan through them and pick up relevant clues:

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

ERROR: test\_root\_url\_resolves\_to\_home\_page\_view (lists.tests.HomePageTest)

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

Traceback (most recent call last):

File "...python-tdd-book/lists/tests.py", line 8, in

test\_root\_url\_resolves\_to\_home\_page\_view

found = resolve('/')

File ".../django/urls/base.py", line 27, in resolve

return get\_resolver(urlconf).resolve(path)

File ".../django/urls/resolvers.py", line 392, in resolve

raise Resolver404({'tried': tried, 'path': new\_path})

django.urls.exceptions.Resolver404: {'tried': [[<RegexURLResolver

<RegexURLPattern list> (admin:admin) ^admin/>]], 'path': ''}

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

[...]

|  |  |
| --- | --- |
|  | The first place you look is usually *the error itself*. Sometimes that’s all you need to see, and it will let you identify the problem immediately. But sometimes, like in this case, it’s not quite self-evident. |
|  | The next thing to double-check is: *which test is failing?* Is it definitely the one we expected—​that is, the one we just wrote? In this case, the answer is yes. |
|  | Then we look for the place in *our test code* that kicked off the failure. We work our way down from the top of the traceback, looking for the filename of the tests file, to check which test function, and what line of code, the failure is coming from. In this case it’s the line where we call the resolve function for the "/" URL. |

There is ordinarily a fourth step, where we look further down for any of *our own application code* which was involved with the problem. In this case it’s all Django code, but we’ll see plenty of examples of this fourth step later in the book.

Pulling it all together, we interpret the traceback as telling us that, when trying to resolve “/”, Django raised a 404 error—​in other words, Django can’t find a URL mapping for “/”. Let’s help it out.

**urls.py**

Our tests are telling us that we need a URL mapping. Django uses a file called *urls.py* to map URLs to view functions. There’s a main *urls.py* for the whole site in the *superlists/superlists* folder. Let’s go take a look:

superlists/urls.py

"""superlists URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:

https://docs.djangoproject.com/en/1.11/topics/http/urls/

Examples:

Function views

1. Add an import: from my\_app import views

2. Add a URL to urlpatterns: url(r'^$', views.home, name='home')

Class-based views

1. Add an import: from other\_app.views import Home

2. Add a URL to urlpatterns: url(r'^$', Home.as\_view(), name='home')

Including another URLconf

1. Import the include() function: from django.conf.urls import url, include

2. Add a URL to urlpatterns: url(r'^blog/', include('blog.urls'))

"""

from django.conf.urls import url

from django.contrib import admin

urlpatterns = [

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

]

As usual, lots of helpful comments and default suggestions from Django.

A url entry starts with a regular expression that defines which URLs it applies to, and goes on to say where it should send those requests—​either to a view function you’ve imported, or maybe to another *urls.py* file somewhere else.

The first example entry has the regular expression ^$, which means an empty string—​could this be the same as the root of our site, which we’ve been testing with “/”? Let’s find out—​what happens if we include it?

|  |  |
| --- | --- |
|  | If you’ve never come across regular expressions, you can get away with just taking my word for it, for now—​but you should make a mental note to go learn about them. |

We’ll also get rid of the admin URL, because we won’t be using the Django admin site for now:

superlists/urls.py

from django.conf.urls import url

from lists import views

urlpatterns = [

url(r'^$', views.home\_page, name='home'),

]

Run the unit tests again, with **python manage.py test**:

[...]

TypeError: view must be a callable or a list/tuple in the case of include().

That’s progress! We’re no longer getting a 404.

The traceback is messy, but the message at the end is telling us what’s going on: the unit tests have actually made the link between the URL "/" and the home\_page = None in *lists/views.py*, and are now complaining that the home\_page view is not callable. And that gives us a justification for changing it from being None to being an actual function. Every single code change is driven by the tests!

Back in *lists/views.py*:

lists/views.py

from django.shortcuts import render

# Create your views here.

def home\_page():

pass

And now?

$ **python manage.py test**

Creating test database for alias 'default'...

.

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

Ran 1 test in 0.003s

OK

System check identified no issues (0 silenced).

Destroying test database for alias 'default'...

Hooray! Our first ever unit test pass! That’s so momentous that I think it’s worthy of a commit:

$ **git diff** # should show changes to urls.py, tests.py, and views.py

$ **git commit -am "First unit test and url mapping, dummy view"**

That was the last variation on git commit I’ll show, the a and m flags together, which adds all changes to tracked files and uses the commit message from the command line.

|  |  |
| --- | --- |
|  | git commit -am is the quickest formulation, but also gives you the least feedback about what’s being committed, so make sure you’ve done a git status and a git diff beforehand, and are clear on what changes are about to go in. |

**Unit Testing a View**

On to writing a test for our view, so that it can be something more than a do-nothing function, and instead be a function that returns a real response with HTML to the browser. Open up *lists/tests.py*, and add a new *test method*. I’ll explain each bit:

lists/tests.py

from django.urls import resolve

from django.test import TestCase

from django.http import HttpRequest

from lists.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)

html = response.content.decode('utf8')

self.assertTrue(html.startswith('<html>'))

self.assertIn('<title>To-Do lists</title>', html)

self.assertTrue(html.endswith('</html>'))

What’s going on in this new test?

|  |  |
| --- | --- |
|  | We create an HttpRequest object, which is what Django will see when a user’s browser asks for a page. |
|  | We pass it to our home\_page view, which gives us a response. You won’t be surprised to hear that this object is an instance of a class called HttpResponse. |
|  | Then, we extract the .content of the response. These are the raw bytes, the ones and zeros that would be sent down the wire to the user’s browser. We call .decode() to convert them into the string of HTML that’s being sent to the user. |
|  | We want it to start with an <html> tag which gets closed at the end. |
|  | And we want a <title> tag somewhere in the middle, with the words "To-Do lists" in it—​because that’s what we specified in our functional test. |

Once again, the unit test is driven by the functional test, but it’s also much closer to the actual code—​we’re thinking like programmers now.

Let’s run the unit tests now and see how we get on:

TypeError: home\_page() takes 0 positional arguments but 1 was given

**The Unit-Test/Code Cycle**

We can start to settle into the TDD *unit-test/code cycle* now:

1. In the terminal, run the unit tests and see how they fail.
2. In the editor, make a minimal code change to address the current test failure.

And repeat!

The more nervous we are about getting our code right, the smaller and more minimal we make each code change—​the idea is to be absolutely sure that each bit of code is justified by a test.

This may seem laborious, and at first, it will be. But once you get into the swing of things, you’ll find yourself coding quickly even if you take microscopic steps—​this is how we write all of our production code at work.

Let’s see how fast we can get this cycle going:

* Minimal code change:

lists/views.py

def home\_page(request):

pass

* Tests:
* html = response.content.decode('utf8')

AttributeError: 'NoneType' object has no attribute 'content'

* Code—​we use django.http.HttpResponse, as predicted:

lists/views.py

from django.http import HttpResponse

# Create your views here.

def home\_page(request):

return HttpResponse()

* Tests again:
* self.assertTrue(html.startswith('<html>'))

AssertionError: False is not true

* Code again:

lists/views.py

def home\_page(request):

return HttpResponse('<html>')

* Tests:

AssertionError: '<title>To-Do lists</title>' not found in '<html>'

* Code:

lists/views.py

def home\_page(request):

return HttpResponse('<html><title>To-Do lists</title>')

* Tests—​almost there?
* self.assertTrue(html.endswith('</html>'))

AssertionError: False is not true

* Come on, one last effort:

lists/views.py

def home\_page(request):

return HttpResponse('<html><title>To-Do lists</title></html>')

* Surely?
* $ **python manage.py test**
* Creating test database for alias 'default'...
* ..
* ---------------------------------------------------------------------
* Ran 2 tests in 0.001s
* OK
* System check identified no issues (0 silenced).

Destroying test database for alias 'default'...

Yes! Now, let’s run our functional tests. Don’t forget to spin up the dev server again, if it’s not still running. It feels like the final heat of the race here; surely this is it…​could it be?

$ **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 19, in

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

self.fail('Finish the test!')

AssertionError: Finish the test!

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

Ran 1 test in 1.609s

FAILED (failures=1)

Failed? What? Oh, it’s just our little reminder? Yes? Yes! We have a web page!

Ahem. Well, *I* thought it was a thrilling end to the chapter. You may still be a little baffled, perhaps keen to hear a justification for all these tests, and don’t worry, all that will come, but I hope you felt just a tinge of excitement near the end there.

Just a little commit to calm down, and reflect on what we’ve covered:

$ **git diff** # should show our new test in tests.py, and the view in views.py

$ **git commit -am "Basic view now returns minimal HTML"**

That was quite a chapter! Why not try typing git log, possibly using the --oneline flag, for a reminder of what we got up to:

$ **git log --oneline**

a6e6cc9 Basic view now returns minimal HTML

450c0f3 First unit test and url mapping, dummy view

ea2b037 Add app for lists, with deliberately failing unit test

[...]

Not bad—​we covered:

* Starting a Django app
* The Django unit test runner
* The difference between FTs and unit tests
* Django URL resolving and *urls.py*
* Django view functions, request and response objects
* And returning basic HTML

Useful Commands and Concepts

Running the Django dev server

**python manage.py runserver**

Running the functional tests

**python functional\_tests.py**

Running the unit tests

**python manage.py test**

The unit-test/code cycle

1. Run the unit tests in the terminal.
2. Make a minimal code change in the editor.
3. Repeat!

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