Table of Contents

Praise for *Test-Driven Development with Python*
Preface
Preface to the Third Edition: TDD in the Age of AI
Prerequisites and Assumptions
Acknowledgments
The Basics of TDD and Django
1. Getting Django Set Up Using a Functional Test
2. Extending Our Functional Test Using the unittest Module
3. Testing a Simple Home Page with Unit Tests
4. What Are We Doing with All These Tests? (And, Refactoring)
5. Saving User Input: Testing the Database
    5.1. Wiring Up Our Form to Send a POST Request
    5.2. Testing the Contract Between Frontend and Backend
    5.3. Debugging Functional Tests
    5.4. Processing a POST Request on the Server
    5.5. Passing Python Variables to Be Rendered in the Template
    5.6. Three Strikes and Refactor
    5.7. The Django ORM and Our First Model
    5.8. Saving the POST to the Database
    5.9. Redirect After a POST
    5.10. Better Unit Testing Practice: Each Test Should Test One Thing
    5.11. Rendering Items in the Template
    5.12. Creating Our Production Database with migrate
    5.13. Recap
6. Improving Functional Tests: Ensuring Isolation and Removing Magic Sleeps
7. Working Incrementally
8. Prettification: Layout and Styling, and What to Test About It
Going to Production
9. Containerization aka Docker
10. Making Our App Production-Ready
11. Getting a Server Ready for Deployment
12. Infrastructure as Code: Automated Deployments with Ansible
Forms and Validation
13. Splitting Our Tests into Multiple Files, and a Generic Wait Helper

14. Validation at the Database Layer
15. A Simple Form
16. More Advanced Forms
More Advanced Topics in Testing
17. A Gentle Excursion into JavaScript
18. Deploying Our New Code
19. User Authentication, Spiking, and De-Spiking
20. Using Mocks to Test External Dependencies
21. Using Mocks for Test Isolation
22. Test Fixtures and a Decorator for Explicit Waits
23. Debugging and Testing Server Issues
24. Finishing "My Lists": Outside-In TDD
25. CI: Continuous Integration
26. The Token Social Bit, the Page Pattern, and an Exercise for the Reader
27. Fast Tests, Slow Tests, and Hot Lava
Appendix A: Obey the Testing Goat!
Appendix B: The Subtleties of Functionally Testing External Dependencies
Appendix C: Continuous Deployment (CD)
Appendix D: Behaviour-Driven Development (BDD) Tools
Appendix E: Test Isolation, and "Listening to Your Tests"
Appendix F: Building a REST API: JSON, Ajax, and Mocking with JavaScript
Appendix G: Cheat Sheet
Appendix H: What to Do Next
Appendix I: Source Code Examples
Appendix J: Bibliography

Saving User Input: Testing the Database

So far, we've managed to return a static HTML page with an input box in it. Next, we want to take the text that the user types into that input box and send it to the server, so that we can save it somehow and display it back to them later.

The first time I started writing code for this chapter, I immediately wanted to skip to what I thought was the right design: multiple database tables for lists and list items, a bunch of different URLs for adding new lists and items, three new view functions, and about half a dozen new unit tests for all of the above. But I stopped myself. Although I was pretty sure I was smart enough to

handle coding all those problems at once, the point of TDD is to enable you to do one thing at a time, when you need to. So I decided to be deliberately short-sighted, and at any given moment *only* do what was necessary to get the functional tests (FTs) a little further.
This will be a demonstration of how TDD can support an incremental, iterative style of development—it may not be the quickest route, but you do get there in the end.[1] There's a neat side benefit, which is that it enables me to introduce new concepts like models, dealing with POST requests, Django template tags, and so on, *one at a time* rather than having to dump them on you all at once.

None of this says that you *shouldn't* try to think ahead and be clever. In the next chapter, we'll use a bit more design and up-front thinking, and show how that fits in with TDD. But for now, let's plough on mindlessly and just do what the tests tell us to.

Wiring Up Our Form to Send a POST Request

At the end of the last chapter, the tests were telling us we weren't able to save the user's input:

```
  File "...goat-book/functional_tests.py", line 40, in
test_can_start_a_todo_list
[...]
AssertionError: False is not true : New to-do item did not appear in table
```

To get it to the server, for now we'll use a standard HTML POST request. A little boring, but also nice and easy to deliver—we can use all sorts of sexy HTML5 and JavaScript later in the book.

To get our browser to send a POST request, we need to do two things:

1. Give the `<input>` element a `name=` attribute.
2. Wrap it in a `<form>` tag[2] with `method="POST"`.

Testing the Contract Between Frontend and Backend

If you remember in the last chapter, we said we wanted to come back and revisit the smoke test of our home page template content. Let's have a quick look at our unit tests:

lists/tests.py

```
class HomePageTest(TestCase):
    def test_uses_home_template(self):
        response = self.client.get("/")
        self.assertTemplateUsed(response, "home.html")

    def test_renders_homepage_content(self):
        response = self.client.get("/")
        self.assertContains(response, "To-Do")
```
PYTHON

What's important about our home page content? How can we obey both the "don't test constants" rule and the "test behaviour, not implementation" rule?

The specific spelling of the word "To-Do" is not important. As we've just seen, the most important *behaviour* that our home page is enabling, is the ability to submit a to-do item. The way we're going to deliver that is by adding a `<form>` tag with `method="POST"`, and inside that, making sure our `<input>` has a `name="item_text"`.

Our FTs are telling us that it's not working at a high level, so what unit tests can we write at the lower level? Let's start with the form:

lists/tests.py (ch05l001)

```
class HomePageTest(TestCase):
    def test_uses_home_template(self):
        [...]

    def test_renders_input_form(self):   1
        response = self.client.get("/")
        self.assertContains(response, '<form method="POST">')   2
```
PYTHON

1 We change the name of the test.

2 And we assert on the `<form>` tag specifically.

That gives us:

```
$ python manage.py test
[...]
AssertionError: False is not true : Couldn't find '<form method="POST">' in the
following response
b'<html>\n  <head>\n    <title>To-Do lists</title>\n  </head>\n  <body>\n
<h1>Your To-Do list</h1>\n    <input id="id_new_item" placeholder="Enter a
to-do item" />\n    <table id="id_list_table">\n    </table>\n
</body>\n</html>\n'
```

Let's adjust our template at *lists/templates/home.html*:

lists/templates/home.html (ch05l002)

```
HTML
<h1>Your To-Do list</h1>
<form method="POST">
  <input id="id_new_item" placeholder="Enter a to-do item" />
</form>
```

That gives us passing unit tests:

```
OK
```

And next, let's add a test for the `name=` attribute on the `<input>` tag:

lists/tests.py (ch05l003)

```
PYTHON
def test_renders_input_form(self):
    response = self.client.get("/")
    self.assertContains(response, '<form method="POST">')
    self.assertContains(response, '<input name="item_text"')
```

That gives us this expected failure:

```
[...]
AssertionError: False is not true : Couldn't find '<input name="item_text"' in
the following response
b'<html>\n  <head>\n    <title>To-Do lists</title>\n  </head>\n  <body>\n
<h1>Your To-Do list</h1>\n    <form method="POST">\n      <input
id="id_new_item" placeholder="Enter a to-do item" />\n    </form>\n    <table
id="id_list_table">\n    </table>\n  </body>\n</html>\n'
```

And we fix it like this:

lists/templates/home.html (ch05l004)

```
HTML
<h1>Your To-Do list</h1>
<form method="POST">
  <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
</form>
<table id="id_list_table">
```

That gives us passing unit tests:

```
OK
```

The lesson here is that we've tried to identify the "contract" between the frontend and the backend of our site. For our HTML form to work, it needs the form with the right `method`, and the input with the right `name`. Everything else is cosmetic. So that's what we test for in our unit tests.

Debugging Functional Tests

Time to go back to our FT. It gives us a slightly cryptic, unexpected error:

```
$ python functional_tests.py
[...]
Traceback (most recent call last):
  File "...goat-book/functional_tests.py", line 38, in
test_can_start_a_todo_list
    table = self.browser.find_element(By.ID, "id_list_table")
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_list_table"]; [...]
```

Oh dear, we're now failing two lines *earlier*, after we submit the form, but before we are able to do the assert. Selenium seems to be unable to find our list table. Why on earth would that happen? Let's take another look at our code:

functional_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)

        table = self.browser.find_element(By.ID, "id_list_table")   1
        rows = table.find_elements(By.TAG_NAME, "tr")
        self.assertTrue(
            any(row.text == "1: Buy peacock feathers" for row in rows),
            "New to-do item did not appear in table",
        )
```

PYTHON

1 Our test unexpectedly fails on this line. How do we figure out what's going on?

When a functional test fails with an unexpected failure, there are several things we can do to debug it:

- Add `print` statements to show, for example, what the current page text is.
- Improve the *error message* to show more info about the current state.
- Manually visit the site yourself.
- Use `time.sleep` to pause the test during execution so you can inspect what was happening. [3]

We'll look at all of these over the course of this book, but the `time.sleep` option is the one that leaps to mind with this kind of error in an FT. Let's try it now.

Debugging with time.sleep

Conveniently, we've already got a sleep just before the error occurs; let's just extend it a little:

functional_tests.py (ch05l005)

```
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(10)

        table = self.browser.find_element(By.ID, "id_list_table")
```

Depending on how fast Selenium runs on your PC, you may have caught a glimpse of this already, but when we run the FTs again, we've got time to see what's going on: you should see a page that looks like Django debug page showing CSRF error, with lots of Django debug information.

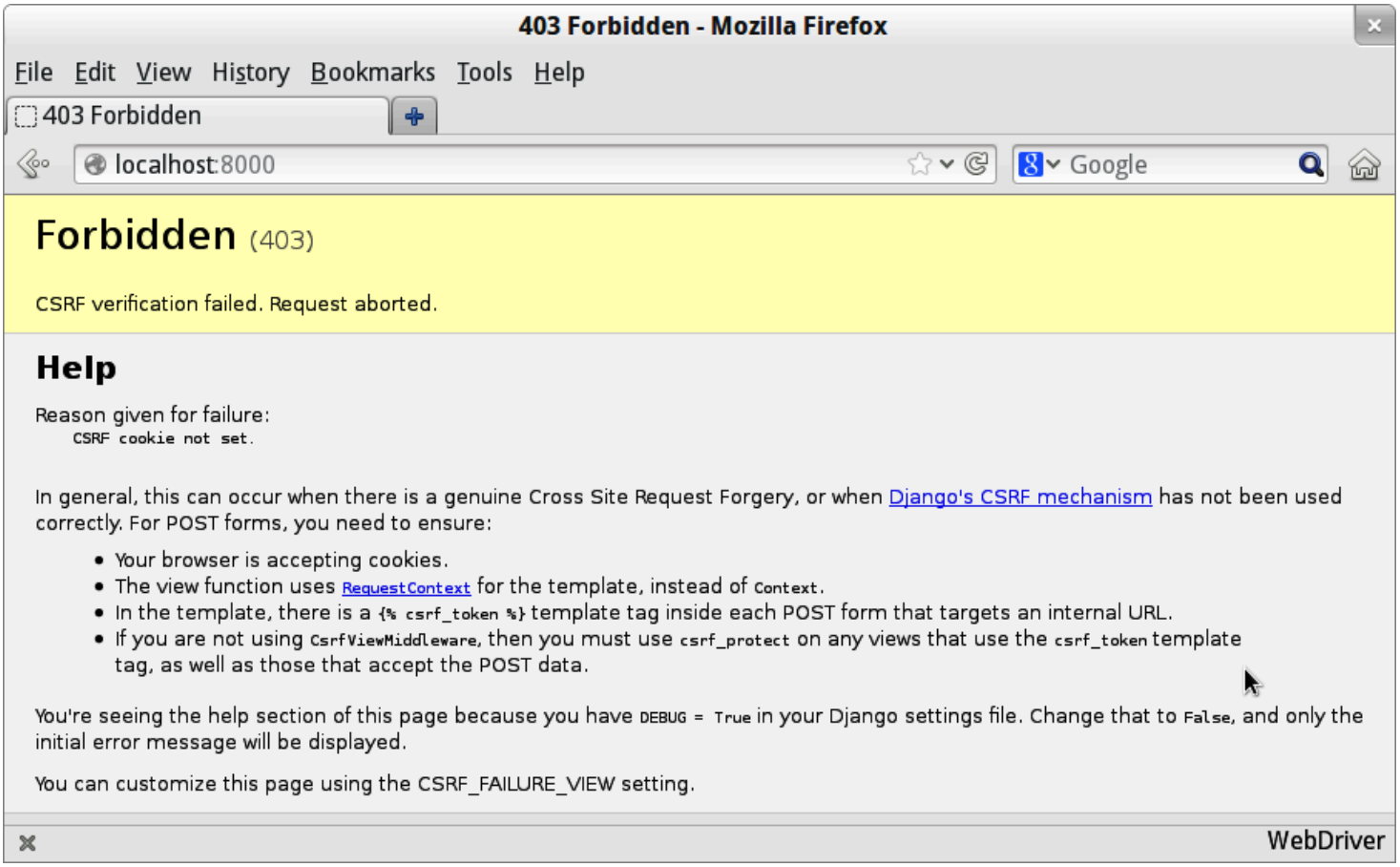


Figure 1. Django debug page showing CSRF error

Security: Surprisingly Fun!

If you've never heard of a *cross-site request forgery* (CSRF) exploit, why not look it up now? Like all security exploits, it's entertaining to read about, being an ingenious use of a system in unexpected ways.

When I went to university to get my computer science degree, I signed up for the "security" module out of a sense of duty: *Oh well, it'll probably be very dry and boring, but I suppose I'd better take it. Eat your vegetables, and so forth*. It turned out to be one of the most fascinating modules of the whole course! Absolutely full of the joy of hacking, of the particular mindset it takes to think about how systems can be used in unintended ways.

I want to recommend the textbook from that course, Ross Anderson's *Security Engineering* (https://oreil.ly/TKmYQ). It's quite light on pure crypto, but it's absolutely full of interesting discussions of unexpected topics like lock picking, forging bank notes, inkjet printer cartridge economics, and spoofing South African Air Force jets with replay attacks. It's a huge tome, about three inches thick, and I promise you it's an absolute page-turner.

Django's CSRF protection involves placing a little autogenerated unique token into each generated form, to be able to verify that POST requests have definitely come from the form generated by the server. So far, our template has been pure HTML, and in this step we make the first use of Django's template magic. To add the CSRF token, we use a *template tag*, which has the curly-bracket/percent syntax, `{% ... %}` —famous for being the world's most annoying two-key touch-typing combination:

lists/templates/home.html (ch05l006)

```HTML
<form method="POST">
  <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
  {% csrf_token %}
</form>
```

Django will substitute the template tag during rendering with an `<input type="hidden">` containing the CSRF token. Rerunning the functional test will now bring us back to our previous (expected) failure:

```
  File "...goat-book/functional_tests.py", line 40, in
test_can_start_a_todo_list
[...]
AssertionError: False is not true : New to-do item did not appear in table
```

Because our long `time.sleep` is still there, the test will pause on the final screen, showing us that the new item text disappears after the form is submitted, and the page refreshes to show an empty form again. That's because we haven't wired up our server to deal with the POST request yet—it just ignores it and displays the normal home page.

We can put our normal short `time.sleep` back now though:

functional_tests.py (ch05l007)

```
        # "1: Buy peacock feathers" as an item in a to-do list table
        inputbox.send_keys(Keys.ENTER)
        time.sleep(1)

        table = self.browser.find_element(By.ID, "id_list_table")
```

PYTHON

Processing a POST Request on the Server

Because we haven't specified an `action=` attribute in the form, it is submitting back to the same URL it was rendered from by default (i.e., `/`), which is dealt with by our `home_page` function. That's fine for now; let's adapt the view to be able to deal with a POST request.

That means a new unit test for the `home_page` view. Open up *lists/tests.py*, and add a new method to `HomePageTest`:

lists/tests.py (ch05l008)

```
class HomePageTest(TestCase):
    def test_uses_home_template(self):
        [...]
    def test_renders_input_form(self):
        response = self.client.get("/")
        self.assertContains(response, '<form method="POST">')
        self.assertContains(response, '<input name="item_text"')   2

    def test_can_save_a_POST_request(self):
        response = self.client.post("/", data={"item_text": "A new list item"})   1
2
        self.assertContains(response, "A new list item")   3
```

PYTHON

1 To do a POST, we call `self.client.post` and, as you can see, it takes a `data` argument that contains the form data we want to send.

2 Notice the echo of the `item_text` name from earlier.[4]

3 Then we check that the text from our POST request ends up in the rendered HTML.

That gives us our expected fail:

```
$ python manage.py test
[...]
AssertionError: False is not true : Couldn't find 'A new list item' in the
following response
b'<html>\n  <head>\n    <title>To-Do lists</title>\n  </head>\n  <body>\n
<h1>Your To-Do list</h1>\n    <form method="POST">\n      <input
name="item_text" id="id_new_item" placeholder="Enter a to-do item" />\n
<input type="hidden" name="csrfmiddlewaretoken"
value="[...]
</form>\n    <table id="id_list_table">\n    </table>\n  </body>\n</html>\n'
```

In (slightly exaggerated) TDD style, we can single-mindedly do "the simplest thing that could possibly work" to address this test failure, which is to add an `if` and a new code path for POST requests, with a deliberately silly return value:

lists/views.py (ch05l009)

```
from django.http import HttpResponse
from django.shortcuts import render


def home_page(request):
    if request.method == "POST":   1
        return HttpResponse("You submitted: " + request.POST["item_text"])   2
    return render(request, "home.html")
```

PYTHON

1 `request.method` lets us check whether we got a POST or a GET request.

2 `request.POST` is a dictionary-like object containing the form data (in this case, the `item_text` value we expect from the form `input` tag).

Fine, that gets our unit tests passing:

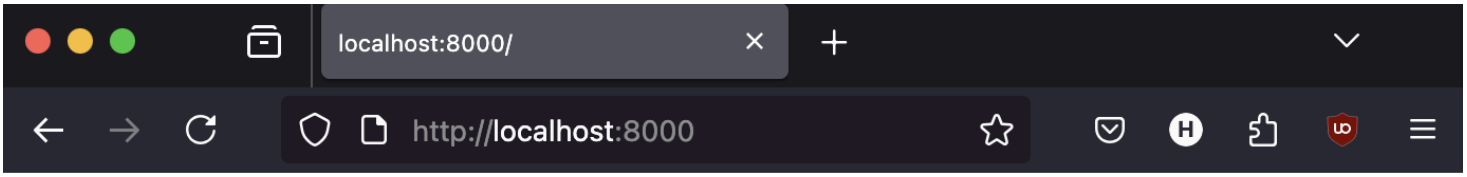
```
OK
```

...but it's not really what we want.[5]

And even if we were genuinely hoping this was the right solution, our FTs are here to remind us that this isn't how things are supposed to work:

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_list_table"]; [...]
```

The list table disappears after the form submission. If you didn't see it in the FT run, try it manually with `runserver` ; you'll see something like I see my item text but no table....



You submitted: Buy asparagus

Figure 2. I see my item text but no table...

What we really want to do is add the POST submission to the to-do items table in the home page template. We need some sort of way to pass data from our view, to be shown in the template.

Passing Python Variables to Be Rendered in the Template

We've already had a hint of it, and now it's time to start to get to know the real power of the Django template syntax, which is to pass variables from our Python view code into HTML templates.

Let's start by seeing how the template syntax lets us include a Python object in our template. The notation is `{{ ... }}` , which displays the object as a string:

lists/templates/home.html (ch05l010)

```
HTML
<body>
  <h1>Your To-Do list</h1>
  <form method="POST">
    <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
    {% csrf_token %}
  </form>
  <table id="id_list_table">
    <tr><td>{{ new_item_text }}</td></tr>   1
  </table>
</body>
```

1 Here's our template variable. `new_item_text` will be the variable name for the user input we display in the template.

Let's adjust our unit test so that it checks whether we are still using the template:

lists/tests.py (ch05l011)

```
PYTHON
    def test_can_save_a_POST_request(self):
        response = self.client.post("/", data={"item_text": "A new list item"})
        self.assertContains(response, "A new list item")
        self.assertTemplateUsed(response, "home.html")
```

And that will fail as expected:

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

Good; our deliberately silly return value is now no longer fooling our tests, so we are allowed to rewrite our view, and tell it to pass the POST parameter to the template. The `render` function takes, as its third argument, a dictionary, which maps template variable names to their values.

In theory, we can use it for the POST case as well as the default GET case, so let's remove the `if request.method == "POST"` and simplify our view right down to:

lists/views.py (ch05l012)

```
def home_page(request):
    return render(
        request,
        "home.html",
        {"new_item_text": request.POST["item_text"]},
    )
```
PYTHON

What do the tests think?

```
ERROR: test_uses_home_template
(lists.tests.HomePageTest.test_uses_home_template)

[...]
    {"new_item_text": request.POST["item_text"]},
                      ~~~~~~~~~~~~^^^^^^^^^^^^^
[...]
django.utils.datastructures.MultiValueDictKeyError: 'item_text'
```

An Unexpected Failure

Oops, an *unexpected failure*.

If you remember the rules for reading tracebacks, you'll spot that it's actually a failure in a *different* test. We got the actual test we were working on to pass, but the unit tests have picked up an unexpected consequence, a regression: we broke the code path where there is no POST request.

This is the whole point of having tests. Yes, perhaps we could have predicted this would happen, but imagine if we'd been having a bad day or weren't paying attention: our tests have just saved us from accidentally breaking our application and, because we're using TDD, we found out immediately. We didn't have to wait for a QA team, or switch to a web browser and click through our site manually, so we can get on with fixing it straight away. Here's how:

lists/views.py (ch05l013)

```
def home_page(request):
    return render(
        request,
        "home.html",
        {"new_item_text": request.POST.get("item_text", "")},
    )
```
PYTHON

We use `dict.get` (https://docs.python.org/3/library/stdtypes.html#dict.get) to supply a default value, for the case where we are doing a normal GET request, when the POST dictionary is empty.

The unit tests should now pass. Let's see what the FTs say:

```
AssertionError: False is not true : New to-do item did not appear in table
```

If your functional tests show you a different error at this point, or at any point in this chapter, complaining about a `StaleElementReferenceException`, you may need to increase the `time.sleep` explicit wait—try two or three seconds instead of one; then read on to the next chapter for a more robust solution.

Improving Error Messages in Tests

Hmm, not a wonderfully helpful error. Let's use another of our FT debugging techniques: improving the error message. This is probably the most constructive technique, because those improved error messages stay around to help debug any future errors:

functional_tests.py (ch05l014)

```
self.assertTrue(
    any(row.text == "1: Buy peacock feathers" for row in rows),
    f"New to-do item did not appear in table. Contents were:\n{table.text}",
)
```
PYTHON

That gives us a more helpful message:

```
AssertionError: False is not true : New to-do item did not appear in table.
Contents were:
Buy peacock feathers
```

Actually, you know what would be even better? Making that assertion a bit less clever! As you may remember from [chapter_04_philosophy_and_refactoring], I was very pleased with myself for using the `any()` function, but one of my early release readers (thanks, Jason!) suggested a much simpler implementation. We can replace all four lines of the `assertTrue` with a single `assertIn`:

functional_tests.py (ch05l015)

```
        self.assertIn("1: Buy peacock feathers", [row.text for row in rows])
```
PYTHON

Much better. You should always be very worried whenever you think you're being clever, because what you're probably being is *overcomplicated*.

Now we get the error message for free:

```
    self.assertIn("1: Buy peacock feathers", [row.text for row in rows])
AssertionError: '1: Buy peacock feathers' not found in ['Buy peacock feathers']
```

Consider me suitably chastened.

If, instead, your FT seems to be saying the table is empty ("not found in ['']"), check your `<input>` tag—does it have the correct `name="item_text"` attribute? And does it have `method="POST"`? Without them, the user's input won't be in the right place in `request.POST`.

The point is that the FT wants us to enumerate list items with a "1:" at the beginning of the first list item.

The fastest way to get that to pass is with another quick "cheating" change to the template:

lists/templates/home.html (ch05l016)

```
        <tr><td>1: {{ new_item_text }}</td></tr>
```
HTML

When Should You Stop Cheating? DRY Versus Triangulation

People often ask about when it's OK to "stop cheating", and change from an implementation we know to be wrong, to one we're happy with.

One justification is *eliminate duplication*—aka DRY (don't repeat yourself)—which (with some caveats) is a good guideline for any kind of code.

If your test uses a magic constant (like the "1:" in front of our list item), and your application code also uses it, some people say *that* counts as duplication, so it justifies refactoring. Removing the magic constant from the application code usually means you have to stop cheating.

It's a judgement call, but I feel that this is stretching the definition of "repetition" a little, so I often like to use a second technique, which is called *triangulation*: if your tests let you get away with writing "cheating" code that you're not happy with (like returning a magic constant), then *write another test* that forces you to write some better code. That's what we're doing when we extend the FT to check that we get a "2:" when inputting a second list item.

See also Three Strikes and Refactor for a further note of caution on applying DRY too quickly.

Now we get to the `self.fail('Finish the test!')`. If we get rid of that and finish writing our FT, to add the check for adding a second item to the table (copy and paste is our friend), we begin to see that our first cut solution really isn't going to, um, cut it:

functional_tests.py (ch05l017)

```python
        # 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)
        time.sleep(1)

        # The page updates again, and now shows both items on her list
        table = self.browser.find_element(By.ID, "id_list_table")
        rows = table.find_elements(By.TAG_NAME, "tr")
        self.assertIn(
            "2: Use peacock feathers to make a fly",
            [row.text for row in rows],
        )
        self.assertIn(
            "1: Buy peacock feathers",
            [row.text for row in rows],
        )

        # Satisfied, she goes back to sleep
```

Sure enough, the FTs return an error:

```
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Use
peacock feathers to make a fly']
```

Three Strikes and Refactor

But before we go further—we've got a bad *code smell*[6] in this FT. We have three almost identical code blocks checking for new items in the list table. When we want to apply the DRY principle, I like to follow the motto *three strikes and refactor*. You can copy and paste code once, and it may be premature to try to remove the duplication it causes, but once you get three occurrences, it's time to tidy up.

Let's start by committing what we have so far. Even though we know our site has a major flaw—it can only handle one list item—it's still further ahead than it was. We may have to rewrite it all, and we may not, but the rule is that before you do any refactoring, always do a commit:

```
$ git diff
# should show changes to functional_tests.py, home.html,
# tests.py and views.py
$ git commit -a
```

Always do a commit before embarking on a refactor.

Onto our functional test refactor. Let's use a helper method—remember, only methods that begin with `test_` will be run as tests, so you can use other methods for your own purposes:

functional_tests.py (ch05l018)

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

    def check_for_row_in_list_table(self, row_text):
        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])

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

I like to put helper methods near the top of the class, between the `tearDown` and the first test. Let's use it in the FT:

functional_tests.py (ch05l019)

```
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")

        # 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)
        time.sleep(1)

        # The page updates again, and now shows both items on her list
        self.check_for_row_in_list_table("2: Use peacock feathers to make a fly")
        self.check_for_row_in_list_table("1: Buy peacock feathers")

        # Satisfied, she goes back to sleep
```

We run the FT again to check that it still behaves in the same way:

```
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Use
peacock feathers to make a fly']
```

Good. Now we can commit the FT refactor as its own small, atomic change:

```
$ git diff # check the changes to functional_tests.py
$ git commit -a
```

There are a couple more bits of duplication in the FTs, like the repetition of finding the `inputbox`, but they're not as egregious yet, so we'll deal with them later.

Instead, back to work. If we're ever going to handle more than one list item, we're going to need some kind of persistence, and databases are a stalwart solution in this area.

The Django ORM and Our First Model

An object-relational mapper (ORM) is a layer of abstraction for data stored in a database with tables, rows, and columns. It lets us work with databases using familiar object-oriented metaphors that work well with code. Classes map to database tables, attributes map to columns,

and an individual instance of the class represents a row of data in the database.
Django comes with an excellent ORM, and writing a unit test that uses it is actually an excellent way of learning it, because it exercises code by specifying how we want it to work.

Let's create a new class in *lists/tests.py*:

lists/tests.py (ch05l020)

```
PYTHON
from django.test import TestCase
from lists.models import Item


class HomePageTest(TestCase):
    [...]


class ItemModelTest(TestCase):
    def test_saving_and_retrieving_items(self):
        first_item = Item()
        first_item.text = "The first (ever) list item"
        first_item.save()

        second_item = Item()
        second_item.text = "Item the second"
        second_item.save()

        saved_items = Item.objects.all()
        self.assertEqual(saved_items.count(), 2)

        first_saved_item = saved_items[0]
        second_saved_item = saved_items[1]
        self.assertEqual(first_saved_item.text, "The first (ever) list item")
        self.assertEqual(second_saved_item.text, "Item the second")
```

You can see that creating a new record in the database is a relatively simple matter of creating an object, assigning some attributes, and calling a `.save()` function. Django also gives us an API for querying the database via a class attribute, `.objects`, and we use the simplest possible query, `.all()`, which retrieves all the records for that table. The results are returned as a list-like object called a `QuerySet`, from which we can extract individual objects, and also call further functions, like `.count()`. We then check the objects as saved to the database, to check whether the right information was saved.

Django's ORM has many other helpful and intuitive features; this might be a good time to skim through the Django tutorial (https://docs.djangoproject.com/en/5.2/intro/tutorial01), which has an excellent intro to them.

I've written this unit test in a very verbose style, as a way of introducing the Django ORM. I wouldn't recommend writing your model tests like this "in real life", because it's testing the framework, rather than testing our own code. We'll actually rewrite this test to be much more concise in [chapter_16_advanced_forms] (specifically, at [rewrite-model-test]).

Unit Tests Versus Integration Tests, and the Database

Some people will tell you that a "real" unit test should never touch the database, and that the test I've just written should be more properly called an "integration" test, because it doesn't *only* test our code, but also relies on an external system—that is, a database.

It's OK to ignore this distinction for now—we have two types of test: the high-level FTs, which test the application from the user's point of view, and these lower-level tests, which test it from the programmer's point of view.

We'll come back to this topic and talk about the differences between unit tests, integration tests, and more in [chapter_27_hot_lava], at the end of the book.

Let's try running the unit test. Here comes another unit-test/code cycle:

```
ImportError: cannot import name 'Item' from 'lists.models'
```

Very well, let's give it something to import from *lists/models.py*. We're feeling confident so we'll skip the `Item = None` step, and go straight to creating a class:

lists/models.py (ch05l021)

```
from django.db import models

# Create your models here.
class Item:
    pass
```

PYTHON

That gets our test as far as:

```
[...]
  File "...goat-book/lists/tests.py", line 25, in
test_saving_and_retrieving_items
    first_item.save()
    ^^^^^^^^^^^^^^^^
AttributeError: 'Item' object has no attribute 'save'
```

To give our `Item` class a `save` method, and to make it into a real Django model, we make it inherit from the `Model` class:

lists/models.py (ch05l022)

```
from django.db import models


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

PYTHON

Our First Database Migration

The next thing that happens is a huuuuge traceback, the long and short of which is that there's a problem with the database:

```
django.db.utils.OperationalError: no such table: lists_item
```

In Django, the ORM's job is to model and read and write from database tables, but there's a second system that's in charge of actually *creating* the tables in the database called "migrations". Its job is to let you add, remove, and modify tables and columns, based on changes you make to your *models.py* files.

One way to think of it is as a version control system (VCS) for your database. As we'll see later, it proves particularly useful when we need to upgrade a database that's deployed on a live server.

For now all we need to know is how to build our first database migration, which we do using the `makemigrations` command:[7]

```
$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0001_initial.py
    + Create model Item
$ ls lists/migrations
0001_initial.py  __init__.py  __pycache__
```

If you're curious, you can go and take a look in the migrations file, and you'll see it's a representation of our additions to *models.py*.

In the meantime, we should find that our tests get a little further.

The Test Gets Surprisingly Far

The test actually gets surprisingly far:

```
$ python manage.py test
[...]
    self.assertEqual(first_saved_item.text, "The first (ever) list item")
                     ^^^^^^^^^^^^^^^^^^^^
AttributeError: 'Item' object has no attribute 'text'
```

That's a full eight lines later than the last failure—we've been all the way through saving the two `Item`s, and we've checked that they're saved in the database, but Django just doesn't seem to have "remembered" the `.text` attribute.

If you're new to Python, you might have been surprised that we were allowed to assign the `.text` attribute at all. In a language like Java, you would probably get a compilation error. Python is more relaxed.

Classes that inherit from `models.Model` will map to tables in the database. By default, they get an autogenerated `id` attribute, which will be a primary key column[8] in the database, but you have to define any other columns and attributes you want explicitly. Here's how we set up a text column:

lists/models.py (ch05l024)

```
class Item(models.Model):
    text = models.TextField()
```

PYTHON

Django has many other field types, like `IntegerField`, `CharField`, `DateField`, and so on. I've chosen `TextField` rather than `CharField` because the latter requires a length restriction, which seems arbitrary at this point. You can read more on field types in the Django tutorial (https://docs.djangoproject.com/en/5.2/intro/tutorial02/#creating-models) and in the documentation (https://docs.djangoproject.com/en/5.2/ref/models/fields).

A New Field Means a New Migration

Running the tests gives us another database error:

```
django.db.utils.OperationalError: table lists_item has no column named text
```

It's because we've added another new field to our database, which means we need to create another migration. Nice of our tests to let us know!

Let's try it:

```
$ python manage.py makemigrations
It is impossible to add a non-nullable field 'text' to item without specifying
a default. This is because the database needs something to populate existing
rows.
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a null
value for this column)
 2) Quit and manually define a default value in models.py.
Select an option:2
```

Ah. It won't let us add the column without a default value. Let's pick option 2 and set a default in *models.py*. I think you'll find the syntax reasonably self-explanatory:

lists/models.py (ch05l025)

```
class Item(models.Model):
    text = models.TextField(default="")
```
PYTHON

And now the migration should complete:

```
$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0002_item_text.py
    + Add field text to item
```

So, two new lines in *models.py*, two database migrations, and as a result, the `.text` attribute on our model objects is now recognised as a special attribute, so it does get saved to the database, and the tests pass:

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

Ran 4 tests in 0.010s
OK
```

So let's do a commit for our first ever model!

```
$ git status # see tests.py, models.py, and 2 untracked migrations
$ git diff # review changes to tests.py and models.py
$ git add lists
$ git commit -m "Model for list Items and associated migration"
```

Saving the POST to the Database

So, we have a model; now we need to use it!

Let's adjust the test for our home page POST request, and say we want the view to save a new item to the database instead of just passing it through to its response. We can do that by adding three new lines to the existing test called `test_can_save_a_POST_request`:

lists/tests.py (ch05l027)

```
def test_can_save_a_POST_request(self):
    response = self.client.post("/", data={"item_text": "A new list item"})

    self.assertEqual(Item.objects.count(), 1)   1
    new_item = Item.objects.first()   2
    self.assertEqual(new_item.text, "A new list item")   3

    self.assertContains(response, "A new list item")
    self.assertTemplateUsed(response, "home.html")
```
PYTHON

1 We check that one new `Item` has been saved to the database. `objects.count()` is a shorthand for `objects.all().count()`.

2 `objects.first()` is the same as doing `objects.all()[0]`, except it will return `None` if there are no objects.[9]

3 We check that the item's text is correct.

This test is getting a little long-winded. It seems to be testing lots of different things. That's another *code smell*—a long unit test either needs to be broken into two, or it may be an indication that the thing you're testing is too complicated. Let's add that to a little to-do list of our own, perhaps on a piece of scrap paper:

- *Code smell: POST test is too long?*

An Alternative Testing Strategy: Staying at the HTTP Level

It's a very common pattern in Django to test POST views by asserting on the side effects, as seen in the database. Sandi Metz, a TDD legend from the Ruby world, puts it like this: "test commands via public side effects".[10]

But is the database really a public API? That's arguable. Certainly it's at a different level of abstraction, or a different conceptual "layer" in the application, to the HTTP requests we're working with in our current unit tests.

If you wanted to write our tests in a way that stays at the HTTP level—that treats the application as more of an "opaque box"—you can prove to yourself that to-do items are persisted, by sending more than one:

lists/tests/tests.py

```
    def test_can_save_multiple_items(self):
        self.client.post("/", data={"item_text": "first item"})
        response = self.client.post("/", data={"item_text": "second item"})
        self.assertContains(response, "first item")
        self.assertContains(response, "second item")
```

PYTHON

If you feel like going off road, why not give it a try?

Writing things down on a scratchpad like this reassures us that we won't forget them, so we are comfortable getting back to what we were working on. We rerun the tests and see an expected failure:

```
    self.assertEqual(Item.objects.count(), 1)
AssertionError: 0 != 1
```

Let's adjust our view:

lists/views.py (ch05l028)

```
from django.shortcuts import render
from lists.models import Item


def home_page(request):
    item = Item()
    item.text = request.POST.get("item_text", "")
    item.save()

    return render(
        request,
        "home.html",
        {"new_item_text": request.POST.get("item_text", "")},
    )
```

PYTHON

I've coded a very naive solution and you can probably spot a very obvious problem, which is that we're going to be saving empty items with every request to the home page. Let's add that to our list of things to fix later. You know, along with the painfully obvious fact that we currently have no way at all of having different lists for different people. That we'll keep ignoring for now.

Remember, I'm not saying you should always ignore glaring problems like this in "real life". Whenever we spot problems in advance, there's a judgement call to make over whether to stop what you're doing and start again, or leave them until later. Sometimes finishing off what you're doing is still worth it, and sometimes the problem may be so major as to warrant a stop and rethink.

Let's see how the unit tests get on...

```
Ran 4 tests in 0.010s

OK
```

They pass! Good. Let's have a little look at our scratchpad. I've added a couple of the other things that are on our mind:

- *Don't save blank items for every request.*
- *Code smell: POST test is too long?*
- *Display multiple items in the table.*
- *Support more than one list!*

Let's start with the first scratchpad item: "Don't save blank items for every request". We could tack on an assertion to an existing test, but it's best to keep unit tests to testing one thing at a time, so let's add a new one:

lists/tests.py (ch05l029)

```
class HomePageTest(TestCase):
    def test_uses_home_template(self):
        [...]

    def test_can_save_a_POST_request(self):
        [...]

    def test_only_saves_items_when_necessary(self):
        self.client.get("/")
        self.assertEqual(Item.objects.count(), 0)
```

PYTHON

That gives us a `1 != 0` failure. Let's fix it by bringing the `if request.method` check back and putting the `Item` creation in there:

lists/views.py (ch05l030)

```
def home_page(request):
    if request.method == "POST":   1
        item = Item()
        item.text = request.POST["item_text"]   2
        item.save()

    return render(
        request,
        "home.html",
        {"new_item_text": request.POST.get("item_text", "")},
    )
```

PYTHON

1. We bring back the `request.method` check.
2. And we can switch from using `request.POST.get()` to `request.POST[]` with square brackets, because we know for sure that the `item_text` key should be in there, and it's better to fail hard if it isn't.

And that gets the test passing:

```
Ran 5 tests in 0.010s

OK
```

Redirect After a POST

But, yuck—those duplicated `request.POST` accesses are making me pretty unhappy. Thankfully we are about to have the opportunity to fix it. A view function has two jobs: processing user input and returning an appropriate response. We've taken care of the first part, which is saving the user's input to the database, so now let's work on the second part.

Always redirect after a POST (https://oreil.ly/yGSl0), they say, so let's do that. Once again we change our unit test for saving a POST request: instead of expecting a response with the item in it, we want it to expect a redirect back to the home page.

lists/tests.py (ch05l031)

```
                                                                    PYTHON
    def test_can_save_a_POST_request(self):
        response = self.client.post("/", data={"item_text": "A new list item"})

        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, "A new list item")

        self.assertRedirects(response, "/")   1

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

1 We no longer expect a response with HTML content rendered by a template, so we lose the `assertContains` calls that looked at that. Instead, we use Django's `assertRedirects` helper, which checks that we return an HTTP 302 redirect, back to the home URL.

That gives us this expected failure:

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

We can now tidy up our view substantially:

lists/views.py (ch05l032)

```python
from django.shortcuts import redirect, render
from lists.models import Item


def home_page(request):
    if request.method == "POST":
        item = Item()
        item.text = request.POST["item_text"]
        item.save()
        return redirect("/")

    return render(
        request,
        "home.html",
        {"new_item_text": request.POST.get("item_text", "")},
    )
```

PYTHON

And the tests should now pass:

```
Ran 5 tests in 0.010s

OK
```

We're at green; time for a little refactor!

Let's have a look at *views.py* and see what opportunities for improvement there might be:

lists/views.py

```python
def home_page(request):
    if request.method == "POST":
        item = Item()  # 1
        item.text = request.POST["item_text"]  # 1
        item.save()  # 1
        return redirect("/")

    return render(
        request,
        "home.html",
        {"new_item_text": request.POST.get("item_text", "")},  # 2
    )
```

PYTHON

There's a quicker way to do these three lines with `.objects.create()`.

This line doesn't seem quite right now; in fact, it won't work at all. Let's make a note on our scratchpad to sort out passing list items to the template. It's actually closely related to "Display multiple items", so we'll put it just before that one:

- ~~*Don't save blank items for every request.*~~
- *Code smell: POST test is too long?*
- *Pass existing list items to the template somehow.*
- *Display multiple items in the table.*
- *Support more than one list!*

And here's the refactored version of *views.py* using the `.objects.create()` helper method that Django provides, for one-line creation of objects:

lists/views.py (ch05l033)

```
def home_page(request):
    if request.method == "POST":
        Item.objects.create(text=request.POST["item_text"])
        return redirect("/")

    return render(
        request,
        "home.html",
        {"new_item_text": request.POST.get("item_text", "")},
    )
```

PYTHON

Better Unit Testing Practice: Each Test Should Test One Thing

Let's address the "POST test is too long" code smell.

Good unit testing practice says that each test should only test one thing. The reason is that it makes it easier to track down bugs. Having multiple assertions in a test means that, if the test fails on an early assertion, you don't know what the statuses of the later assertions are. As we'll see in the next chapter, if we ever break this view accidentally, we want to know whether it's the saving of objects that's broken, or the type of response.

You may not always write perfect unit tests with single assertions on your first go, but now feels like a good time to separate out our concerns:

lists/tests.py (ch05l034)

```
PYTHON
    def test_can_save_a_POST_request(self):
        self.client.post("/", data={"item_text": "A new list item"})
        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, "A new list item")

    def test_redirects_after_POST(self):
        response = self.client.post("/", data={"item_text": "A new list item"})
        self.assertRedirects(response, "/")
```

And we should now see six tests pass instead of five:

```
Ran 6 tests in 0.010s

OK
```

Rendering Items in the Template

Much better! Back to our to-do list:

- ~~*Don't save blank items for every request*~~.
- ~~*Code smell: POST test is too long?*~~
- *Pass existing list items to the template somehow.*
- *Display multiple items in the table.*
- *Support more than one list!*

Crossing things off the list is almost as satisfying as seeing tests pass!

The third and fourth items are the last of the "easy" ones. Our view now does the right thing for POST requests; it saves new list items to the database. Now we want GET requests to load all currently existing list items, and pass them to the template for rendering. Let's have a new unit test for that:

lists/tests.py (ch05l035)

```
class HomePageTest(TestCase):
    def test_uses_home_template(self):
        [...]
    def test_renders_input_form(self):
        [...]

    def test_displays_all_list_items(self):
        Item.objects.create(text="itemey 1")
        Item.objects.create(text="itemey 2")

        response = self.client.get("/")

        self.assertContains(response, "itemey 1")
        self.assertContains(response, "itemey 2")

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

PYTHON

Arrange-Act-Assert or Given-When-Then

Did you notice the use of whitespace in this test? I'm visually separating out the code into three blocks:

lists/tests.py

```
    def test_displays_all_list_items(self):
        Item.objects.create(text="itemey 1")   1
        Item.objects.create(text="itemey 2")   1

        response = self.client.get("/")   2

        self.assertContains(response, "itemey 1")   3
        self.assertContains(response, "itemey 2")   3
```

PYTHON

1. Arrange: where we set up the data we need for the test.
2. Act: where we call the code under test
3. Assert: where we check on the results

This isn't obligatory, but it's a common convention, and it does help see the structure of the test.

Another popular way to talk about this structure is *given-when-then*:

- *Given* the database contains our list with two items,
- *When* I make a GET request for our list,
- *Then* I see the both items in our list.

This latter phrasing comes from the world of behaviour-driven development (BDD), and I actually prefer it somewhat. You can see that it encourages phrasing things in a more natural way, and we're gently nudged to think of things in terms of behaviour and the perspective of the user.

That fails as expected:

```
AssertionError: False is not true : Couldn't find 'itemey 1' in the following
response
b'<html>\n  <head>\n    <title>To-Do lists</title>\n  </head>\n  <body>\n
[...]
```

The Django template syntax has a tag for iterating through lists, `{% for .. in .. %}`; we can use it like this:

lists/templates/home.html (ch05l036)

```
HTML
<table id="id_list_table">
  {% for item in items %}
    <tr><td>1: {{ item.text }}</td></tr>
  {% endfor %}
</table>
```

This is one of the major strengths of the templating system. Now the template will render with multiple `<tr>` rows, one for each item in the variable `items`. Pretty neat! I'll introduce a few more bits of Django template magic as we go, but at some point you'll want to go and read up on the rest of them in the Django docs (https://docs.djangoproject.com/en/5.2/topics/templates).

Just changing the template doesn't get our tests to green; we need to actually pass the items to it from our home page view:

lists/views.py (ch05l037)

```
PYTHON
def home_page(request):
    if request.method == "POST":
        Item.objects.create(text=request.POST["item_text"])
        return redirect("/")

    items = Item.objects.all()
    return render(request, "home.html", {"items": items})
```

That does get the unit tests to pass. Moment of truth…will the functional test pass?

```
$ python functional_tests.py
[...]
AssertionError: 'To-Do' not found in 'OperationalError at /'
```

Oops, apparently not. Let's use another FT debugging technique, and it's one of the most straightforward: manually visiting the site! Open up *http://localhost:8000* in your web browser, and you'll see a Django debug page saying "no such table: lists_item", as in Another helpful debug message.

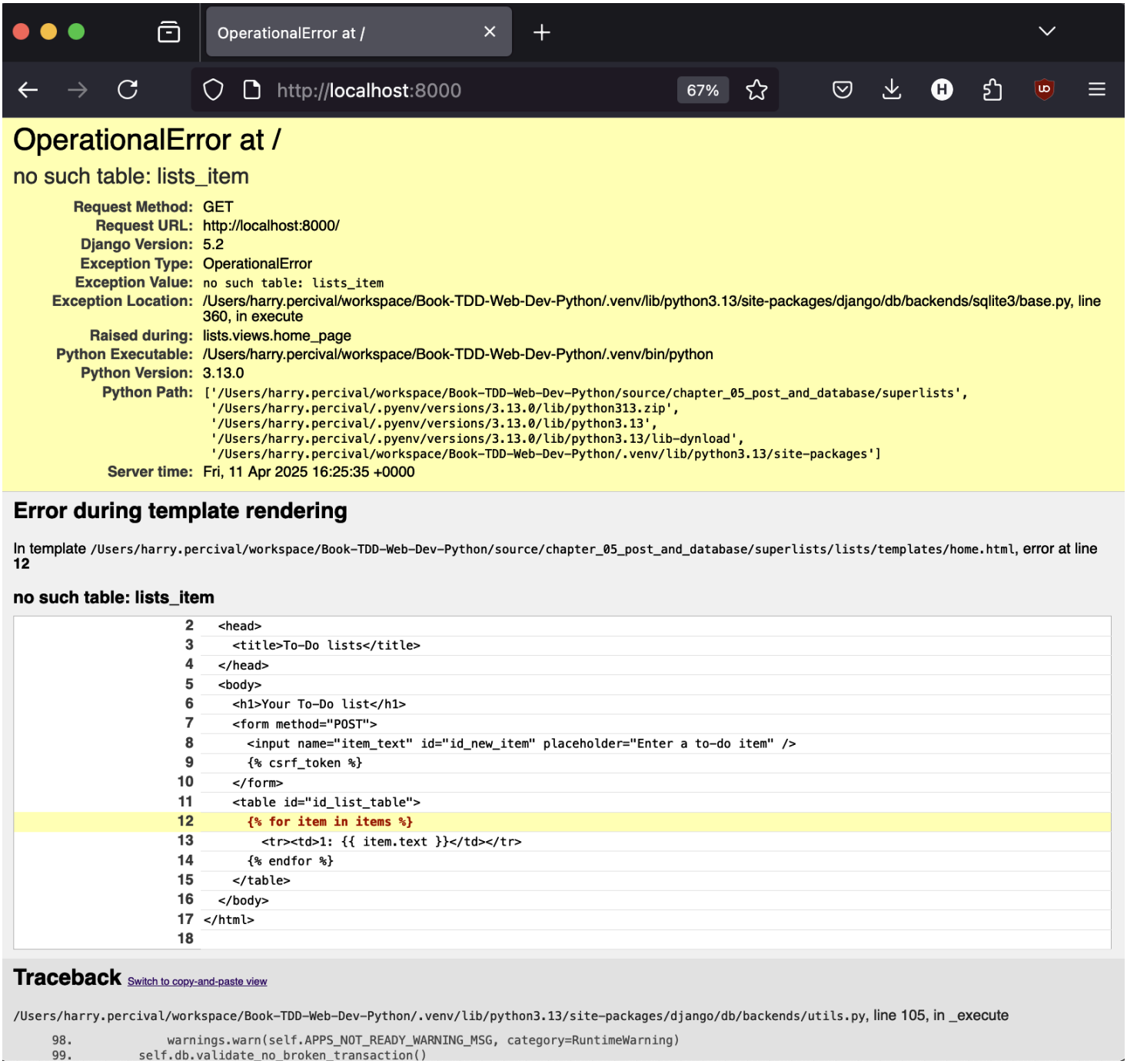


Figure 3. Another helpful debug message

Creating Our Production Database with migrate

So, we've got another helpful error message from Django, which is basically complaining that we haven't set up the database properly. How come everything worked fine in the unit tests, I hear you ask? Because Django creates a special *test database* for unit tests; it's one of the magical things that Django's `TestCase` does.

To set up our "real" database, we need to explicitly create it. SQLite databases are just a file on disk, and you'll see in *settings.py* that Django, by default, will just put it in a file called *db.sqlite3* in the base project directory:

superlists/settings.py

```
PYTHON
[...]
# Database
# https://docs.djangoproject.com/en/5.2/ref/settings/#databases

DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.sqlite3",
        "NAME": BASE_DIR / "db.sqlite3",
    }
}
```

We've told Django everything it needs to create the database, first via *models.py* and then when we created the migrations file. To actually apply it to creating a real database, we use another Django Swiss Army knife *manage.py* command, `migrate`:

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, lists, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying lists.0001_initial... OK
  Applying lists.0002_item_text... OK
  Applying sessions.0001_initial... OK
```

It seems to be doing quite a lot of work! That's because it's the first ever migration, and Django is creating tables for all its built-in "batteries included" apps, like the admin site and the built-in auth modules. We don't need to pay attention to them for now. But you can see our `lists.0001_initial` and `lists.0002_item_text` in there!

At this point, you can refresh the page on *localhost* and see that the error is gone. Let's try running the functional tests again:[11]

```
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy
peacock feathers', '1: Use peacock feathers to make a fly']
```

So close! We just need to get our list numbering right. Another awesome Django template tag, `forloop.counter`, will help here:

lists/templates/home.html (ch05l038)

```
    {% for item in items %}
      <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
    {% endfor %}
```
HTML

If you try it again, you should now see the FT gets to the end:

```
$ python functional_tests.py
.
 ----------------------------------------------------------------------
Ran 1 test in 5.036s

OK
```

Hooray! But, as it's running, you may notice something is amiss, like in There are list items left over from the last run of the test.

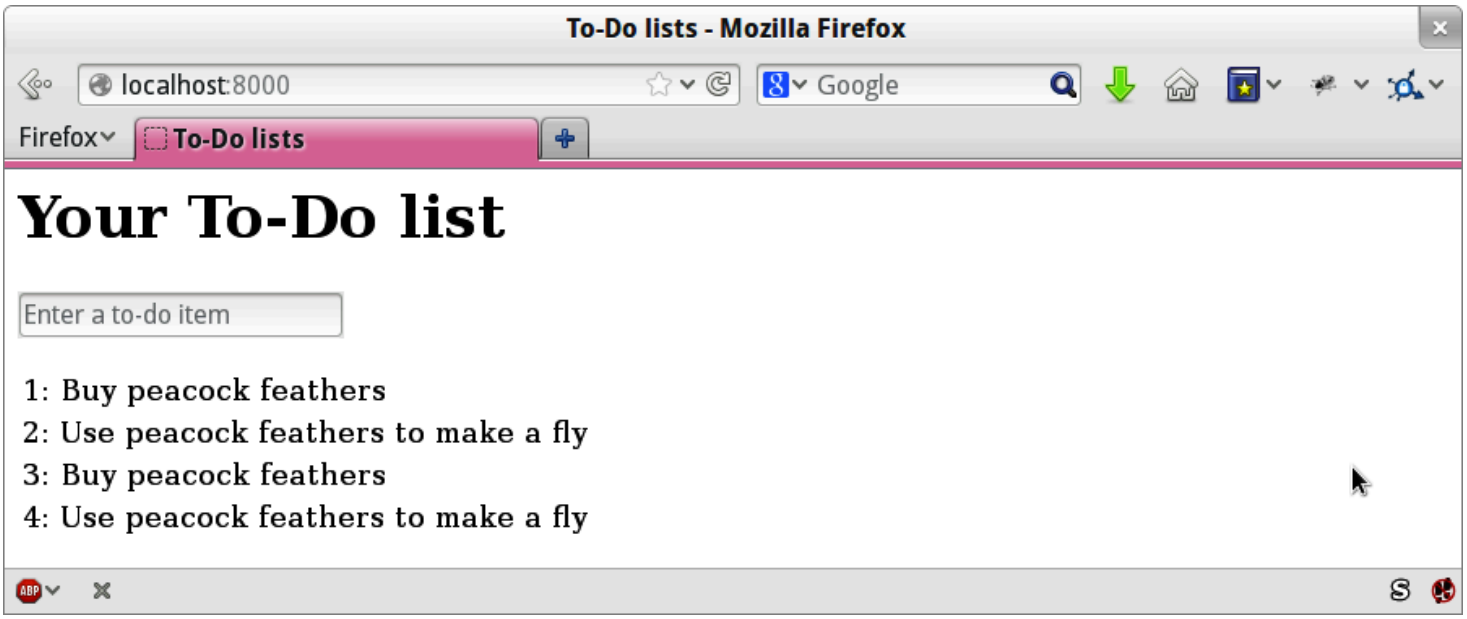


Figure 4. There are list items left over from the last run of the test

Oh dear. It looks like previous runs of the test are leaving stuff lying around in our database. In fact, if you run the tests again, you'll see it gets worse:

```
1: Buy peacock feathers
2: Use peacock feathers to make a fly
3: Buy peacock feathers
4: Use peacock feathers to make a fly
5: Buy peacock feathers
6: Use peacock feathers to make a fly
```

Grrr. We're so close! We're going to need some kind of automated way of tidying up after ourselves. For now, if you feel like it, you can do it manually by deleting the database and re-creating it fresh with `migrate` (you'll need to shut down your Django server first):

```
$ rm db.sqlite3
$ python manage.py migrate --noinput
```

And then (after restarting your server!) reassure yourself that the FT still passes.

Apart from that little bug in our functional testing, we've got some code that's more or less working. Let's do a commit.

Start by doing a `git status` and a `git diff`, and you should see changes to *home.html*, *tests.py*, and *views.py*. Let's add them:

```
$ git add lists
$ git commit -m "Redirect after POST, and show all items in template"
```

You might find it useful to add markers for the end of each chapter, like `git tag end-of-chapter-05`.

Recap

Where are we? How is progress on our app, and what have we learned?

- We've got a form set up to add new items to the list using POST.
- We've set up a simple model in the database to save list items.
- We've learned about creating database migrations, both for the test database (where they're applied automatically) and for the real database (where we have to apply them manually).
- We've used our first couple of Django template tags: `{% csrf_token %}` and the `{% for ... endfor %}` loop.
- And we've used two different FT debugging techniques: `time.sleep` s, and improving the error messages.

But we've got a couple of items on our own to-do list, namely getting the FT to clean up after itself, [illegible] re than one list:

- ~~*Don't save blank items for every request*~~.
- ~~*Code smell: POST test is too long?*~~
- ~~*Pass existing list items to the template somehow*~~.
- ~~*Display multiple items in the table*~~.
- *Clean up after FT runs*.
- *Support more than one list!*

I mean, we *could* ship the site as it is, but people might find it strange that the entire human population has to share a single to-do list. I suppose it might get people to stop and think about how connected we all are to one another, how we all share a common destiny here on Spaceship Earth, and how we must all work together to solve the global problems that we face.

But in practical terms, the site wouldn't be very useful.

Ah well.

Useful TDD Concepts

Regression

When a change unexpectedly breaks some aspect of the application that used to work.

Unexpected failure

When a test fails in a way we weren't expecting. This either means that we've made a mistake in our tests, or that the tests have helped us find a regression, and we need to fix something in our code.

Triangulation

Adding a test case with a new specific example for some existing code, to justify generalising the implementation (which may be a "cheat" until that point).

Three strikes and refactor

A rule of thumb for when to remove duplication from code. When two pieces of code look very similar, it often pays to wait until you see a third use case, so that you're more sure about what part of the code really is the common, reusable part to refactor out.

The scratchpad to-do list

A place to write down things that occur to us as we're coding, so that we can finish up what we're doing and come back to them later. Love a good old-fashioned piece of paper now and again!

---

1. "Geepaw" Hill, another one of the TDD OGs, has a series of blog posts (https://oreil.ly/qTCLk) advocating for taking "Many More Much Smaller Steps (MMMSS)". In this chapter I'm being unrealistically short-sighted for effect, so don't do that! But Geepaw argues that in the real world, when you slice your work into tiny increments, not only do you get there in the end, but you end up delivering business value *faster*.
2. Did you know that you don't need a button to make a form submit? I can't remember when I learned that, but readers have mentioned that it's unusual so I thought I'd draw your attention to it.
3. Another common technique for debugging tests is to use `breakpoint()` to drop into a debugger like `pdb`. This is more useful for *unit* tests rather than FTs though, because in an FT you usually can't step into actual application code. Personally, I only find debuggers useful for really fiddly algorithms, which we won't see in this book.
4. You could even define a constant for this, to make the link more explicit.
5. But we *did* learn about `request.method` and `request.POST`, right? I know it might seem that I'm overdoing it, but doing things in tiny little steps really does have a lot of advantages, and one of them is that you can really think about (or in this case, learn) one thing at a time.
6. If you've not come across the concept, a "code smell" is something about a piece of code that makes you want to rewrite it. Jeff Atwood has a compilation on his blog, *Coding Horror* (https://oreil.ly/GFrNp). The more experience you gain as a programmer, the more fine-tuned your nose becomes to code smells…
7. If you've done a bit of Django before, you may be wondering about when we're going to run "migrate" as well as "makemigrations"? Read on; that's coming up later in the chapter.
8. Database tables usually have a special column called a "primary key", which is the unique identifier for each row in the table. It's worth brushing up on a *tiny* bit of relational database theory, if you're not familiar with the concept or why it's useful. The top three articles I found when searching for "introduction to databases" all seemed pretty good, at the time of writing.
9. You can also use `objects.get()`, which will immediately raise an exception if there are no objects in the database, or if there are more than one. On the plus side you get a more immediate failure, and you get warned if there are too many objects. The downside is that I find it slightly less readable. As so often, it's a trade-off.
10. This advice is in her talk The Magic Tricks of Testing (https://oreil.ly/Gqxgg), which I highly recommend watching.
11. If you get a different error at this point, try restarting your dev server—it may have gotten confused by the changes to the database happening under its feet.

Comments

ALSO ON **OBEY THE TESTING GOAT!**

Finishing "My Lists": Outside-In TDD

2 years ago • 3 comments

The @property Decorator in Python This is a powerful feature of the language, …

Deployment Part 1: Containerization aka …

2 years ago • 10 comments

Is all fun and game until you are need of put it in production. Devops Borat It's time to …

Getting A Server Ready For Deployment

a year ago • 1 comment

With a PaaS, you don't get your own server, instead you're renting a "service" at a …

Working I

2 years ago • 3

Enough hous
URLs. It's ti
bullet and ch

Emoji?

5 Responses

 Upvote  Funny  Love  Surprised  Angry  Sad

4 Comments Kgotso Koete ▾

Join the discussion…

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

Gad

a year ago

Hi, Great work, thank you.
The link to the Django tutorial in the ORM part is not working. looks like they moved it to https://docs.djangoproject.com/en/5.1/intro/tutorial02/

0 0 Reply Share ›

Deepstop

2 years ago edited

I jumped on the helper functions bandwagon before you described them. I could see that the repetition would be a problem, and also the jumble of code was getting smelly. Inspired by the Dave Farley's 4 layers, I relegated them to a superclass to keep the functional test class itself clean. Now the functional test for the home page is mostly method calls referencing the superclass, and I expect that I'll introduce another level when I start working on the second user story so that I have have 3 layers. I can see the 4th layer further down the road, to allow for more variation in test cases.

Edit: The 3 layers are now a reality. (1) the bottom layer is just a method call with some parameters (2) The next layer up makes calls to navigate, find a table and test its contents, fill in a form, etc. (3) The next layer up is the only one which uses Selenium. Farley's 4th layer is application so at least in my own mind it conforms.

Each time I go through these chapters I get more out of them, after practicing on my own projects.

0 0 Reply Share ›

SSteve

2 years ago

Typo: "If get rid of that and finish writing our FT" is missing a pronoun.

0 0 Reply Share ›

hjwp Mod ➚ SSteve
2 years ago

fixed, thank you!