**Saving User Input: Testing the Database**

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

We want to take the to-do item input from the user and send it to the server, so that we can save it somehow and display it back to her later.

As I started writing this chapter, I immediately skipped to what I thought was the right design: multiple models 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 all those problems at once, the point of TDD is to allow 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 a little further.

It’s a demonstration of how TDD can support an iterative style of development—​it may not be the quickest route, but you do get there in the end. There’s a neat side benefit, which is that it allows 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. 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 with method="POST".

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

lists/templates/home.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">

Now, running our FTs gives us a slightly cryptic, unexpected error:

$ **python functional\_tests.py**

[...]

Traceback (most recent call last):

File "functional\_tests.py", line 40, in

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

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

[...]

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

element: [id="id\_list\_table"]

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[[1](http://www.obeythetestinggoat.com/book/chapter_post_and_database.html#_footnote_1)].

We’ll look at all of these over the course of this book, but the time.sleep option is one I find myself using very often. Let’s try it now.

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

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(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 functional tests 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](http://www.obeythetestinggoat.com/book/chapter_post_and_database.html#csrf_error_screenshot), with lots of Django debug information.

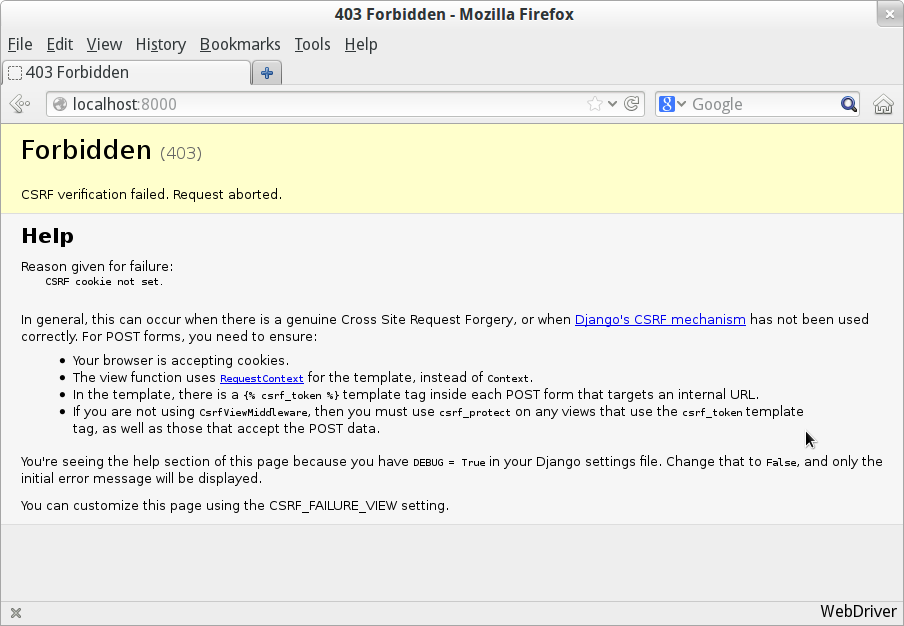


Figure 1. Django DEBUG page showing CSRF error

Security: Surprisingly Fun!

If you’ve never heard of a *Cross-Site Request Forgery* 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 back 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*. 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 for my course, Ross Anderson’s [*Security Engineering*](http://www.obeythetestinggoat.com/book/bibliography.html#seceng). 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 auto-generated token into each generated form, to be able to identify POST requests as having come from the original site. 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

<form method="POST">

<input name="item\_text" id="id\_new\_item" placeholder="Enter a to-do item" />

{% csrf\_token %}

</form>

Django will substitute that during rendering with an <input type="hidden"> containing the CSRF token. Rerunning the functional test will now give us an expected failure:

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

Since 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

# "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')

**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. 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 (ch05l005)

def test\_uses\_home\_template(self):

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

self.assertTemplateUsed(response, 'home.html')

def test\_can\_save\_a\_POST\_request(self):

response = self.client.post('/', data={'item\_text': 'A new list item'})

self.assertIn('A new list item', response.content.decode())

To do a POST, we call self.client.post, and as you can see it takes a data argument which contains the form data we want to send. 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: 'A new list item' not found in '<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"

[...]

</body>\n</html>\n'

We can get the test to pass by adding an if and providing a different code path for POST requests. In typical TDD style, we start with a deliberately silly return value:

lists/views.py

from django.http import HttpResponse

from django.shortcuts import render

def home\_page(request):

if request.method == 'POST':

return HttpResponse(request.POST['item\_text'])

return render(request, 'home.html')

That gets our unit tests passing, but it’s not really what we want. What we really want to do is add the POST submission to the table in the home page 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

<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>

</table>

</body>

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

lists/tests.py

def test\_can\_save\_a\_POST\_request(self):

response = self.client.post('/', data={'item\_text': 'A new list item'})

self.assertIn('A new list item', response.content.decode())

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:

lists/views.py (ch05l009)

def home\_page(request):

return render(request, 'home.html', {

'new\_item\_text': request.POST['item\_text'],

})

Running the unit tests again:

ERROR: test\_uses\_home\_template (lists.tests.HomePageTest)

[...]

File "...python-tdd-book/lists/views.py", line 5, in home\_page

'new\_item\_text': request.POST['item\_text'],

[...]

django.utils.datastructures.MultiValueDictKeyError: "'item\_text'"

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, 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, and we can get on with fixing it straight away. Here’s how:

lists/views.py

def home\_page(request):

return render(request, 'home.html', {

'new\_item\_text': request.POST.get('item\_text', ''),

})

Look up [dict.get](http://docs.python.org/3/library/stdtypes.html#dict.get) if you’re not sure what’s going on there.

The unit tests should now pass. Let’s see what the functional tests 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 2 or 3 seconds instead of 1; then read on to the next chapter for a more robust solution. |

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 (ch05l011)

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}"

)

|  |  |
| --- | --- |
|  | If you’ve not seen this syntax before, it’s the new Python "f-string" syntax (probably the most exciting new feature from Python 3.6). You just prepend a string with an f, and then you can use the curly-bracket syntax to insert local variables. There’s more info in the [Python 3.6 release notes](https://docs.python.org/3/whatsnew/3.6.html#pep-498-formatted-string-literals). |

That gives us a more helpful error message:

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

Contents were:

Buy peacock feathers

You know what could be even better than that? Making that assertion a bit less clever. As you may remember, 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 (ch05l012)

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

Much better. You should always be very worried whenever you think you’re being clever, because what you’re probably being is *overcomplicated*. And 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? Without it, the user’s input won’t be associated with the right key 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 a quick "cheating" change to the template:

lists/templates/home.html

<tr><td>1: {{ new\_item\_text }}</td></tr>

Red/Green/Refactor and Triangulation

The unit-test/code cycle is sometimes taught as *Red, Green, Refactor*:

* Start by writing a unit test which fails (*Red*).
* Write the simplest possible code to get it to pass (*Green*), *even if that means cheating*.
* *Refactor* to get to better code that makes more sense.

So what do we do during the Refactor stage? What justifies moving from an implementation where we "cheat" to one we’re happy with?

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

I find that leaves things a little too vague, so I usually 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, *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.

Now we get to the self.fail('Finish the test!'). If we extend our FT to 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

# 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('1: Buy peacock feathers', [row.text for row in rows])

self.assertIn(

'2: Use peacock feathers to make a fly',

[row.text for row in rows]

)

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

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

# explanatory text to that effect.

self.fail('Finish the test!')

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

Sure enough, the functional tests return an error:

AssertionError: '1: Buy peacock feathers' not found in ['1: Use peacock

feathers to make a fly']

**Three Strikes and Refactor**

Before we go further—​we’ve got a bad *code smell*[[2](http://www.obeythetestinggoat.com/book/chapter_post_and_database.html#_footnote_2)] in this FT. We have three almost identical code blocks checking for new items in the list table. There’s a principle called *Don’t Repeat Yourself* (DRY), which we like to apply by following the mantra *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 remove duplication.

We 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**

Back to our functional test refactor: we could use an inline function, but that upsets the flow of the test slightly. Let’s use a helper method—​remember, only methods that begin with test\_ will get run as tests, so you can use other methods for your own purposes:

functional\_tests.py

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\_list\_and\_retrieve\_it\_later(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

# 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('1: Buy peacock feathers')

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

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

[...]

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

AssertionError: '1: Buy peacock feathers' 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**

And 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 which 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, since it exercises code by specifying how we want it to work.

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

lists/tests.py

from lists.models import Item

[...]

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/1.11/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". We’ll actually rewrite this test to be much more concise [later on](http://www.obeythetestinggoat.com/book/chapter_advanced_forms.html#rewrite-model-test), in [[chapter\_advanced\_forms]](http://www.obeythetestinggoat.com/book/chapter_advanced_forms.html). |

Terminology 2: Unit Tests Versus Integrated Tests, and the Database

Purists 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 integrated 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 functional tests 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 and talk about unit tests and integrated tests in [[chapter\_purist\_unit\_tests]](http://www.obeythetestinggoat.com/book/chapter_purist_unit_tests.html), towards the end of the book.

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

ImportError: cannot import name 'Item'

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

from django.db import models

class Item(object):

pass

That gets our test as far as:

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

from django.db import models

class Item(models.Model):

pass

**Our First Database Migration**

The next thing that happens is a database error:

django.db.utils.OperationalError: no such table: lists\_item

In Django, the ORM’s job is to model the database, but there’s a second system that’s in charge of actually building the database called *migrations*. Its job is to give you the ability to add and remove 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 for your database. As we’ll see later, it comes in 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:[[3](http://www.obeythetestinggoat.com/book/chapter_post_and_database.html#_footnote_3)]

$ **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 our tests get a little further.

**The Test Gets Surprisingly Far**

The test actually gets surprisingly far:

$ **python manage.py test lists**

[...]

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 Items, and we’ve checked that they’re saved in the database, but Django just doesn’t seem to have remembered the .text attribute.

Incidentally, if you’re new to Python, you might have been surprised 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 map to tables in the database. By default they get an auto-generated id attribute, which will be a primary key column in the database, but you have to define any other columns you want explicitly; here’s how we set up a text field:

lists/models.py

class Item(models.Model):

text = models.TextField()

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](http://bit.ly/1slDAGH) and in the [documentation](https://docs.djangoproject.com/en/1.11/ref/models/fields/).

**A New Field Means a New Migration**

Running the tests gives us another database error:

django.db.utils.OperationalError: no such column: lists\_item.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**

You are trying to add a non-nullable field 'text' to item without a default; we

can't do that (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 let me add a default 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

class Item(models.Model):

text = models.TextField(default='')

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 lists**

[...]

Ran 3 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**

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

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.assertIn('A new list item', response.content.decode())

self.assertTemplateUsed(response, 'home.html')

|  |  |
| --- | --- |
|  | We check that one new Item has been saved to the database. objects.count() is a shorthand for objects.all().count(). |
|  | objects.first() is the same as doing objects.all()[0]. |
|  | 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?*

Writing it down on a scratchpad like this reassures us that we won’t forget, 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

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', ''),

})

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…​they pass! Good. We can do a bit of refactoring:

lists/views.py

return render(request, 'home.html', {

'new\_item\_text': item.text

})

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 one. 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

class HomePageTest(TestCase):

[...]

def test\_only\_saves\_items\_when\_necessary(self):

self.client.get('/')

self.assertEqual(Item.objects.count(), 0)

That gives us a 1 != 0 failure. Let’s fix it. Watch out; although it’s quite a small change to the logic of the view, there are quite a few little tweaks to the implementation in code:

lists/views.py

def home\_page(request):

if request.method == 'POST':

new\_item\_text = request.POST['item\_text']

Item.objects.create(text=new\_item\_text)

else:

new\_item\_text = ''

return render(request, 'home.html', {

'new\_item\_text': new\_item\_text,

})

|  |  |
| --- | --- |
|  | We use a variable called new\_item\_text, which will either hold the POST contents, or the empty string. |
|  | .objects.create is a neat shorthand for creating a new Item, without needing to call .save(). |

And that gets the test passing:

Ran 4 tests in 0.010s

OK

**Redirect After a POST**

But, yuck, that whole new\_item\_text = '' dance is making me pretty unhappy. Thankfully we now have an 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 users' input to the database, so now let’s work on the second part.

[Always redirect after a POST](https://en.wikipedia.org/wiki/Post/Redirect/Get), they say, so let’s do that. Once again we change our unit test for saving a POST request to say that, instead of rendering a response with the item in it, it should redirect back to the home page:

lists/tests.py

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.assertEqual(response.status\_code, 302)

self.assertEqual(response['location'], '/')

We no longer expect a response with a .content rendered by a template, so we lose the assertions that look at that. Instead, the response will represent an HTTP *redirect*, which should have status code 302, and points the browser towards a new location.

That gives us the error 200 != 302. We can now tidy up our view substantially:

lists/views.py (ch05l028)

from django.shortcuts import redirect, render

from lists.models import Item

def home\_page(request):

if request.method == 'POST':

Item.objects.create(text=request.POST['item\_text'])

return redirect('/')

return render(request, 'home.html')

And the tests should now pass:

Ran 4 tests in 0.010s

OK

**Better Unit Testing Practice: Each Test Should Test One Thing**

Our view now does a redirect after a POST, which is good practice, and we’ve shortened the unit test somewhat, but we can still do better.

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 status of the later assertions is. 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

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.assertEqual(response.status\_code, 302)

self.assertEqual(response['location'], '/')

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

Ran 5 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?*
* *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 item is the last of the "easy" ones. Let’s have a new unit test that checks that the template can also display multiple list items:

lists/tests.py

class HomePageTest(TestCase):

[...]

def test\_displays\_all\_list\_items(self):

Item.objects.create(text='itemey 1')

Item.objects.create(text='itemey 2')

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

self.assertIn('itemey 1', response.content.decode())

self.assertIn('itemey 2', response.content.decode())

|  |  |
| --- | --- |
|  | Are you wondering about the line spacing in the test? I’m grouping together two lines at the beginning which set up the test, one line in the middle which actually calls the code under test, and the assertions at the end. This isn’t obligatory, but it does help see the structure of the test. Setup, Exercise, Assert is the typical structure for a unit test. |

That fails as expected:

AssertionError: 'itemey 1' not found in '<html>\n <head>\n [...]

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

lists/templates/home.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/1.11/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

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 functional test 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](http://www.obeythetestinggoat.com/book/chapter_post_and_database.html#operationalerror).

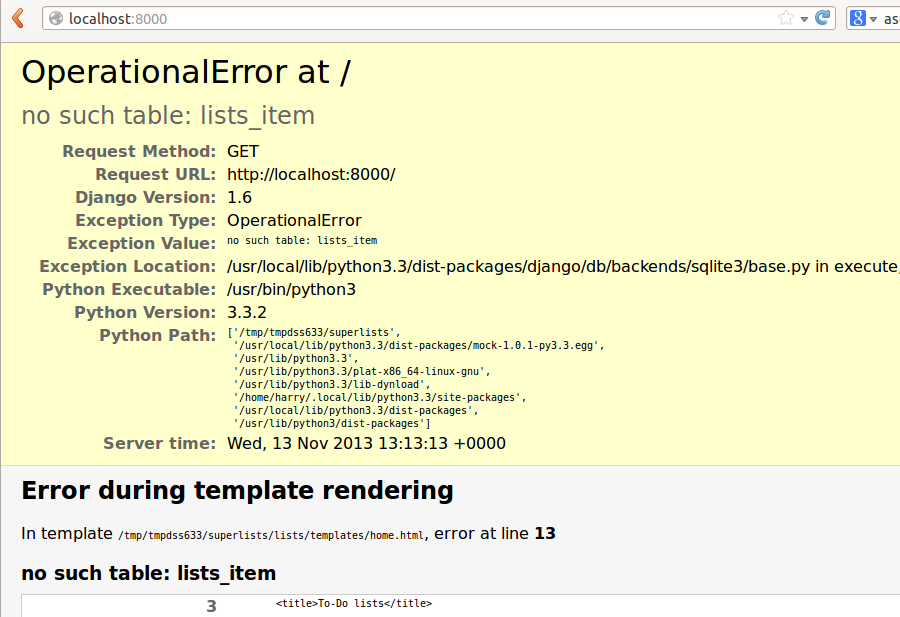


Figure 2. Another helpful debug message

**Creating Our Production Database with migrate**

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 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

[...]

# Database

# https://docs.djangoproject.com/en/1.11/ref/settings/#databases

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.sqlite3',

'NAME': os.path.join(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 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 lists.0001\_initial... OK

Applying lists.0002\_item\_text... OK

Applying sessions.0001\_initial... OK

Now we can refresh the page on *localhost*, see that our error is gone, and try running the functional tests again:[[4](http://www.obeythetestinggoat.com/book/chapter_post_and_database.html#_footnote_4)]

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

{% for item in items %}

<tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>

{% endfor %}

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

self.fail('Finish the test!')

AssertionError: Finish the test!

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](http://www.obeythetestinggoat.com/book/chapter_post_and_database.html#items_left_over_from_previous_run).

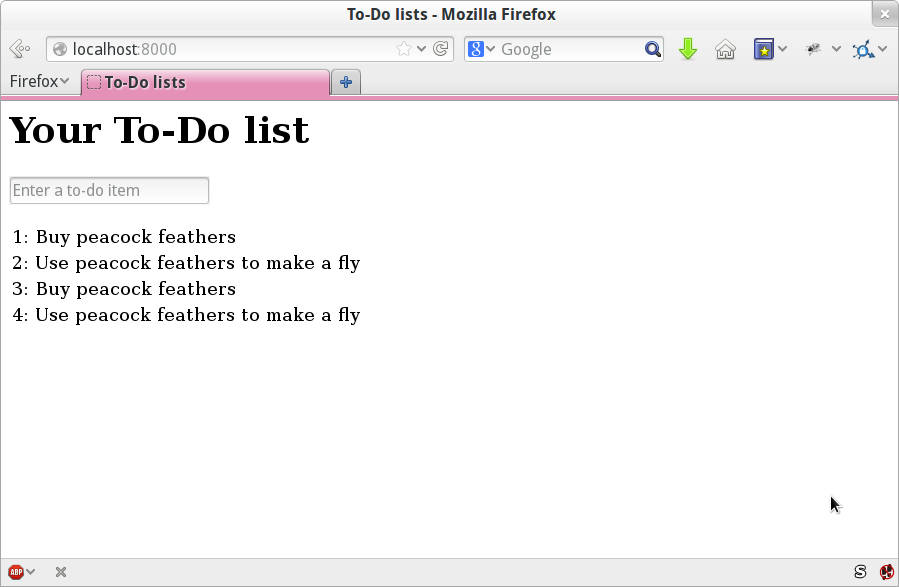


Figure 3. There are list items left over from the last run of the test

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:

$ **rm db.sqlite3**

$ **python manage.py migrate --noinput**

And then 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?

* 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 at least three different FT debugging techniques: in-line print statements, time.sleeps, 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, and perhaps more critically, adding support for more than one list.

* *Don’t save blank items for every request*
* *Code smell: POST test is too long?*
* *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 new code breaks some aspect of the application which 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.

Red/Green/Refactor

Another way of describing the TDD process. Write a test and see it fail (Red), write some code to get it to pass (Green), then Refactor to improve the implementation.

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, re-usable 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.

[1](http://www.obeythetestinggoat.com/book/chapter_post_and_database.html#_footnoteref_1). Lots of people also swear by using pdb.set\_trace() to be able to drop into a debugger, particularly for unit tests. I’m not enough of a pdb user to be able to give a good intro to it, but you should definitely check it out at some point in your testing career.

[2](http://www.obeythetestinggoat.com/book/chapter_post_and_database.html#_footnoteref_2). 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](http://www.codinghorror.com/blog/2006/05/code-smells.html). The more experience you gain as a programmer, the more fine-tuned your nose becomes to code smells…​

[3](http://www.obeythetestinggoat.com/book/chapter_post_and_database.html#_footnoteref_3). Are you wondering about when we’re going to run "migrate" as well as "makemigrations"? Read on; that’s coming up later in the chapter.

[4](http://www.obeythetestinggoat.com/book/chapter_post_and_database.html#_footnoteref_4). 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.

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