**7. Working Incrementally**

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

Now let’s address our real problem, which is that our design only allows for one global list. In this chapter I’ll demonstrate a critical TDD technique: how to adapt existing code using an incremental, step-by-step process which takes you from working state to working state. Testing Goat, not Refactoring Cat.

**7.1. Small Design When Necessary**

Let’s have a think about how we want support for multiple lists to work. Currently the FT (which is the closest we have to a design document) says this:

functional\_tests/tests.py

# 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.

# Satisfied, she goes back to sleep

But really we want to expand on this, by saying that different users don’t see each other’s lists, and each get their own URL as a way of going back to their saved lists. What might a new design look like?

**Not Big Design Up Front**

TDD is closely associated with the agile movement in software development, which includes a reaction against *Big Design Up Front*, the traditional software engineering practice whereby, after a lengthy requirements gathering exercise, there is an equally lengthy design stage where the software is planned out on paper. The agile philosophy is that you learn more from solving problems in practice than in theory, especially when you confront your application with real users as soon as possible. Instead of a long up-front design phase, we try to put a *minimum viable application* out there early, and let the design evolve gradually based on feedback from real-world usage.

But that doesn’t mean that thinking about design is outright banned! In the last big chapter we saw how just blundering ahead without thinking can *eventually* get us to the right answer, but often a little thinking about design can help us get there faster. So, let’s think about our minimum viable lists app, and what kind of design we’ll need to deliver it:

* We want each user to be able to store their own list—​at least one, for now.
* A list is made up of several items, whose primary attribute is a bit of descriptive text.
* We need to save lists from one visit to the next. For now, we can give each user a unique URL for their list. Later on we may want some way of automatically recognising users and showing them their lists.

To deliver the "for now" items, it sounds like we’re going to store lists and their items in a database. Each list will have a unique URL, and each list item will be a bit of descriptive text, associated with a particular list.

**YAGNI!**

Once you start thinking about design, it can be hard to stop. All sorts of other thoughts are occurring to us—​we might want to give each list a name or title, we might want to recognise users using usernames and passwords, we might want to add a longer notes field as well as short descriptions to our list, we might want to store some kind of ordering, and so on. But we obey another tenet of the agile gospel: "YAGNI" (pronounced yag-knee), which stands for "You ain’t gonna need it!" As software developers, we have fun creating things, and sometimes it’s hard to resist the urge to build things just because an idea occurred to us and we *might* need it. The trouble is that more often than not, no matter how cool the idea was, you *won’t* end up using it. Instead you have a load of unused code, adding to the complexity of your application. YAGNI is the mantra we use to resist our overenthusiastic creative urges.

**REST (ish)**

We have an idea of the data structure we want—​the Model part of Model-View-Controller (MVC). What about the View and Controller parts? How should the user interact with Lists and their Items using a web browser?

Representational State Transfer (REST) is an approach to web design that’s usually used to guide the design of web-based APIs. When designing a user-facing site, it’s not possible to stick *strictly* to the REST rules, but they still provide some useful inspiration (skip ahead to [[appendix\_rest\_api]](http://www.obeythetestinggoat.com/book/appendix_rest_api.html) if you want to see a real REST API).

REST suggests that we have a URL structure that matches our data structure, in this case lists and list items. Each list can have its own URL:

/lists/<list identifier>/

That will fulfill the requirement we’ve specified in our FT. To view a list, we use a GET request (a normal browser visit to the page).

To create a brand new list, we’ll have a special URL that accepts POST requests:

/lists/new

To add a new item to an existing list, we’ll have a separate URL, to which we can send POST requests:

/lists/<list identifier>/add\_item

(Again, we’re not trying to perfectly follow the rules of REST, which would use a PUT request here—​we’re just using REST for inspiration. Apart from anything else, you can’t use PUT in a standard HTML form.)

In summary, our scratchpad for this chapter looks something like this:

* *Adjust model so that items are associated with different lists*
* *Add unique URLs for each list*
* *Add a URL for creating a new list via POST*
* *Add URLs for adding a new item to an existing list via POST*

**Implementing the New Design Incrementally Using TDD**

How do we use TDD to implement the new design? Let’s take another look at the flowchart for the TDD process in [The TDD process with functional and unit tests](http://www.obeythetestinggoat.com/book/chapter_working_incrementally.html#TDD-double-loop).

At the top level, we’re going to use a combination of adding new functionality (by adding a new FT and writing new application code), and refactoring our application—​that is, rewriting some of the existing implementation so that it delivers the same functionality to the user but using aspects of our new design. We’ll be able to use the existing functional test to verify we don’t break what already works, and the new functional test to drive the new features.

At the unit test level, we’ll be adding new tests or modifying existing ones to test for the changes we want, and we’ll be able to similarly use the unit tests we don’t touch to help make sure we don’t break anything in the process.

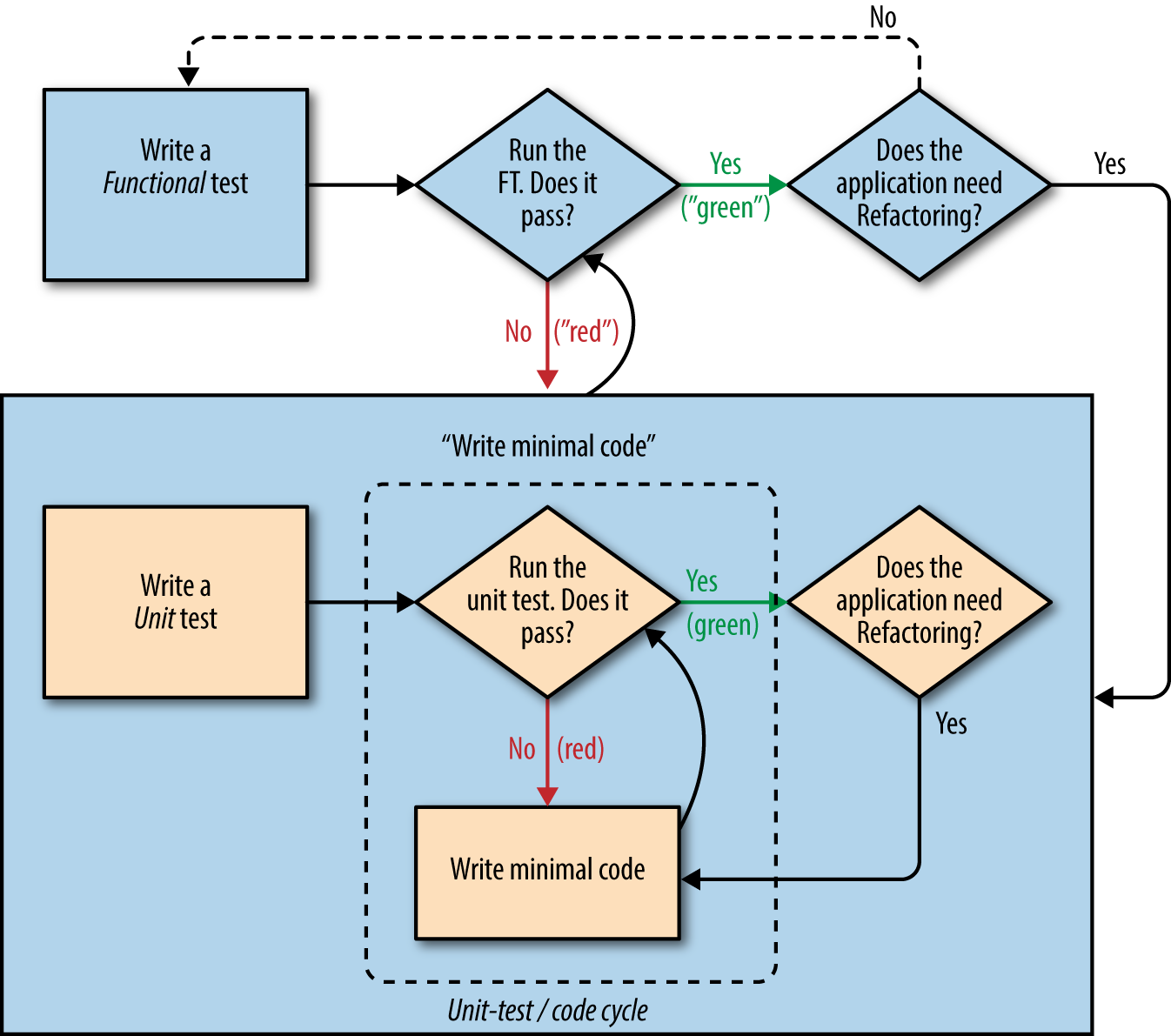


Figure 1. The TDD process with functional and unit tests

**Ensuring We Have a Regression Test**

Let’s translate our scratchpad into a new functional test method, which introduces a second user and checks that their to-do list is separate from Edith’s.

We’ll start out very similarly to the first. Edith adds a first item to create a to-do list, but we introduce our first new assertion—Edith’s list should live at its own, unique URL:

functional\_tests/tests.py (ch07l005)

def test\_can\_start\_a\_list\_for\_one\_user(self):

# Edith has heard about a cool new online to-do app. She goes

[...]

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

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

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

# Satisfied, she goes back to sleep

def test\_multiple\_users\_can\_start\_lists\_at\_different\_urls(self):

# Edith starts a new to-do list

self.browser.get(self.live\_server\_url)

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

inputbox.send\_keys('Buy peacock feathers')

inputbox.send\_keys(Keys.ENTER)

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

# She notices that her list has a unique URL

edith\_list\_url = self.browser.current\_url

self.assertRegex(edith\_list\_url, '/lists/.+')

|  |  |
| --- | --- |
|  | assertRegex is a helper function from unittest that checks whether a string matches a regular expression. We use it to check that our new REST-ish design has been implemented. Find out more in the [unittest documentation](http://docs.python.org/3/library/unittest.html). |

Next we imagine a new user coming along. We want to check that they don’t see any of Edith’s items when they visit the home page, and that they get their own unique URL for their list:

functional\_tests/tests.py (ch07l006)

[...]

self.assertRegex(edith\_list\_url, '/lists/.+')

# Now a new user, Francis, comes along to the site.

## We use a new browser session to make sure that no information

## of Edith's is coming through from cookies etc

self.browser.quit()

self.browser = webdriver.Firefox()

# Francis visits the home page. There is no sign of Edith's

# list

self.browser.get(self.live\_server\_url)

page\_text = self.browser.find\_element\_by\_tag\_name('body').text

self.assertNotIn('Buy peacock feathers', page\_text)

self.assertNotIn('make a fly', page\_text)

# Francis starts a new list by entering a new item. He

# is less interesting than Edith...

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

inputbox.send\_keys('Buy milk')

inputbox.send\_keys(Keys.ENTER)

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

# Francis gets his own unique URL

francis\_list\_url = self.browser.current\_url

self.assertRegex(francis\_list\_url, '/lists/.+')

self.assertNotEqual(francis\_list\_url, edith\_list\_url)

# Again, there is no trace of Edith's list

page\_text = self.browser.find\_element\_by\_tag\_name('body').text

self.assertNotIn('Buy peacock feathers', page\_text)

self.assertIn('Buy milk', page\_text)

# Satisfied, they both go back to sleep

|  |  |
| --- | --- |
|  | I’m using the convention of double-hashes (##) to indicate "meta-comments"—comments about *how* the test is working and why—​so that we can distinguish them from regular comments in FTs which explain the User Story. They’re a message to our future selves, which might otherwise be wondering why the heck we’re quitting the browser and starting a new one…​ |

Other than that, the new test is fairly self-explanatory. Let’s see how we do when we run our FTs:

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

[...]

.F

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

FAIL: test\_multiple\_users\_can\_start\_lists\_at\_different\_urls

(functional\_tests.tests.NewVisitorTest)

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

Traceback (most recent call last):

File "...python-tdd-book/functional\_tests/tests.py", line 83, in

test\_multiple\_users\_can\_start\_lists\_at\_different\_urls

self.assertRegex(edith\_list\_url, '/lists/.+')

AssertionError: Regex didn't match: '/lists/.+' not found in

'http://localhost:8081/'

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

Ran 2 tests in 5.786s

FAILED (failures=1)

Good, our first test still passes, and the second one fails where we might expect. Let’s do a commit, and then go and build some new models and views:

$ **git commit -a**

**Iterating Towards the New Design**

Being all excited about our new design, I had an overwhelming urge to dive in at this point and start changing *models.py*, which would have broken half the unit tests, and then pile in and change almost every single line of code, all in one go. That’s a natural urge, and TDD, as a discipline, is a constant fight against it. Obey the Testing Goat, not Refactoring Cat! We don’t need to implement our new, shiny design in a single big bang. Let’s make small changes that take us from a working state to a working state, with our design guiding us gently at each stage.

There are four items on our to-do list. The FT, with its Regexp didn't match, is telling us that the second item—​giving lists their own URL and identifier—​is the one we should work on next. Let’s have a go at fixing that, and only that.

The URL comes from the redirect after POST. In *lists/tests.py*, find test\_redirects\_after\_POST, and change the expected redirect location:

lists/tests.py

self.assertEqual(response.status\_code, 302)

self.assertEqual(response['location'], '/lists/the-only-list-in-the-world/')

Does that seem slightly strange? Clearly, */lists/the-only-list-in-the-world* isn’t a URL that’s going to feature in the final design of our application. But we’re committed to changing one thing at a time. While our application only supports one list, this is the only URL that makes sense. We’re still moving forwards, in that we’ll have a different URL for our list and our home page, which is a step along the way to a more REST-ful design. Later, when we have multiple lists, it will be easy to change.

|  |  |
| --- | --- |
|  | Another way of thinking about it is as a problem-solving technique: our new URL design is currently not implemented, so it works for 0 items. Ultimately, we want to solve for *n* items, but solving for 1 item is a good step along the way. |

Running the unit tests gives us an expected fail:

$ **python manage.py test lists**

[...]

AssertionError: '/' != '/lists/the-only-list-in-the-world/'

We can go adjust our home\_page view in *lists/views.py*:

lists/views.py

def home\_page(request):

if request.method == 'POST':

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

return redirect('/lists/the-only-list-in-the-world/')

items = Item.objects.all()

return render(request, 'home.html', {'items': items})

Of course, that will now totally break the functional tests, because there is no such URL on our site yet. Sure enough, if you run them, you’ll find they fail just after trying to submit the first item, saying that they can’t find the list table; it’s because the URL */the-only-list-in-the-world/* doesn’t exist yet!

File "...python-tdd-book/functional\_tests/tests.py", line 57, in

test\_can\_start\_a\_list\_for\_one\_user

[...]

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

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

[...]

File "...python-tdd-book/functional\_tests/tests.py", line 79, in

test\_multiple\_users\_can\_start\_lists\_at\_different\_urls

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

[...]

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

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

Not only is our new test failing, but the old one is too. That tells us we’ve introduced a *regression*. Let’s try to get back to a working state as quickly as possible by building a URL for our one and only list.

**Taking a First, Self-Contained Step: One New URL**

Open up *lists/tests.py*, and add a new test class called ListViewTest. Then copy the method called test\_displays\_all\_list\_items across from HomePageTest into our new class, rename it, and adapt it slightly:

lists/tests.py (ch07l009)

class ListViewTest(TestCase):

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

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

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

response = self.client.get('/lists/the-only-list-in-the-world/')

self.assertContains(response, 'itemey 1')

self.assertContains(response, 'itemey 2')

|  |  |
| --- | --- |
|  | Here’s a new helper method: instead of using the slightly annoying assertIn/response.content.decode() dance, Django provides the assertContains method, which knows how to deal with responses and the bytes of their content. |

Let’s try running this test now:

self.assertContains(response, 'itemey 1')

[...]

AssertionError: 404 != 200 : Couldn't retrieve content: Response code was 404

Here’s a nice side effect of using assertContains: it tells us straight away that the test is failing because our new URL doesn’t exist yet, and is returning a 404.

**A New URL**

Our singleton list URL doesn’t exist yet. We fix that in *superlists/urls.py*.

|  |  |
| --- | --- |
|  | Watch out for trailing slashes in URLs, both here in the tests and in *urls.py*. They’re a common source of bugs. |

superlists/urls.py

urlpatterns = [

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

url(r'^lists/the-only-list-in-the-world/$', views.view\_list, name='view\_list'),

]

Running the tests again, we get:

AttributeError: module 'lists.views' has no attribute 'view\_list'

**A New View Function**

Nicely self-explanatory. Let’s create a dummy view function in *lists/views.py*:

lists/views.py

def view\_list(request):

pass

Now we get:

ValueError: The view lists.views.view\_list didn't return an HttpResponse

object. It returned None instead.

[...]

FAILED (errors=1)

Down to just one failure, and it’s pointing us in the right direction. Let’s copy the two last lines from the home\_page view and see if they’ll do the trick:

lists/views.py

def view\_list(request):

items = Item.objects.all()

return render(request, 'home.html', {'items': items})

Rerun the unit tests and they should pass:

Ran 7 tests in 0.016s

OK

Now let’s try the FTs again and see what they tell us:

FAIL: test\_can\_start\_a\_list\_for\_one\_user

[...]

File "...python-tdd-book/functional\_tests/tests.py", line 67, in

test\_can\_start\_a\_list\_for\_one\_user

[...]

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

peacock feathers']

FAIL: test\_multiple\_users\_can\_start\_lists\_at\_different\_urls

[...]

AssertionError: 'Buy peacock feathers' unexpectedly found in 'Your To-Do

list\n1: Buy peacock feathers'

[...]

Both of them are getting a little further than they were before, but they’re still failing. It would be nice to get back to a working state and get that first one passing again. What’s it trying to tell us?

It’s failing when we try to add the second item. We have to put our debugging hats on here. We know the home page is working, because the test has got all the way down to line 67 in the FT, so we’ve at least added a first item. And our unit tests are all passing, so we’re pretty sure the URLs and views are doing what they should—​the home page displays the right template, and can handle POST requests, and the *only-list-in-the-world* view knows how to display all items…​but it doesn’t know how to handle POST requests. Ah, that gives us a clue.

A second clue is the rule of thumb that, when all the unit tests are passing but the functional tests aren’t, it’s often pointing at a problem that’s not covered by the unit tests, and in our case, that’s often a template problem.

The answer is that our *home.html* input form currently doesn’t specify an explicit URL to POST to:

lists/templates/home.html

<form method="POST">

By default the browser sends the POST data back to the same URL it’s currently on. When we’re on the home page that works fine, but when we’re on our *only-list-in-the-world* page, it doesn’t.

Now we could dive in and add POST request handling to our new view, but that would involve writing a bunch more tests and code, and at this point we’d like to get back to a working state as quickly as possible. Actually the quickest thing we can do to get things fixed is to just use the existing home page view, which already works, for all POST requests:

lists/templates/home.html

<form method="POST" action="/">

Try that, and we’ll see our FTs get back to a happier place:

FAIL: test\_multiple\_users\_can\_start\_lists\_at\_different\_urls

[...]

AssertionError: 'Buy peacock feathers' unexpectedly found in 'Your To-Do

list\n1: Buy peacock feathers'

Ran 2 tests in 8.541s

FAILED (failures=1)

Our original test passes once again, so we know we’re back to a working state. The new functionality may not be working yet, but at least the old stuff works as well as it used to.

**Green? Refactor**

Time for a little tidying up.

In the *Red/Green/Refactor* dance, we’ve arrived at green, so we should see what needs a refactor. We now have two views, one for the home page, and one for an individual list. Both are currently using the same template, and passing it all the list items currently in the database. If we look through our unit test methods, we can see some stuff we probably want to change:

$ **grep -E "class|def" lists/tests.py**

class HomePageTest(TestCase):

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

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

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

def test\_redirects\_after\_POST(self):

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

class ListViewTest(TestCase):

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

class ItemModelTest(TestCase):

def test\_saving\_and\_retrieving\_items(self):

We can definitely delete the test\_displays\_all\_list\_items method from HomePageTest; it’s no longer needed. If you run **manage.py test lists** now, it should say it ran 6 tests instead of 7:

Ran 6 tests in 0.016s

OK

Next, since we don’t actually need the home page template to display all list items any more, it should just show a single input box inviting you to start a new list.

**Another Small Step: A Separate Template for Viewing Lists**

Since the home page and the list view are now quite distinct pages, they should be using different HTML templates; *home.html* can have the single input box, whereas a new template, *list.html*, can take care of showing the table of existing items.

Let’s add a new test to check that it’s using a different template:

lists/tests.py

class ListViewTest(TestCase):

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

response = self.client.get('/lists/the-only-list-in-the-world/')

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

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

[...]

assertTemplateUsed is one of the more useful functions that the Django Test Client gives us. Let’s see what it says:

AssertionError: False is not true : Template 'list.html' was not a template

used to render the response. Actual template(s) used: home.html

Great! Let’s change the view:

lists/views.py

def view\_list(request):

items = Item.objects.all()

return render(request, 'list.html', {'items': items})

But, obviously, that template doesn’t exist yet. If we run the unit tests, we get:

django.template.exceptions.TemplateDoesNotExist: list.html

Let’s create a new file at *lists/templates/list.html*:

$ **touch lists/templates/list.html**

A blank template, which gives us this error—​good to know the tests are there to make sure we fill it in:

AssertionError: False is not true : Couldn't find 'itemey 1' in response

The template for an individual list will reuse quite a lot of the stuff we currently have in *home.html*, so we can start by just copying that:

$ **cp lists/templates/home.html lists/templates/list.html**

That gets the tests back to passing (green). Now let’s do a little more tidying up (refactoring). We said the home page doesn’t need to list items, it only needs the new list input field, so we can remove some lines from *lists/templates/home.html*, and maybe slightly tweak the h1 to say "Start a new To-Do list":

lists/templates/home.html

<body>

<h1>Start a new To-Do list</h1>

<form method="POST">

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

{% csrf\_token %}

</form>

</body>

We rerun the unit tests to check that hasn’t broken anything—​good…​

There’s actually no need to pass all the items to the *home.html* template in our home\_page view, so we can simplify that:

lists/views.py

def home\_page(request):

if request.method == 'POST':

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

return redirect('/lists/the-only-list-in-the-world/')

return render(request, 'home.html')

Rerun the unit tests once more; they still pass. Time to run the functional tests:

AssertionError: '1: Buy milk' not found in ['1: Buy peacock feathers', '2: Buy

milk']

Not bad! Our regression test (the first FT) is passing, and our new test is now getting slightly further forwards—​it’s telling us that Francis isn’t getting his own list page (because he still sees some of Edith’s list items).

It may feel like we haven’t made much headway since, functionally, the site still behaves almost exactly like it did when we started the chapter, but this really is progress. We’ve started on the road to our new design, and we’ve implemented a number of stepping stones *without making anything worse than it was before*. Let’s commit our progress so far:

$ **git status** # should show 4 changed files and 1 new file, list.html

$ **git add lists/templates/list.html**

$ **git diff** # should show we've simplified home.html,

# moved one test to a new class in lists/tests.py added a new view

# in views.py, and simplified home\_page and made one addition to

# urls.py

$ **git commit -a** # add a message summarising the above, maybe something like

# "new URL, view and template to display lists"

**A Third Small Step: A URL for Adding List Items**

Where are we with our own to-do list?

* *Adjust model so that items are associated with different lists*
* *Add unique URLs for each list* …​
* *Add a URL for creating a new list via POST*
* *Add URLs for adding a new item to an existing list via POST*

We’ve *sort of* made progress on the second item, even if there’s still only one list in the world. The first item is a bit scary. Can we do something about items 3 or 4?

Let’s have a new URL for adding new list items. If nothing else, it’ll simplify the home page view.

**A Test Class for New List Creation**

Open up *lists/tests.py*, and *move* the test\_can\_save\_a\_POST\_request and test\_redirects\_after\_POST methods into a new class, then change the URL they POST to:

lists/tests.py (ch07l021-1)

class NewListTest(TestCase):

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

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

self.assertEqual(response.status\_code, 302)

self.assertEqual(response['location'], '/lists/the-only-list-in-the-world/')

|  |  |
| --- | --- |
|  | This is another place to pay attention to trailing slashes, incidentally. It’s /new, with no trailing slash. The convention I’m using is that URLs without a trailing slash are "action" URLs which modify the database. |

While we’re at it, let’s learn a new Django Test Client method, assertRedirects:

lists/tests.py (ch07l021-2)

def test\_redirects\_after\_POST(self):

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

self.assertRedirects(response, '/lists/the-only-list-in-the-world/')

There’s not much to it, but it just nicely replaces two asserts with a single one…​

Try running that:

self.assertEqual(Item.objects.count(), 1)

AssertionError: 0 != 1

[...]

self.assertRedirects(response, '/lists/the-only-list-in-the-world/')

[...]

AssertionError: 404 != 302 : Response didn't redirect as expected: Response

code was 404 (expected 302)

The first failure tells us we’re not saving a new item to the database, and the second says that, instead of returning a 302 redirect, our view is returning a 404. That’s because we haven’t built a URL for */lists/new*, so the client.post is just getting a "not found" response.

|  |  |
| --- | --- |
|  | Do you remember how we split this out into two tests earlier? If we only had one test that checked both the saving and the redirect, it would have failed on the 0 != 1 failure, which would have been much harder to debug. Ask me how I know this. |

**A URL and View for New List Creation**

Let’s build our new URL now:

superlists/urls.py

urlpatterns = [

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

url(r'^lists/new$', views.new\_list, name='new\_list'),

url(r'^lists/the-only-list-in-the-world/$', views.view\_list, name='view\_list'),

]

Next we get a no attribute 'new\_list', so let’s fix that, in *lists/views.py*:

lists/views.py (ch07l023-1)

def new\_list(request):

pass

Then we get "The view lists.views.new\_list didn’t return an HttpResponse object". (This is getting rather familiar!) We could return a raw HttpResponse, but since we know we’ll need a redirect, let’s borrow a line from home\_page:

lists/views.py (ch07l023-2)

def new\_list(request):

return redirect('/lists/the-only-list-in-the-world/')

That gives:

self.assertEqual(Item.objects.count(), 1)

AssertionError: 0 != 1

Seems reasonably straightforward. We borrow another line from home\_page:

lists/views.py (ch07l023-3)

def new\_list(request):

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

return redirect('/lists/the-only-list-in-the-world/')

And everything now passes:

Ran 7 tests in 0.030s

OK

And the FTs show me that I’m back to the working state:

[...]

AssertionError: '1: Buy milk' not found in ['1: Buy peacock feathers', '2: Buy

milk']

Ran 2 tests in 8.972s

FAILED (failures=1)

**Removing Now-Redundant Code and Tests**

We’re looking good. Since our new views are now doing most of the work that home\_page used to do, we should be able to massively simplify it. Can we remove the whole if request.method == 'POST' section, for example?

lists/views.py

def home\_page(request):

return render(request, 'home.html')

Yep!

OK

And while we’re at it, we can remove the now-redundant test\_only\_saves\_​items\_when\_necessary test too!

Doesn’t that feel good? The view functions are looking much simpler. We rerun the tests to make sure…​

Ran 6 tests in 0.016s

OK

and the FTs?

**A Regression! Pointing Our Forms at the New URL**

Oops:

ERROR: test\_can\_start\_a\_list\_for\_one\_user

[...]

File "...python-tdd-book/functional\_tests/tests.py", line 57, in

test\_can\_start\_a\_list\_for\_one\_user

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

File "...python-tdd-book/functional\_tests/tests.py", line 23, in

wait\_for\_row\_in\_list\_table

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

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

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

ERROR: test\_multiple\_users\_can\_start\_lists\_at\_different\_urls

[...]

File "...python-tdd-book/functional\_tests/tests.py", line 79, in

test\_multiple\_users\_can\_start\_lists\_at\_different\_urls

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

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

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

[...]

Ran 2 tests in 11.592s

FAILED (errors=2)

It’s because our forms are still pointing to the old URL. In *both* *home.html* and *lists.html*, let’s change them to:

lists/templates/home.html, lists/templates/list.html

<form method="POST" action="/lists/new">

And that should get us back to working again:

AssertionError: '1: Buy milk' not found in ['1: Buy peacock feathers', '2: Buy

milk']

[...]

FAILED (failures=1)

That’s another nicely self-contained commit, in that we’ve made a bunch of changes to our URLs, our *views.py* is looking much neater and tidier, and we’re sure the application is still working as well as it did before. We’re getting good at this working-state-to-working-state malarkey!

$ **git status** # 5 changed files

$ **git diff** # URLs for forms x2, moved code in views + tests, new URL

$ **git commit -a**

And we can cross out an item on the to-do list:

* *Adjust model so that items are associated with different lists*
* *Add unique URLs for each list*
* *Add a URL for creating a new list via POST*
* *Add URLs for adding a new item to an existing list via POST*

**Biting the Bullet: Adjusting Our Models**

Enough housekeeping with our URLs. It’s time to bite the bullet and change our models. Let’s adjust the model unit test. Just for a change, I’ll present the changes in the form of a diff:

lists/tests.py

@@ -1,5 +1,5 @@

from django.test import TestCase

-from lists.models import Item

+from lists.models import Item, List

class HomePageTest(TestCase):

@@ -44,22 +44,32 @@ class ListViewTest(TestCase):

-class ItemModelTest(TestCase):

+class ListAndItemModelsTest(TestCase):

def test\_saving\_and\_retrieving\_items(self):

+ list\_ = List()

+ list\_.save()

+

first\_item = Item()

first\_item.text = 'The first (ever) list item'

+ first\_item.list = list\_

first\_item.save()

second\_item = Item()

second\_item.text = 'Item the second'

+ second\_item.list = list\_

second\_item.save()

+ saved\_list = List.objects.first()

+ self.assertEqual(saved\_list, list\_)

+

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(first\_saved\_item.list, list\_)

self.assertEqual(second\_saved\_item.text, 'Item the second')

+ self.assertEqual(second\_saved\_item.list, list\_)

We create a new List object, and then we assign each item to it by assigning it as its .list property. We check that the list is properly saved, and we check that the two items have also saved their relationship to the list. You’ll also notice that we can compare list objects with each other directly (saved\_list and list\_)—behind the scenes, these will compare themselves by checking that their primary key (the .id attribute) is the same.

|  |  |
| --- | --- |
|  | I’m using the variable name list\_ to avoid "shadowing" the Python built-in list function. It’s ugly, but all the other options I tried were equally ugly or worse (my\_list, the\_list, list1, listey…​). |

Time for another unit-test/code cycle.

For the first couple of iterations, rather than explicitly showing you what code to enter in between every test run, I’m only going to show you the expected error messages from running the tests. I’ll let you figure out what each minimal code change should be on your own.

|  |  |
| --- | --- |
|  | Need a hint? Go back and take a look at the steps we took to introduce the Item model in [the chapter before last](http://www.obeythetestinggoat.com/book/chapter_post_and_database.html#first-django-model). |

Your first error should be:

ImportError: cannot import name 'List'

Fix that, and then you should see:

AttributeError: 'List' object has no attribute 'save'

Next you should see:

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

So we run a makemigrations:

$ **python manage.py makemigrations**

Migrations for 'lists':

lists/migrations/0003\_list.py

- Create model List

And then you should see:

self.assertEqual(first\_saved\_item.list, list\_)

AttributeError: 'Item' object has no attribute 'list'

**A Foreign Key Relationship**

How do we give our Item a list attribute? Let’s just try naively making it like the text attribute (and here’s your chance to see whether your solution so far looks like mine by the way):

lists/models.py

from django.db import models

class List(models.Model):

pass

class Item(models.Model):

text = models.TextField(default='')

list = models.TextField(default='')

As usual, the tests tell us we need a migration:

$ **python manage.py test lists**

[...]

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

$ **python manage.py makemigrations**

Migrations for 'lists':

lists/migrations/0004\_item\_list.py

- Add field list to item

Let’s see what that gives us:

AssertionError: 'List object' != <List: List object>

We’re not quite there. Look closely at each side of the !=. Django has only saved the string representation of the List object. To save the relationship to the object itself, we tell Django about the relationship between the two classes using a ForeignKey:

lists/models.py

from django.db import models

class List(models.Model):

pass

class Item(models.Model):

text = models.TextField(default='')

list = models.ForeignKey(List, default=None)

That’ll need a migration too. Since the last one was a red herring, let’s delete it and replace it with a new one:

$ **rm lists/migrations/0004\_item\_list.py**

$ **python manage.py makemigrations**

Migrations for 'lists':

lists/migrations/0004\_item\_list.py

- Add field list to item

|  |  |
| --- | --- |
|  | Deleting migrations is dangerous. We do need to do it now and again, because we don’t always get our models code right on the first go. But if you delete a migration that’s already been applied to a database somewhere, Django will be confused about what state it’s in, and how to apply future migrations. You should only do it when you’re sure the migration hasn’t been used. A good rule of thumb is that you should never delete or modify a migration that’s already been committed to your VCS. |

**Adjusting the Rest of the World to Our New Models**

Back in our tests, now what happens?

$ **python manage.py test lists**

[...]

ERROR: test\_displays\_all\_items (lists.tests.ListViewTest)

django.db.utils.IntegrityError: NOT NULL constraint failed: lists\_item.list\_id

[...]

ERROR: test\_redirects\_after\_POST (lists.tests.NewListTest)

django.db.utils.IntegrityError: NOT NULL constraint failed: lists\_item.list\_id

[...]

ERROR: test\_can\_save\_a\_POST\_request (lists.tests.NewListTest)

django.db.utils.IntegrityError: NOT NULL constraint failed: lists\_item.list\_id

Ran 6 tests in 0.021s

FAILED (errors=3)

Oh dear!

There is some good news. Although it’s hard to see, our model tests are passing. But three of our view tests are failing nastily.

The reason is because of the new relationship we’ve introduced between Items and Lists, which requires each item to have a parent list, which our old tests and code aren’t prepared for.

Still, this is exactly why we have tests! Let’s get them working again. The easiest is the ListViewTest; we just create a parent list for our two test items:

lists/tests.py (ch07l031)

class ListViewTest(TestCase):

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

list\_ = List.objects.create()

Item.objects.create(text='itemey 1', list=list\_)

Item.objects.create(text='itemey 2', list=list\_)

That gets us down to two failing tests, both on tests that try to POST to our new\_list view. Decoding the tracebacks using our usual technique, working back from error to line of test code to, buried in there somewhere, the line of our own code that caused the failure:

File "...python-tdd-book/lists/views.py", line 9, in new\_list

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

It’s when we try to create an item without a parent list. So we make a similar change in the view:

lists/views.py

from lists.models import Item, List

[...]

def new\_list(request):

list\_ = List.objects.create()

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

return redirect('/lists/the-only-list-in-the-world/')

And that gets our tests passing again:

Ran 6 tests in 0.030s

OK

Are you cringing internally at this point? *Arg! This feels so wrong; we create a new list for every single new item submission, and we’re still just displaying all items as if they belong to the same list!* I know, I feel the same. The step-by-step approach, in which you go from working code to working code, is counterintuitive. I always feel like just diving in and trying to fix everything all in one go, instead of going from one weird half-finished state to another. But remember the Testing Goat! When you’re up a mountain, you want to think very carefully about where you put each foot, and take one step at a time, checking at each stage that the place you’ve put it hasn’t caused you to fall off a cliff.

So just to reassure ourselves that things have worked, we rerun the FT:

AssertionError: '1: Buy milk' not found in ['1: Buy peacock feathers', '2: Buy

milk']

[...]

Sure enough, it gets all the way through to where we were before. We haven’t broken anything, and we’ve made a change to the database. That’s something to be pleased with! Let’s commit:

$ **git status** # 3 changed files, plus 2 migrations

$ **git add lists**

$ **git diff --staged**

$ **git commit**

And we can cross out another item on the to-do list:

* *Adjust model so that items are associated with different lists*
* *Add unique URLs for each list*
* *Add a URL for creating a new list via POST*
* *Add URLs for adding a new item to an existing list via POST*

**Each List Should Have Its Own URL**

What shall we use as the unique identifier for our lists? Probably the simplest thing, for now, is just to use the auto-generated id field from the database. Let’s change ListViewTest so that the two tests point at new URLs.

We’ll also change the old test\_displays\_all\_items test and call it test\_displays\_only\_items\_for\_that\_list instead, and make it check that only the items for a specific list are displayed:

lists/tests.py (ch07l033)

class ListViewTest(TestCase):

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

list\_ = List.objects.create()

response = self.client.get(f'/lists/{list\_.id}/')

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

def test\_displays\_only\_items\_for\_that\_list(self):

correct\_list = List.objects.create()

Item.objects.create(text='itemey 1', list=correct\_list)

Item.objects.create(text='itemey 2', list=correct\_list)

other\_list = List.objects.create()

Item.objects.create(text='other list item 1', list=other\_list)

Item.objects.create(text='other list item 2', list=other\_list)

response = self.client.get(f'/lists/{correct\_list.id}/')

self.assertContains(response, 'itemey 1')

self.assertContains(response, 'itemey 2')

self.assertNotContains(response, 'other list item 1')

self.assertNotContains(response, 'other list item 2')

|  |  |
| --- | --- |
|  | A couple more of those lovely f-strings in this listing! If they’re still a bit of a mystery, take a look at the [docs](https://docs.python.org/3/reference/lexical_analysis.html#f-strings) (although if your formal CS education is as bad as mine, you’ll probably skip the formal grammar). |

Running the unit tests gives an expected 404, and another related error:

FAIL: test\_displays\_only\_items\_for\_that\_list (lists.tests.ListViewTest)

AssertionError: 404 != 200 : Couldn't retrieve content: Response code was 404

(expected 200)

[...]

FAIL: test\_uses\_list\_template (lists.tests.ListViewTest)

AssertionError: No templates used to render the response

**Capturing Parameters from URLs**

It’s time to learn how we can pass parameters from URLs to views:

superlists/urls.py

urlpatterns = [

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

url(r'^lists/new$', views.new\_list, name='new\_list'),

url(r'^lists/(.+)/$', views.view\_list, name='view\_list'),

]

We adjust the regular expression for our URL to include a *capture group*, (.+), which will match any characters, up to the following /. The captured text will get passed to the view as an argument.

In other words, if we go to the URL */lists/1/*, view\_list will get a second argument after the normal request argument, namely the string "1". If we go to */lists/foo/*, we get view\_list(request, "foo").

But our view doesn’t expect an argument yet! Sure enough, this causes problems:

ERROR: test\_displays\_only\_items\_for\_that\_list (lists.tests.ListViewTest)

[...]

TypeError: view\_list() takes 1 positional argument but 2 were given

[...]

ERROR: test\_uses\_list\_template (lists.tests.ListViewTest)

[...]

TypeError: view\_list() takes 1 positional argument but 2 were given

[...]

ERROR: test\_redirects\_after\_POST (lists.tests.NewListTest)

[...]

TypeError: view\_list() takes 1 positional argument but 2 were given

FAILED (errors=3)

We can fix that easily with a dummy parameter in *views.py*:

lists/views.py

def view\_list(request, list\_id):

[...]

Now we’re down to our expected failure:

FAIL: test\_displays\_only\_items\_for\_that\_list (lists.tests.ListViewTest)

[...]

AssertionError: 1 != 0 : Response should not contain 'other list item 1'

Let’s make our view discriminate over which items it sends to the template:

lists/views.py

def view\_list(request, list\_id):

list\_ = List.objects.get(id=list\_id)

items = Item.objects.filter(list=list\_)

return render(request, 'list.html', {'items': items})

**Adjusting new\_list to the New World**

Oops, now we get errors in another test:

ERROR: test\_redirects\_after\_POST (lists.tests.NewListTest)

ValueError: invalid literal for int() with base 10:

'the-only-list-in-the-world'

Let’s take a look at this test then, since it’s moaning:

lists/tests.py

class NewListTest(TestCase):

[...]

def test\_redirects\_after\_POST(self):

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

self.assertRedirects(response, '/lists/the-only-list-in-the-world/')

It looks like it hasn’t been adjusted to the new world of Lists and Items. The test should be saying that this view redirects to the URL of the specific new list it just created:

lists/tests.py (ch07l036-1)

def test\_redirects\_after\_POST(self):

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

new\_list = List.objects.first()

self.assertRedirects(response, f'/lists/{new\_list.id}/')

That still gives us the *invalid literal* error. We take a look at the view itself, and change it so it redirects to a valid place:

lists/views.py (ch07l036-2)

def new\_list(request):

list\_ = List.objects.create()

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

return redirect(f'/lists/{list\_.id}/')

That gets us back to passing unit tests:

$ **python3 manage.py test lists**

[...]

......

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

Ran 6 tests in 0.033s

OK

What about the functional tests? We must be almost there?

**The Functional Tests Detect Another Regression**

Well, almost:

F.

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

FAIL: test\_can\_start\_a\_list\_for\_one\_user

(functional\_tests.tests.NewVisitorTest)

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

Traceback (most recent call last):

File "...python-tdd-book/functional\_tests/tests.py", line 67, in

test\_can\_start\_a\_list\_for\_one\_user

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

[...]

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

peacock feathers to make a fly']

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

Ran 2 tests in 8.617s

FAILED (failures=1)

Our new test is actually passing, and different users can get different lists, but the old test is warning us of a regression. It looks like you can’t add a second item to a list any more. It’s because of our quick-and-dirty hack where we create a new list for every single POST submission. This is exactly what we have functional tests for!

And it correlates nicely with the last item on our to-do list:

* *Adjust model so that items are associated with different lists*
* *Add unique URLs for each list*
* *Add a URL for creating a new list via POST*
* *Add URLs for adding a new item to an existing list via POST*

**One More View to Handle Adding Items to an Existing List**

We need a URL and view to handle adding a new item to an existing list (*/lists/<list\_id>/add\_item*). We’re getting pretty good at these now, so let’s knock one together quickly:

lists/tests.py

class NewItemTest(TestCase):

def test\_can\_save\_a\_POST\_request\_to\_an\_existing\_list(self):

other\_list = List.objects.create()

correct\_list = List.objects.create()

self.client.post(

f'/lists/{correct\_list.id}/add\_item',

data={'item\_text': 'A new item for an existing list'}

)

self.assertEqual(Item.objects.count(), 1)

new\_item = Item.objects.first()

self.assertEqual(new\_item.text, 'A new item for an existing list')

self.assertEqual(new\_item.list, correct\_list)

def test\_redirects\_to\_list\_view(self):

other\_list = List.objects.create()

correct\_list = List.objects.create()

response = self.client.post(

f'/lists/{correct\_list.id}/add\_item',

data={'item\_text': 'A new item for an existing list'}

)

self.assertRedirects(response, f'/lists/{correct\_list.id}/')

|  |  |
| --- | --- |
|  | Are you wondering about other\_list? A bit like in the tests for viewing a specific list, it’s important that we add items to a specific list. Adding this second object to the database prevents me from using a hack like List.objects.first() in the implementation. That would be a stupid thing to do, and you can go too far down the road of testing for all the stupid things you must not do (there are an infinite number of those, after all). It’s a judgement call, but this one feels worth it. There’s some more discussion of this in [[testing-for-stupidity]](http://www.obeythetestinggoat.com/book/chapter_advanced_forms.html#testing-for-stupidity). |

We get:

AssertionError: 0 != 1

[...]

AssertionError: 301 != 302 : Response didn't redirect as expected: Response

code was 301 (expected 302)

**Beware of Greedy Regular Expressions!**

That’s a little strange. We haven’t actually specified a URL for */lists/1/add\_item* yet, so our expected failure is 404 != 302. Why are we getting a 301?

This was a bit of a puzzler! It’s because we’ve used a very "greedy" regular expression in our URL:

superlists/urls.py

url(r'^lists/(.+)/$', views.view\_list, name='view\_list'),

Django has some built-in code to issue a permanent redirect (301) whenever someone asks for a URL which is *almost* right, except for a missing slash. In this case, */lists/1/add\_item/* would be a match for lists/(.+)/, with the (.+) capturing 1/add\_item. So Django "helpfully" guesses that we actually wanted the URL with a trailing slash.

We can fix that by making our URL pattern explicitly capture only numerical digits, by using the regular expression \d:

superlists/urls.py

url(r'^lists/(\d+)/$', views.view\_list, name='view\_list'),

That gives us the failure we expected:

AssertionError: 0 != 1

[...]

AssertionError: 404 != 302 : Response didn't redirect as expected: Response

code was 404 (expected 302)

**The Last New URL**

Now we’ve got our expected 404, let’s add a new URL for adding new items to existing lists:

superlists/urls.py

urlpatterns = [

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

url(r'^lists/new$', views.new\_list, name='new\_list'),

url(r'^lists/(\d+)/$', views.view\_list, name='view\_list'),

url(r'^lists/(\d+)/add\_item$', views.add\_item, name='add\_item'),

]

Three very similar-looking URLs there. Let’s make a note on our to-do list; they look like good candidates for a refactoring:

* *Adjust model so that items are associated with different lists*
* *Add unique URLs for each list*
* *Add a URL for creating a new list via POST*
* *Add URLs for adding a new item to an existing list via POST*
* *Refactor away some duplication in urls.py*

Back to the tests, we get the usual missing module view objects:

AttributeError: module 'lists.views' has no attribute 'add\_item'

**The Last New View**

Let’s try:

lists/views.py

def add\_item(request):

pass

Aha:

TypeError: add\_item() takes 1 positional argument but 2 were given

lists/views.py

def add\_item(request, list\_id):

pass

And then:

ValueError: The view lists.views.add\_item didn't return an HttpResponse object.

It returned None instead.

We can copy the redirect from new\_list and the List.objects.get from view\_list:

lists/views.py

def add\_item(request, list\_id):

list\_ = List.objects.get(id=list\_id)

return redirect(f'/lists/{list\_.id}/')

That takes us to:

self.assertEqual(Item.objects.count(), 1)

AssertionError: 0 != 1

Finally we make it save our new list item:

lists/views.py

def add\_item(request, list\_id):

list\_ = List.objects.get(id=list\_id)

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

return redirect(f'/lists/{list\_.id}/')

And we’re back to passing tests.

Ran 8 tests in 0.050s

OK

**Testing the Response Context Objects Directly**

We’ve got our new view and URL for adding items to existing lists; now we just need to actually use it in our *list.html* template. So we open it up to adjust the form tag…​

lists/templates/list.html

<form method="POST" action="but what should we put here?">

…​oh. To get the URL for adding to the current list, the template needs to know what list it’s rendering, as well as what the items are. We want to be able to do something like this:

lists/templates/list.html

<form method="POST" action="/lists/{{ list.id }}/add\_item">

For that to work, the view will have to pass the list to the template. Let’s create a new unit test in ListViewTest:

lists/tests.py (ch07l041)

def test\_passes\_correct\_list\_to\_template(self):

other\_list = List.objects.create()

correct\_list = List.objects.create()

response = self.client.get(f'/lists/{correct\_list.id}/')

self.assertEqual(response.context['list'], correct\_list)

|  |  |
| --- | --- |
|  | response.context represents the context we’re going to pass into the render function—​the Django Test Client puts it on the response object for us, to help with testing. |

That gives us:

KeyError: 'list'

because we’re not passing list into the template. It actually gives us an opportunity to simplify a little:

lists/views.py

def view\_list(request, list\_id):

list\_ = List.objects.get(id=list\_id)

return render(request, 'list.html', {'list': list\_})

That, of course, will break one of our old tests, because the template needed items:

FAIL: test\_displays\_only\_items\_for\_that\_list (lists.tests.ListViewTest)

[...]

AssertionError: False is not true : Couldn't find 'itemey 1' in response

But we can fix it in *list.html*, as well as adjusting the form’s POST action:

lists/templates/list.html (ch07l043)

<form method="POST" action="/lists/{{ list.id }}/add\_item">

[...]

{% for item in list.item\_set.all %}

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

{% endfor %}

|  |  |
| --- | --- |
|  | There’s our new form action. |
|  | .item\_set is called a [reverse lookup](https://docs.djangoproject.com/en/1.11/topics/db/queries/#following-relationships-backward). It’s one of Django’s incredibly useful bits of ORM that lets you look up an object’s related items from a different table…​ |

So that gets the unit tests to pass:

Ran 9 tests in 0.040s

OK

How about the FTs?

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

[...]

..

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

Ran 2 tests in 9.771s

OK

HOORAY! Oh, and a quick check on our to-do list:

* *Adjust model so that items are associated with different lists*
* *Add unique URLs for each list*
* *Add a URL for creating a new list via POST*
* *Add URLs for adding a new item to an existing list via POST*
* *Refactor away some duplication in urls.py*

Irritatingly, the Testing Goat is a stickler for tying up loose ends too, so we’ve got to do this one final thing.

Before we start, we’ll do a commit—​always make sure you’ve got a commit of a working state before embarking on a refactor:

$ **git diff**

$ **git commit -am "new URL + view for adding to existing lists. FT passes :-)"**

**A Final Refactor Using URL includes**

*superlists/urls.py* is really meant for URLs that apply to your entire site. For URLs that only apply to the lists app, Django encourages us to use a separate *lists/urls.py*, to make the app more self-contained. The simplest way to make one is to use a copy of the existing *urls.py*:

$ **cp superlists/urls.py lists/**

Then we replace three lines in *superlists/urls.py* with an include:

superlists/urls.py

from django.conf.urls import include, url

from lists import views as list\_views

from lists import urls as list\_urls

urlpatterns = [

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

url(r'^lists/', include(list\_urls)),

]

|  |  |
| --- | --- |
|  | While we’re at it, we use the import x as y syntax to alias views and urls. This is good practice in your top-level *urls.py*, because it will let us import views and urls from multiple apps if we want—​and indeed we will need to later on in the book. |
|  | Here’s the include. Notice that it can take a part of a URL regex as a prefix, which will be applied to all the included URLs (this is the bit where we reduce duplication, as well as giving our code a better structure). |

Back in *lists/urls.py* we can trim down to only include the latter part of our three URLs, and none of the other stuff from the parent *urls.py*:

lists/urls.py (ch07l046)

from django.conf.urls import url

from lists import views

urlpatterns = [

url(r'^new$', views.new\_list, name='new\_list'),

url(r'^(\d+)/$', views.view\_list, name='view\_list'),

url(r'^(\d+)/add\_item$', views.add\_item, name='add\_item'),

]

Rerun the unit tests to check that everything worked.

When I did it, I couldn’t quite believe I did it correctly on the first go. It always pays to be skeptical of your own abilities, so I deliberately changed one of the URLs slightly, just to check if it broke a test. It did. We’re covered.

Feel free to try it yourself! Remember to change it back, check that the tests all pass again, and then do a final commit:

$ **git status**

$ **git add lists/urls.py**

$ **git add superlists/urls.py**

$ **git diff --staged**

$ **git commit**

Phew. A marathon chapter. But we covered a number of important topics, starting with test isolation, and then some thinking about design. We covered some rules of thumb like "YAGNI" and "three strikes then refactor". But, most importantly, we saw how to adapt an existing site step by step, going from working state to working state, in order to iterate towards a new design.

I’d say we’re pretty close to being able to ship this site, as the very first beta of the superlists website that’s going to take over the world. Maybe it needs a little prettification first…​let’s look at what we need to do to deploy it in the next couple of chapters.

Some More TDD Philosophy

Working State to Working State (aka The Testing Goat vs. Refactoring Cat)

Our natural urge is often to dive in and fix everything at once…​but if we’re not careful, we’ll end up like Refactoring Cat, in a situation with loads of changes to our code and nothing working. The Testing Goat encourages us to take one step at a time, and go from working state to working state.

Split work out into small, achievable tasks

Sometimes this means starting with "boring" work rather than diving straight in with the fun stuff, but you’ll have to trust that YOLO-you in the parallel universe is probably having a bad time, having broken everything, and struggling to get the app working again.

YAGNI

You ain’t gonna need it! Avoid the temptation to write code that you think *might* be useful, just because it suggests itself at the time. Chances are, you won’t use it, or you won’t have anticipated your future requirements correctly. See [[chapter\_outside\_in]](http://www.obeythetestinggoat.com/book/chapter_outside_in.html) for one methodology that helps us avoid this trap.

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