Table of Contents

Working Incrementally

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 that takes you from working state to working state. Testing Goat, not Refactoring Cat!

Small Design When Necessary

Let's have a think about how we want support for multiple lists to work.

At the moment, the only URL for our site is the home page, and that's why there's only one global list. The most obvious way to support multiple lists is to say that each list gets its own URL, so that people can start multiple lists, or so that different people can have different lists. How might that work?

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 product* 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 [chapter_05_post_and_database], 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, we're going to have 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—something like Multiple users with multiple lists at multiple URLs.

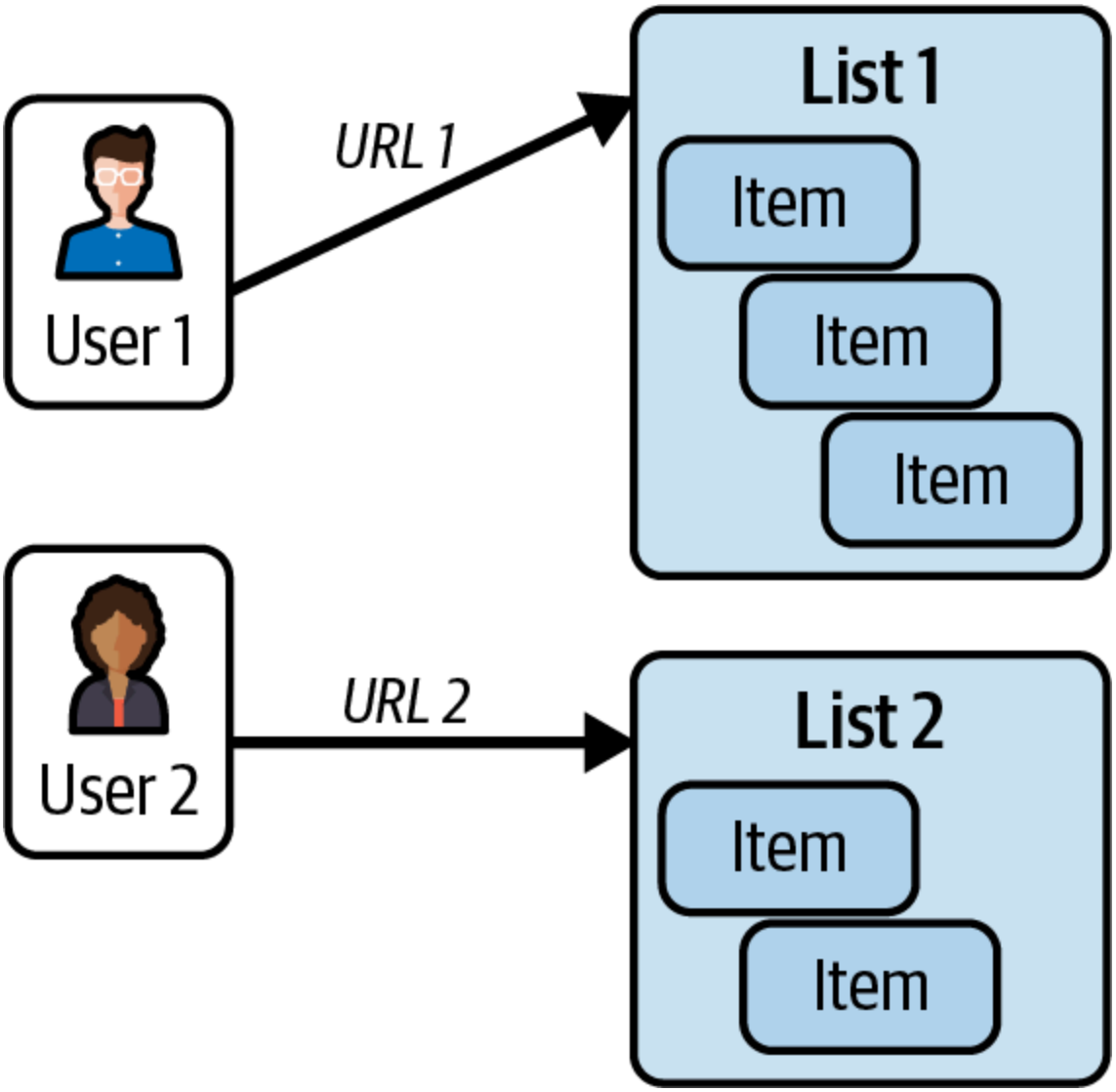

Figure 1. Multiple users with multiple lists at multiple URLs

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 should 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, 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 just end up with a load of unused code, adding to the complexity of your application.

YAGNI is the motto we use to resist our overenthusiastic creative urges. We avoid writing any code that's not strictly required.

Don't write any code unless you absolutely have to.[1]

REST-ish

We have an idea of the data structure we want—the "model" part of model-view-controller (we talked about MVC in [django-mvc]). What about the "view" and "controller" parts? How should the user interact with `List` s and their `Item` s 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 (take a look at Online Appendix: Building a REST API (https://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>/
```

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, duplicated in The TDD process with both functional and unit tests for your convenience.

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 FT to verify that we don't break what already works, and the new FT 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.

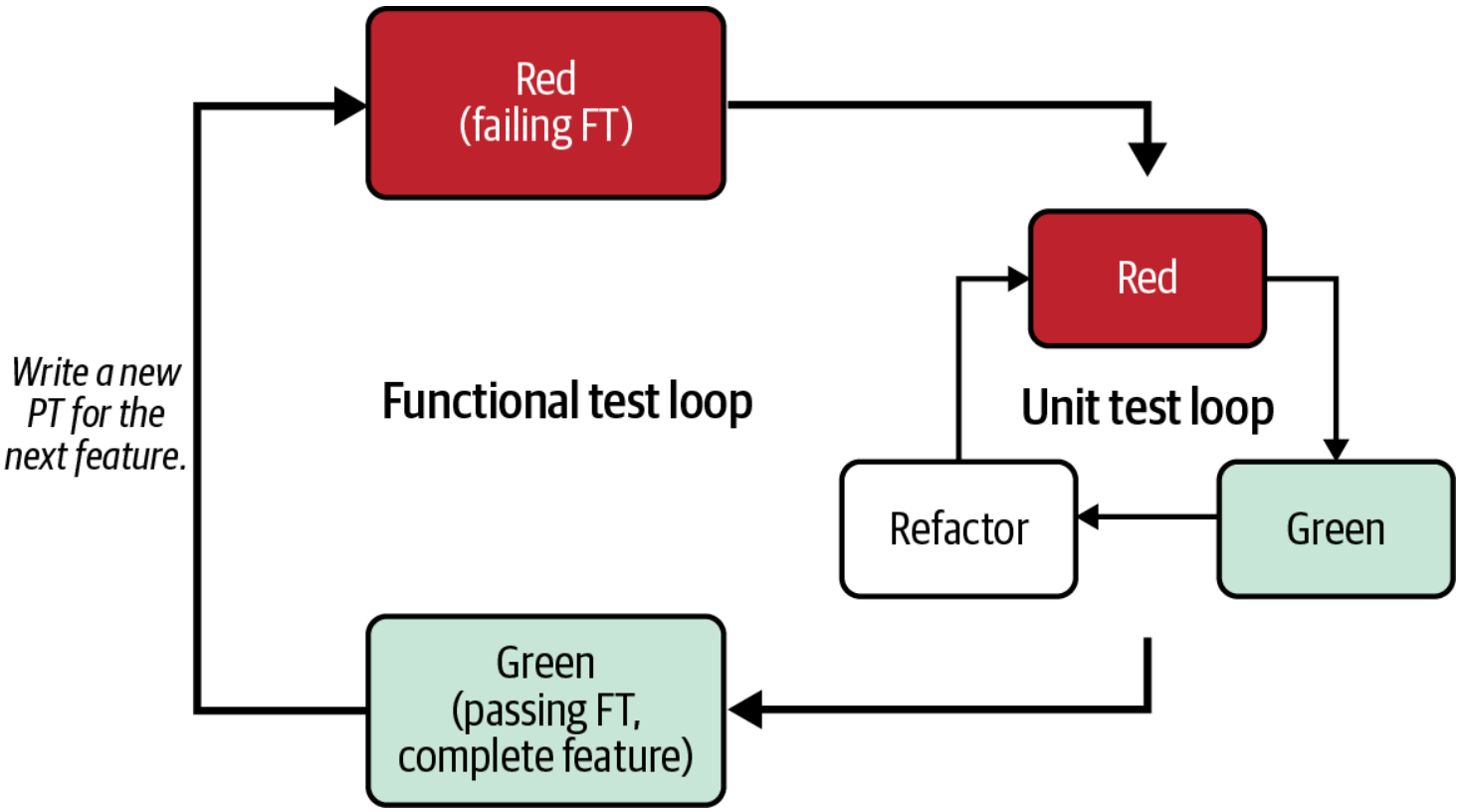

Figure 2. The TDD process with both functional and unit tests

Ensuring We Have a Regression Test

Our existing FT, `test_can_start_a_todo_list()`, is going to act as our regression test.

Let's translate our scratchpad into a new FT 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)

```python
def test_can_start_a_todo_list(self):
    # Edith has heard about a cool new online to-do app.
    [...]
    # 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/.+")   1
```

1 `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 (https://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)

```
PYTHON
    [...]
    self.assertRegex(edith_list_url, "/lists/.+")

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

    ## We delete all the browser's cookies
    ## as a way of simulating a brand new user session   1
    self.browser.delete_all_cookies()

    # 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)

    # 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
```

1 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 we're faffing about deleting cookies...

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.t
ests.NewVisitorTest.test_multiple_users_can_start_lists_at_different_urls)

 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/functional_tests/tests.py", line 77, 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 `Regex didn't match` error, is suggesting to 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*, let's find `test_redirects_after_POST` and change the expected redirect location:

lists/tests.py (ch07l007)

```
    def test_redirects_after_POST(self):
        response = self.client.post("/", data={"item_text": "A new list item"})
        self.assertRedirects(response, "/lists/the-only-list-in-the-world/")
```
PYTHON

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 zero items. Ultimately, we want to solve for *n* items, but solving for one 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/'
- /
+ /lists/the-only-list-in-the-world/
 : Response redirected to '/', expected '/lists/the-only-list-in-the-world/':
Expected '/' to equal '/lists/the-only-list-in-the-world/'.
```

We can go adjust our `home_page` view in *lists/views.py*:

lists/views.py (ch07l008)

```
    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})
```
PYTHON

Django's unit test runner picks up on the fact that this is not a real URL yet:

```
$ python manage.py test lists
[...]
AssertionError: 404 != 200 : Couldn't retrieve redirection page
'/lists/the-only-list-in-the-world/': response code was 404 (expected 200)
```

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

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

superlists/urls.py (ch07l009)

```
from django.urls import path
from lists import views

urlpatterns = [
    path("", views.home_page, name="home"),
    path("lists/the-only-list-in-the-world/", views.home_page, name="view_list"),
 1
]
```

PYTHON

1 We'll just point our new URL at the existing home page view. This is the minimal change.

> Watch out for trailing slashes in URLs, both here in *urls.py* and in the tests. They're a common source of confusion: Django will return a 301 redirect rather than a 404 if you try to access a URL that's missing its trailing slash.[2]

That gets our unit tests passing:

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

What do the FTs think?

```
$ python manage.py test functional_tests
[...]
AssertionError: 'Buy peacock feathers' unexpectedly found in 'Your To-Do
list\n1: Buy peacock feathers'
```

Good, they get a little further along. We now confirm that we have a new URL, but the actual page content is still the same; it shows the old list.

Separating Out Our Home Page and List View Functionality

We now have two URLs, but they're actually doing the exact same thing. Under the hood, they're just pointing at the same function. Continuing to work incrementally, we can start to break apart the responsibilities for these two different URLs:

- The home page only needs to display a static form, and support creating a brand new list based on its first item.
- The list view page needs to be able to display existing list items and add new items to the list.

Let's split out some tests for our new URL.

Open up *lists/tests.py*, and add a new test class called `ListViewTest`. Then:

1. Copy across the `test_renders_input_form()` test from `HomePageTest` into our new class.
2. Move the method called `test_displays_all_list_items()`.
3. In both, change just the URL that is invoked by `self.client.get()`.
4. We *won't* copy across the `test_uses_home_template()` yet, as we're not quite sure what template we want to use. We'll stick to the tests that check behaviour, rather than implementation.

lists/tests.py (ch07l010)

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


class ListViewTest(TestCase):
    def test_renders_input_form(self):
        response = self.client.get("/lists/the-only-list-in-the-world/")
        self.assertContains(response, '<form method="POST">')
        self.assertContains(response, '<input name="item_text"')

    def test_displays_all_list_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")
```

Let's try running these tests now:

```
$ python manage.py test lists
OK
```

It passes, because the URL is still pointing at the `home_page` view.

Let's make it point at a new view:

superlists/urls.py (ch07l011)

```python
from django.urls import path
from lists import views

urlpatterns = [
    path("", views.home_page, name="home"),
    path("lists/the-only-list-in-the-world/", views.view_list, name="view_list"),
]
```

That predictably fails because there is no such view function yet:

```
$ python manage.py test lists
[...]
    path("lists/the-only-list-in-the-world/", views.view_list,
name="view_list"),
                                               ^^^^^^^^^^^^^^^^
AttributeError: module 'lists.views' has no attribute 'view_list'
```

A new view function

Fair enough. Let's create a placeholder view function in *lists/views.py*:

lists/views.py (ch07l012-0)

```python
def view_list(request):
    pass
```

Not quite good enough:

```
ValueError: The view lists.views.view_list didn't return an HttpResponse
object. It returned None instead.
[...]
FAILED (errors=3)
```

Looking for the minimal code change, let's just make the view return our existing *home.html* template, but with nothing in it:

lists/views.py (ch07l012-1)

```
def view_list(request):
    return render(request, "home.html")
```
PYTHON

Now the tests guide us to making sure that our list view shows existing list items:

```
FAIL: test_displays_all_list_items
(lists.tests.ListViewTest.test_displays_all_list_items)
[...]
AssertionError: False is not true : Couldn't find 'itemey 1' in the following
response
```

So let's copy the last two lines from `home_page` more directly:

lists/views.py (ch07l012)

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

That gets us to passing unit tests!

```
Ran 8 tests in 0.035s

OK
```

The FTs Detect a Regression

As always when we get to passing unit tests, we run the FTs to check how things are doing "in real life":

```
$ python manage.py test functional_tests
[...]
FF
======================================================================
FAIL: test_can_start_a_todo_list
(functional_tests.tests.NewVisitorTest.test_can_start_a_todo_list)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/functional_tests/tests.py", line 62, in
test_can_start_a_todo_list
[...]
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 (functional_tests.t
ests.NewVisitorTest.test_multiple_users_can_start_lists_at_different_urls)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/functional_tests/tests.py", line 89, in
test_multiple_users_can_start_lists_at_different_urls
    self.assertNotIn("Buy peacock feathers", page_text)
AssertionError: 'Buy peacock feathers' unexpectedly found in 'Your To-Do
list\n1: Buy peacock feathers'
```

Another Race Condition Example

You may have noticed that the assertions around line 63 are in a slightly unexpected order:

functional_tests/tests.py

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

Try putting them the other way around, 1 then 2, and run the FTs a few times. There's a good chance you'll notice an inconsistency in the results. Sometimes you see:

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

And sometimes you'll see:

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

That's because of a race condition between the Selenium assertions in the FT, and the server returning our new page. Just before we tap Enter, the page is still showing `1: Buy peacock feathers`. Our next assertion is then checking for `1: Buy peacock feathers`, which is already on the page. But, at the same time, the server is busy returning a new page that also says `1: Use peacock feathers to make a fly`.

So, depending on who gets there first, the first assert may pass or fail, meaning that you may get an error on the first assert or on the second.

That's why I put the assertions "backwards", so we check for `2: Use peacock feathers` *first*, because it should *never* be present on the old page. This means that as soon as we detect it, we must be on the new page.

Subtle, right? Selenium tests are fiddly like that.

Not only is our new FT failing, but the old one is too. That tells us we've introduced a *regression*. But what?

Both tests are 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 62 in the first 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 that we *do* have are doing what they should. Let's have a quick look at those unit tests to see what they tell us:

```
$ grep -E "class|def" lists/tests.py
class HomePageTest(TestCase):
    def test_uses_home_template(self):
    def test_renders_input_form(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_renders_input_form(self):
    def test_displays_all_list_items(self):
class ItemModelTest(TestCase):
    def test_saving_and_retrieving_items(self):
```

The home page displays the right form and 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 FTs aren't, it's often pointing at a problem in code that's not covered by the unit tests—and in a Django app, that's often a template problem.

> Have you figured out what the problem is? Why not spend a moment trying to figure it out? Maybe open up the site in your browser, and see where the bug manifests. Perhaps open up the "view source" or browser DevTools and look at the underlying HTML?

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

lists/templates/home.html

```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.

Getting Back to a Working State as Quickly as Possible

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.

In other words, we've identified a new important part of the behaviour we want from our two views and their templates, which is the URL that the form points to. Let's add a check for that URL explicitly, in our two tests for each view (I'll use a diff to show the changes, hopefully that makes it nice and clear):

lists/tests.py (ch07l013-1)

```
DIFF
@@ -10,7 +10,7 @@ class HomePageTest(TestCase):

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

     def test_can_save_a_POST_request(self):
@@ -31,7 +31,7 @@ class HomePageTest(TestCase):
 class ListViewTest(TestCase):
     def test_renders_input_form(self):
         response = self.client.get("/lists/the-only-list-in-the-world/")
-        self.assertContains(response, '<form method="POST">')
+        self.assertContains(response, '<form method="POST" action="/">')
         self.assertContains(response, '<input name="item_text"')

     def test_displays_all_list_items(self):
```

That gives us two expected failures:

```
======================================================================
FAIL: test_renders_input_form
(lists.tests.HomePageTest.test_renders_input_form)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/lists/tests.py", line 13, in test_renders_input_form
    self.assertContains(response, '<form method="POST" action="/">')
    ~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: False is not true : Couldn't find '<form method="POST"
action="/">' in the following response
b'<html>\n  <head>\n    <title>To-Do lists</title>\n  </head>\n  <body>\n
<h1>Your To-Do list</h1>\n    <form method="POST">\n      <input
name="item_text" id="id_new_item" placeholder="Enter a to-do item" />\n
<input type="hidden" name="csrfmiddlewaretoken"
value=[...]
</form>\n    <table id="id_list_table">\n      \n    </table>\n
</body>\n</html>\n'

======================================================================
FAIL: test_renders_input_form
(lists.tests.ListViewTest.test_renders_input_form)
[...]
AssertionError: False is not true : Couldn't find '<form method="POST"
action="/">' in the following response
b'<html>\n  <head>\n    <title>To-Do lists</title>\n  </head>\n  <body>\n
[...]
```

And so we can fix it like this—the input form, for now, will always point at the home URL:

lists/templates/home.html (ch07l013-2)

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

Unit test pass:

```
OK
```

And we should see our FTs get back to a happier place:

```
FAIL: test_multiple_users_can_start_lists_at_different_urls (functional_tests.t
ests.NewVisitorTest.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 old FT (the one we're using as a regression 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, our unit tests pass and all our old FTs pass, so we've arrived at green. That means it's time to see if anything 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. Post requests are only handled by the home page though.

It feels like the responsibilities of our two views are a little tangled up. Let's try and disentangle them.

Another Small Step: A Separate Template for Viewing Lists

As 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.

We held off on copying across `test_uses_home_template()` until now, because we weren't quite sure what we wanted. Now let's add an explicit test to say that this view uses a different template:

lists/tests.py (ch07l014)

```
PYTHON
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_renders_input_form(self):
        [...]

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

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
```

Looks about right, let's change the view:

lists/views.py (ch07l015)

```
PYTHON
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
[...]
FAILED (errors=4)
```

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

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

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

```
$ python manage.py test lists
[...]
======================================================================
FAIL: test_displays_all_list_items
(lists.tests.ListViewTest.test_displays_all_list_items)
 ----------------------------------------------------------------------
[...]
AssertionError: False is not true : Couldn't find 'itemey 1' in the following
response
b''

======================================================================
FAIL: test_renders_input_form
(lists.tests.ListViewTest.test_renders_input_form)
----------------------------------------------------------------------
[...]
AssertionError: False is not true : Couldn't find '<form method="POST"
action="/">' in the following 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).

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

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".

I'll present the code change as a diff again, as I think that shows nice and clearly what we need to modify:

lists/templates/home.html (ch07l018)

```
                                                                    DIFF
    <body>
-     <h1>Your To-Do list</h1>
+     <h1>Start a new To-Do list</h1>
      <form method="POST" action="/">
        <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
        {% csrf_token %}
      </form>
-     <table id="id_list_table">
-       {% for item in items %}
-         <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
-       {% endfor %}
-     </table>
    </body>
```

We rerun the unit tests to check that hasn't broken anything...

```
OK
```

Good.

Now 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 and delete a few lines:

lists/views.py (ch07l019)

```
                                                                    DIFF
      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})
+     return render(request, "home.html")
```

Rerun the unit tests once more; they still pass:

```
OK
```

Time to run the FTs:

```
  File "...goat-book/functional_tests/tests.py", line 96, in
test_multiple_users_can_start_lists_at_different_urls
    self.wait_for_row_in_list_table("1: Buy milk")
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^
[...]
AssertionError: '1: Buy milk' not found in ['1: Buy peacock feathers', '2: Buy
milk']

----------------------------------------------------------------------
Ran 2 tests in 10.606s

FAILED (failures=1)
```

Great! Only one failure, so we know our regression test (the first FT) is passing. Let's see where we're getting to with the new FT.

Let's take a look at it again:

functional_tests/tests.py

```
    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")   1
        [...]

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

        # 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)   2

        # 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")    3
        [...]
```

PYTHON

1. Edith's list says "Buy peacock feathers".
2. When Francis loads the home page, there's no sign of Edith's list.
3. (This is the line where our test fails.) When Francis adds a new item, he sees Edith's item as number 1, and his appears as number 2.

Still, that's progress! The new FT *is* getting a little further along.

It may feel like we haven't made much headway because, 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 work 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,
           # changed the redirect in homepageTest & the home_page() view
           # added a new view view_list() in views.py,
           # and and added a line to urls.py.
$ git commit -a # add a message summarising the above, maybe something like
                # "new URL, view and template to display lists"
```

If this is all feeling a little abstract, now might be a good time to load up the site with `manage.py runserver` and try adding a couple of different lists yourself, and get a feel for how the site is currently behaving.

A Third Small Step: A New 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 at */lists/new*: 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 called `NewListTest`. Then, change the URL they POST to:

lists/tests.py (ch07l020)

PYTHON

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


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.get()
        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.assertRedirects(response, "/lists/the-only-list-in-the-world/")


class ListViewTest(TestCase):
    def test_uses_list_template(self):
        [...]
```

This is another place to pay attention to trailing slashes, incidentally. It's `/lists/new`, with no trailing slash. The convention I'm using is that URLs without a trailing slash are "action" URLs, which modify the database.[3]

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 (ch07l021)

```
urlpatterns = [
    path("", views.home_page, name="home"),
    path("lists/new", views.new_list, name="new_list"),
    path("lists/the-only-list-in-the-world/", views.view_list, name="view_list"),
]
```

PYTHON

Next we get a `no attribute 'new_list'`, so let's fix that, in *lists/views.py*:

lists/views.py (ch07l022)

```
def new_list(request):
    pass
```

PYTHON

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 because we know we'll need a redirect, let's borrow a line from `home_page`:

lists/views.py (ch07l023)

```
def new_list(request):
    return redirect("/lists/the-only-list-in-the-world/")
```

PYTHON

That gives:

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

Seems reasonably straightforward. We borrow another line from `home_page`:

lists/views.py (ch07l024)

```
def new_list(request):
    Item.objects.create(text=request.POST["item_text"])
    return redirect("/lists/the-only-list-in-the-world/")
```

PYTHON

And everything now passes:

```
Ran 9 tests in 0.030s

OK
```

And we can run the FTs to check that we're still in the same place:

```
[...]
AssertionError: '1: Buy milk' not found in ['1: Buy peacock feathers', '2: Buy
milk']
Ran 2 tests in 8.972s
FAILED (failures=1)
```

Our regression test passes, and the new FT gets to the same point.

Removing Now-Redundant Code and Tests

We're looking good. As 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 (ch07l025)

```
def home_page(request):
    return render(request, "home.html")
```

PYTHON

Yep! The unit tests pass:

```
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 8 tests in 0.016s
OK
```

And the FTs?

A Regression! Pointing Our Forms at the New URL

Oops. When we run the FTs:

```
======================================================================
ERROR: test_can_start_a_todo_list
(functional_tests.tests.NewVisitorTest.test_can_start_a_todo_list)
 ----------------------------------------------------------------------
[...]
  File "...goat-book/functional_tests/tests.py", line 52, in
test_can_start_a_todo_list
[...]
    self.wait_for_row_in_list_table("1: Buy peacock feathers")
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^
[...]
    table = self.browser.find_element(By.ID, "id_list_table")
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_list_table"]; For documentation [...]


======================================================================
ERROR: test_multiple_users_can_start_lists_at_different_urls (functional_tests.
tests.NewVisitorTest.test_multiple_users_can_start_lists_at_different_urls)
 ----------------------------------------------------------------------
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_list_table"]; For documentation [...]
[...]

Ran 2 tests in 11.592s
FAILED (errors=2)
```

Once again, the FTs pick up a tricky little bug, something that our unit tests alone would find it hard to catch.

Debugging in DevTools

This is another good time to spin up the dev server, and have a look around with a browser.

Let's also open up DevTools (https://firefox-source-docs.mozilla.org/devtools-user),[4] and click around to see what's going on:

- First I tried submitting a new list item, and saw we get sent back to the home page.
- Then I did the same with the browser DevTools open, and in the "network" tab I saw a POST request to "/". See DevTools shows a POST request to /.
- Finally, I had a look at the HTML source of the home page, and saw that the main form is still pointing at "/".

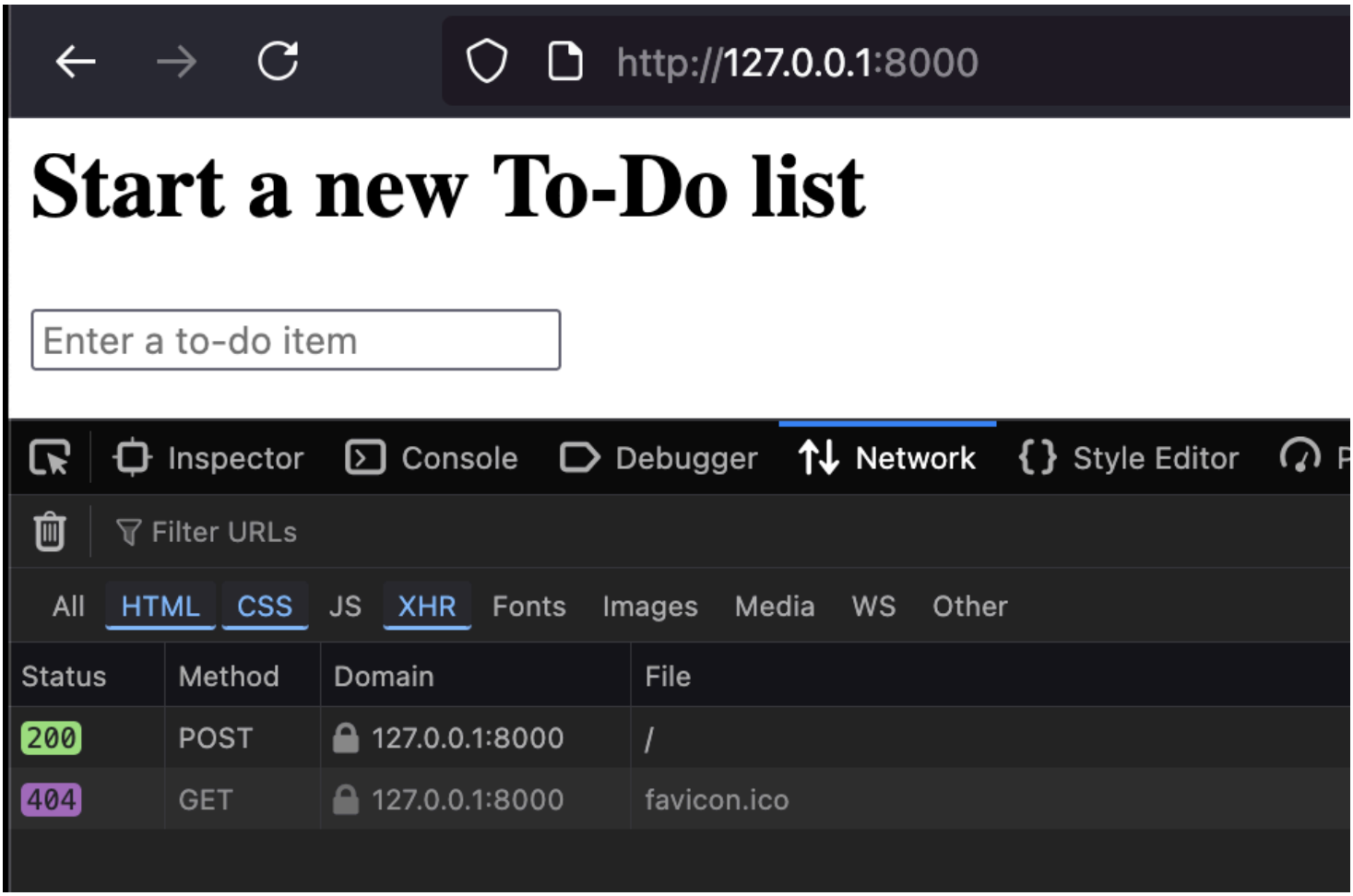

Figure 3. DevTools shows a POST request to /

Actually, *both* our forms are still pointing to the old URL. We have tests for this! Let's amend them:

lists/tests.py (ch07l026)

```
@@ -10,7 +10,7 @@ class HomePageTest(TestCase):

     def test_renders_input_form(self):
         response = self.client.get("/")
-        self.assertContains(response, '<form method="POST" action="/">')
+        self.assertContains(response, '<form method="POST" action="/lists/new">')
         self.assertContains(response, '<input name="item_text"')


@@ -33,7 +33,7 @@ class ListViewTest(TestCase):

     def test_renders_input_form(self):
         response = self.client.get("/lists/the-only-list-in-the-world/")
-        self.assertContains(response, '<form method="POST" action="/">')
+        self.assertContains(response, '<form method="POST" action="/lists/new">')
         self.assertContains(response, '<input name="item_text"')
```
DIFF

That gets us two failures:

```
AssertionError: False is not true : Couldn't find '<form method="POST"
action="/lists/new">' in the following response
[...]
AssertionError: False is not true : Couldn't find '<form method="POST"
action="/lists/new">' in the following response
[...]
```

In *both home.html* and *list.html*, let's change them:

lists/templates/home.html (ch07l027)

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

And:

lists/templates/list.html (ch07l028)

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

And that should get us back to working again:

```
Ran 8 tests in 0.006s

OK
```

And our FTs are still in the familiar "Francis sees Edith's items" place:

```
AssertionError: '1: Buy milk' not found in ['1: Buy peacock feathers', '2: Buy
milk']
[...]
FAILED (failures=1)
```

Perhaps this all seems quite pernickety, but 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 with three very short view functions, and we're sure the application is still working as well as it was before. We're getting good at this working-state-to-working-state malarkey!

```
$ git status # 5 changed files
$ git diff # URLs for forms x2, new test + view with code moves, and 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.

lists/tests.py (ch07l029)

```
DIFF
@@ -1,5 +1,5 @@
 from django.test import TestCase
-from lists.models import Item
+from lists.models import Item, List


 class HomePageTest(TestCase):
@@ -35,20 +35,30 @@ class ListViewTest(TestCase):
         self.assertContains(response, "itemey 2")


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

         second_item = Item()
         second_item.text = "Item the second"
+        second_item.list = mylist
         second_item.save()

+        saved_list = List.objects.get()
+        self.assertEqual(saved_list, mylist)
+
         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, mylist)
         self.assertEqual(second_saved_item.text, "Item the second")
+        self.assertEqual(second_saved_item.list, mylist)
```

Once again, this is a very verbose test, because I'm using it more as a demonstration of how the ORM works. We'll shorten it later,[5] but for now, let's work through and see how things work.

We create a new `List` object and then we assign each item to it by setting 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 `mylist`)—behind the scenes, these will compare themselves by checking that their primary key (the `.id` attribute) is the same.

Time for another unit-test/code cycle.

For the first few 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 [django_ORM_first_model].

Your first error should be:

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

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, mylist)
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 (ch07l033)

```
from django.db import models


class List(models.Model):
    pass


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

PYTHON

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 (1)' != <List: List object (1)>
```

We're not quite there. Look closely at each side of the `!=`. Do you see the quotes (`'`)? 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 (ch07l035)

```python
class Item(models.Model):
    text = models.TextField(default="")
    list = models.ForeignKey(List, default=None, on_delete=models.CASCADE)
```

That'll need a migration too. As 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. Now and again it's nice to do it to keep things tidy, 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 won't be able 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 Git.

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_list_items
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
[...]
ERROR: test_redirects_after_POST
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
[...]
ERROR: test_can_save_a_POST_request
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id

Ran 8 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 cause is the new relationship we've introduced between `Item`s and `List`s, which requires each item to have a parent list, and 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 (ch07l038)

```
class ListViewTest(TestCase):
    [...]
    def test_displays_all_list_items(self):
        mylist = List.objects.create()
        Item.objects.create(text="itemey 1", list=mylist)
        Item.objects.create(text="itemey 2", list=mylist)
```

PYTHON

That gets us down to two failing tests, both on tests that try to POST to our `new_list` view. Decode 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 "...goat-book/lists/tests.py", line 25, in test_redirects_after_POST
    response = self.client.post("/lists/new", data={"item_text": "A new list
item"})
[...]
  File "...goat-book/lists/views.py", line 11, in new_list
    Item.objects.create(text=request.POST["item_text"])
    ~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
[...]
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
```

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 (ch07l039)

```
PYTHON
from lists.models import Item, List
[...]

def new_list(request):
    nulist = List.objects.create()
    Item.objects.create(text=request.POST["item_text"], list=nulist)
    return redirect("/lists/the-only-list-in-the-world/")
```

And that gets our tests passing again:[6]

```
Ran 8 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 big 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

We can get rid of the silly `the-only-list-in-the-world` URL, but what shall we use as the unique identifier for our lists? Probably the simplest thing, for now, is just to use the autogenerated `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_list_items` test and call it `test_displays_only_items_for_that_list` instead, making it check that only the items for a specific list are displayed:

lists/tests.py (ch07l040)

```python
class ListViewTest(TestCase):
    def test_uses_list_template(self):
        mylist = List.objects.create()
        response = self.client.get(f"/lists/{mylist.id}/")   1
        self.assertTemplateUsed(response, "list.html")

    def test_renders_input_form(self):
        mylist = List.objects.create()
        response = self.client.get(f"/lists/{mylist.id}/")   1
        self.assertContains(response, '<form method="POST" action="/lists/new">')
        self.assertContains(response, '<input name="item_text"')

    def test_displays_only_items_for_that_list(self):
        correct_list = List.objects.create()   2
        Item.objects.create(text="itemey 1", list=correct_list)
        Item.objects.create(text="itemey 2", list=correct_list)
        other_list = List.objects.create()   2
        Item.objects.create(text="other list item", list=other_list)

        response = self.client.get(f"/lists/{correct_list.id}/")   3

        self.assertContains(response, "itemey 1")
        self.assertContains(response, "itemey 2")
        self.assertNotContains(response, "other list item")   4
```

1 Here's where we incorporate the ID of our new list into the GET URL.

2 In the "Given" phase of the test, we now set up two lists: the one we're interested in and an extraneous one.

3 We change this URL too, to point at the *correct* list.

4 And now, our "Then" section can check that the irrelevant list's items are definitely not present.

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

```
FAIL: test_displays_only_items_for_that_list
AssertionError: 404 != 200 : Couldn't retrieve content: Response code was 404
(expected 200)
[...]
FAIL: test_renders_input_form
AssertionError: 404 != 200 : Couldn't retrieve content: Response code was 404
(expected 200)
[...]
FAIL: test_uses_list_template
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 (ch07l041-0)

PYTHON

```
urlpatterns = [
    path("", views.home_page, name="home"),
    path("lists/new", views.new_list, name="new_list"),
    path("lists/<int:list_id>/", views.view_list, name="view_list"),
]
```

We adjust the path string for our URL to include a *capture group*, `<int:list_id>`, which will match any numerical characters, up to the following `/`. The captured `id` will be 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 integer `1`.

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

```
ERROR: test_displays_only_items_for_that_list
[...]
TypeError: view_list() got an unexpected keyword argument 'list_id'
[...]
ERROR: test_renders_input_form
[...]
TypeError: view_list() got an unexpected keyword argument 'list_id'
[...]
ERROR: test_uses_list_template
[...]
TypeError: view_list() got an unexpected keyword argument 'list_id'
[...]
FAIL: test_redirects_after_POST
[...]
AssertionError: 404 != 200 : Couldn't retrieve redirection page
'/lists/the-only-list-in-the-world/': response code was 404 (expected 200)
[...]
FAILED (failures=1, errors=3)
```

We can fix that easily with an unused parameter in *views.py*:

lists/views.py (ch07l041)

```
def view_list(request, list_id):
    [...]
```
PYTHON

That takes us down to our expected failure, plus something to do with an */only-list-in-the-world/* that's still hanging around somewhere, which I'm sure we can fix later.

```
FAIL: test_displays_only_items_for_that_list
[...]
AssertionError: 1 != 0 : 'other list item' unexpectedly found in the following
response
[...]
FAIL: test_redirects_after_POST
AssertionError: 404 != 200 : Couldn't retrieve redirection page
'/lists/the-only-list-in-the-world/': response code was 404 (expected 200)
```

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

lists/views.py (ch07l042)

```
def view_list(request, list_id):
    our_list = List.objects.get(id=list_id)
    items = Item.objects.filter(list=our_list)
    return render(request, "list.html", {"items": items})
```
PYTHON

Adjusting new_list to the New World

It's time to address the */only-list-in-the-world/* failure:

```
FAIL: test_redirects_after_POST
[...]
AssertionError: 404 != 200 : Couldn't retrieve redirection page
'/lists/the-only-list-in-the-world/': response code was 404 (expected 200)
```

Let's have a little look and find the test that'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/")
```
PYTHON

It looks like it hasn't been adjusted to the new world of `List` s and `Item` s. The test should be saying that this view redirects to the URL of the specific new list it just created.

lists/tests.py (ch07l043)

```
    def test_redirects_after_POST(self):
        response = self.client.post("/lists/new", data={"item_text": "A new list
item"})
        new_list = List.objects.get()
        self.assertRedirects(response, f"/lists/{new_list.id}/")
```
PYTHON

The test still fails, but we can now take a look at the view itself, and change it so it redirects to the right place:

lists/views.py (ch07l044)

```
def new_list(request):
    nulist = List.objects.create()
    Item.objects.create(text=request.POST["item_text"], list=nulist)
    return redirect(f"/lists/{nulist.id}/")
```

PYTHON

That gets us back to passing unit tests, phew!

```
$ python manage.py test lists
[...]
........
 ----------------------------------------------------------------------
Ran 8 tests in 0.033s

OK
```

What about the FTs?

The Functional Tests Detect Another Regression

It feels like we're done with migrating to the new URL structure; we must be almost there?

Well, almost. When we run the FTs, we get:

```
F.
======================================================================
FAIL: test_can_start_a_todo_list
(functional_tests.tests.NewVisitorTest.test_can_start_a_todo_list)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/functional_tests/tests.py", line 62, in
test_can_start_a_todo_list
    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* FT is actually passing: 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 FTs 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 URL 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 starting to get used to these now, so we know we'll need:

1. A new test for the new URL
2. A new entry in *urls.py*
3. A new view function

So, let's see if we can knock all that together quickly:

lists/tests.py (ch07l045)

```python
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.get()
        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 view. Yes, that would be a silly thing to do, and you can go too far down the road of testing for all the silly 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-silliness]. Oh, and yes it's an unused variable, and your IDE might nag you about it, but I find it helps me to remember what it's for.

So that fails as expected, the list item is not saved, and the new URL currently returns a 404:

```
AssertionError: 0 != 1
[...]
AssertionError: 404 != 302 : Response didn't redirect as expected: Response
code was 404 (expected 302)
```

The Last New urls.py Entry

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

superlists/urls.py (ch07l046)

```
urlpatterns = [
    path("", views.home_page, name="home"),
    path("lists/new", views.new_list, name="new_list"),
    path("lists/<int:list_id>/", views.view_list, name="view_list"),
    path("lists/<int:list_id>/add_item", views.add_item, name="add_item"),
]
```

PYTHON

We've got 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*.

The Last New View

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

```
AttributeError: module 'lists.views' has no attribute 'add_item'
```

Let's try:

lists/views.py (ch07l047)

```
def add_item(request):
    pass
```

PYTHON

Aha:

```
TypeError: add_item() got an unexpected keyword argument 'list_id'
```

lists/views.py (ch07l048)

```
def add_item(request, list_id):
    pass
```

PYTHON

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 (ch07l049)

```
def add_item(request, list_id):
    our_list = List.objects.get(id=list_id)
    return redirect(f"/lists/{our_list.id}/")
```

PYTHON

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 (ch07l050)

```
def add_item(request, list_id):
    our_list = List.objects.get(id=list_id)
    Item.objects.create(text=request.POST["item_text"], list=our_list)
    return redirect(f"/lists/{our_list.id}/")
```

PYTHON

And we're back to passing tests:

```
Ran 10 tests in 0.050s

OK
```

Hooray! Did that feel like quite a nice, fluid, unit-test/code cycle?

Testing Template Context 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. We have a unit test for the form's action; let's amend it:

lists/tests.py (ch07l051)

```
class ListViewTest(TestCase):
    def test_uses_list_template(self):
        [...]

    def test_renders_input_form(self):
        mylist = List.objects.create()
        response = self.client.get(f"/lists/{mylist.id}/")
        self.assertContains(
            response,
            f'<form method="POST" action="/lists/{mylist.id}/add_item">',
        )
        self.assertContains(response, '<input name="item_text"')

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

PYTHON

That fails as expected:

```
AssertionError: False is not true : Couldn't find '<form method="POST"
action="/lists/1/add_item">' in the following response
[...]
```

So, we open it up to adjust the form tag...

lists/templates/list.html

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

HTML

...oh.

To get the URL to add to the current list, the template needs to know what list it's rendering, as well as what the items are.

Well, "programming by wishful thinking",[7] let's just pretend we had access to everything we need, like a `list` variable in the template:

lists/templates/list.html (ch07l052)

```
      <form method="POST" action="/lists/{{ list.id }}/add_item">
```
HTML

That changes our error slightly:

```
AssertionError: False is not true : Couldn't find '<form method="POST"
action="/lists/1/add_item">' in the following response
b'<html>\n  <head>\n    <title>To-Do lists</title>\n  </head>\n  <body>\n
<h1>Your To-Do list</h1>\n    <form method="POST" action="/lists//add_item">\n   1
```

1 Do you see it says `/lists//add_item`? It's because Django templates will just silently ignore any undefined variables, and substitute empty strings for them.

Let's see if we can make our wish come true and pass our list to the template then:

lists/views.py (ch07l053)

```
def view_list(request, list_id):
    our_list = List.objects.get(id=list_id)
    items = Item.objects.filter(list=our_list)
    return render(request, "list.html", {"items": items, "list": our_list})
```
PYTHON

That gets us to passing tests:

```
OK
```

And we now have an opportunity to refactor, as passing both the list and its items together is redundant. Here's the change in the template:

lists/templates/list.html (ch07l054)

```
      {% for item in list.item_set.all %}   1
        <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
      {% endfor %}
```
HTML

1 `.item_set` is called a reverse lookup (https://docs.djangoproject.com/en/5.2/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.

The tests still pass...

```
OK
```

And we can now simplify the view down a little:

lists/views.py (ch07l055)

```
def view_list(request, list_id):
    our_list = List.objects.get(id=list_id)
    return render(request, "list.html", {"list": our_list})
```

PYTHON

And our unit tests still pass:

```
Ran 10 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 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 the three list-specific lines in *superlists/urls.py* with an `include()`:

superlists/urls.py (ch07l057)

```
PYTHON
from django.urls import include, path
from lists import views as list_views   1

urlpatterns = [
    path("", list_views.home_page, name="home"),
    path("lists/", include("lists.urls")),   2
]
```

1 While we're at it, we use the `import x as y` syntax to alias `views`. This is good practice in your top-level *urls.py*, because it will let us import views from multiple apps if we want—and indeed we will need to later on in the book.

2 Here's the `include`. Notice that it can take a part of a URL 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 (ch07l058)

```
PYTHON
from django.urls import path
from lists import views

urlpatterns = [
    path("new", views.new_list, name="new_list"),
    path("<int:list_id>/", views.view_list, name="view_list"),
    path("<int:list_id>/add_item", views.add_item, name="add_item"),
]
```

Rerun the unit tests to check that everything worked.

```
Ran 10 tests in 0.040s

OK
```

Can You Believe It?

When I saw this test pass, 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 (including the FTs), and then do a final commit:

```
$ git status
$ git add lists/urls.py
$ git add superlists/urls.py
$ git diff --staged
$ git commit
```

Phew. This was a marathon chapter. But we covered a number of important topics, starting with some thinking about design. We covered rules of thumb like "YAGNI" and "three strikes and refactor". But, most importantly, we saw how to adapt an existing codebase step by step, going from working state to working state, 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 versus 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_24_outside_in] for one methodology that helps us avoid this trap.

1. This is a much more widely applicable rule for programming in business, actually. If you can solve a problem without any coding at all, that's a big win.
2. The setting that controls this is called `APPEND_SLASH` (https://docs.djangoproject.com/en/5.2/ref/settings/#append-slash).
3. I don't think this is a very common convention anymore these days, but I quite like it. By all means, cast around for a URL naming scheme that makes sense to you in your own projects!
4. If you've not seen it before, DevTools is short for "developer tools". They're tools that Firefox (and other browsers) give you to be able to look "under the hood" and see what's going on with web pages, including the source code, what network requests are being made, and what JavaScript is doing. You can open up DevTools with Ctrl+Shift+I or Cmd-Opt-I.
5. In [rewrite-model-test], if you're curious.
6. Are you wondering about the strange spelling of the "nulist" variable? Other options are "list", which would shadow the built-in `list()` function, and `new_list`, which would shadow the name of the function that contains it. Or `list_` with the trailing underscore, which I find a bit ugly, or `list1` or `listey` or `mylist`, but none are particularly satisfactory.
7. TDD is a bit like programming by wishful thinking, in that, when we write the tests before the implementation, we express a wish: we wish we had some code that worked! The phrase "programming by wishful thinking" actually has a wider meaning, of writing your code in a top-down kind of way. We'll come back and talk about it more in [chapter_24_outside_in].

Comments

Emoji?

7 Responses

Upvote Funny Love Surprised Angry Sad

3 Comments Kgotso Koete ▾

Join the discussion…

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

Florian Bautry

2 years ago

In the comment of the second commit (just before the "third small step" section) the same thing is said twice, slightly differently :
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 added a line to urls.py
moved one test to a new class in lists/tests.py, <- duplicate start here
added a new view and simplified home_page in views.py,
and added a line to urls.py.

1 0 Reply

D

Dima

a year ago

Shouldn't we commit the changes only when all the tests are passing?

0 0 Reply

G

Gad

a year ago

Ough, that was quite a chapter !

0 0 Reply