วิศวกรรมซอฟต์แวร์

Software Engineering

สมเกียรติ วังศิริพิทักษ์

somkiat.wa@kmitl.ac.th

ห้อง 518 หรือ ห้อง 506 (MIV Lab)

PART II

Software Testing & TDD

Test-Driven Development

Test-Driven Development

Testing a Simple Home Page with Unit Tests

Our First Django App, and Our First Unit Test

- Django encourages you to structure your code into apps.
  - One project can have many apps.
- Let's start an app for our to-do lists:

```
$ python manage.py startapp lists
```

```
superlists/
├── db.sqlite3
├── funtional_tests_firefox.py
├── lists
│   ├── admin.py
│   ├── apps.py
│   ├── __init__.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── manage.py
└── superlists
    ├── __init__.py
    ├── __pycache__
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

Unit Tests, and How They Differ from Functional Tests

- The basic distinction is that …
  - **Functional tests** test the application from the outside, from the point of view of the *user*. (high level)
  - **Unit tests** test the application from the inside, from the point of view of the *programmer*. (low level)
- The application will be covered by both types of test.
- The workflow will look a bit like this.

Unit Tests, and How They Differ from Functional Tests

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

Unit Testing in Django

list/test.py

```
from django.test import TestCase

# Create your tests here.
```

Django suggested we use a special version of TestCase.
It's an augmented version of the standard unittest.TestCase, with some additional Django-specific features.

- First, let's see how the unit test will definitely be run by our automated test runner.
  - In the case of `functional_tests_firefox.py`, we're running it directly.
  - But this **file made by Django** is a bit more like magic.

```
superlists/
├── db.sqlite3
├── funtional_tests_firefox.py
├── lists
│   ├── admin.py
│   ├── apps.py
│   ├── __init__.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── manage.py
└── superlists
    ├── __init__.py
    ├── __pycache__
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

- Let's make a deliberately silly failing test:


```
from django.test import TestCase


class SmokeTest(TestCase):

    def test_bad_maths(self):
        self.assertEqual(1 + 1, 3)
```

- Let's invoke this mysterious Django test runner.

Run it from 'superlists' directory

```
$ python manage.py test
Creating test database for alias 'default'...
Found 1 test(s).
System check identified no issues (0 silenced).
F
======================================================================
FAIL: test_bad_maths (lists.tests.SmokeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\somkiat\src\SE\Lecture\superlists\lists\tests.py", line 6,
in test_bad_maths
    self.assertEqual(1 + 1, 3)
AssertionError: 2 != 3

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

Unit Testing in Django

- This is a good point for a **commit**.

Run it from 'superlists' directory

```
$ git status # should show you lists is untracked
$ git add lists
$ git diff --staged # will show you the diff that you're about to commit
$ git commit -m "Add app for lists, with deliberately failing unit test"
```

The **-m flag** lets you **pass** in a **commit message** at the command line,
so you don't need to use an editor.
The key rule is: **make sure** you always **review** what you're about to commit **before you do** it.

Django's MVC, URLs, and View Functions

- **Django** is structured along a classic Model-View-Controller (MVC) pattern.
- As with any web server, **Django's** main job is to decide what to do *when a user asks for a particular URL* on our site.

- Django's workflow goes something like this:

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

Django's MVC, URLs, and View Functions

- We want to test **two** things:
  - Can we **resolve** the URL for the root of the site ("/") to a particular view function we've made?
  - Can we make this view function **return** some HTML which will get the functional test to pass?

Open up lists/**tests.py**, and
change our silly test to something like this:

```
from django.urls import resolve
from django.test import TestCase
from lists.views import home_page

class HomePageTest(TestCase):

    def test_root_url_resolves_to_home_page_view(self):
        found = resolve('/')
        self.assertEqual(found.func, home_page)
```

We're **checking that** resolve, when called with "**/**", the root of the site, **finds a function** called **home_page**.

resolve is the function Django uses internally to **resolve URLs** and **find** what **view function** they should **map to**.

It's the **view function** we're **going to write next**, which will actually **return the HTML** we want.

You can see from the import that **we're planning to store it in `lists/views.py`**.

- When we run the tests …

```
$ python manage.py test
ImportError: cannot import name 'home_page'
```

View Functions

```
superlists/
├── db.sqlite3
├── functional_tests.py
├── lists
│   ├── admin.py
│   ├── apps.py
│   ├── __init__.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── manage.py
└── superlists
    ├── __init__.py
    ├── __pycache__
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

- Since we're learning and only just starting out the TDD …
  - We only allow ourselves to change (or add) one line of code at a time—and each time, we make just the minimal change required to address the current test failure.
- Our current test failure: Cannot import `home_page` from `lists.views`.
  - Let's fix that—and only that.

list/view.py

```
from django.shortcuts import render

# Create your views here.
home_page = None
```

- Then run the test again

Reading Tracebacks

```
$ python manage.py test
Creating test database for alias 'default'...
Found 1 test(s).
System check identified no issues (0 silenced).
E
======================================================================
ERROR: test_root_url_resolves_to_home_page_view (lists.tests.HomePageTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\somkiat\src\SE\Lecture\superlists\lists\tests.py", line 8, in
test_root_url_resolves_to_home_page_view
    found = resolve('/')
  File "C:\Users\w_som\.virtualenvs\superlists\lib\site-
packages\django\urls\base.py", line 24, in resolve
    return get_resolver(urlconf).resolve(path)
  File "C:\Users\w_som\.virtualenvs\superlists\lib\site-
packages\django\urls\resolvers.py", line 725, in resolve
    raise Resolver404({"tried": tried, "path": new_path})
django.urls.exceptions.Resolver404: {'tried': [[<URLResolver <URLPattern list>
(admin:admin) 'admin/'>]], 'path': ''}

----------------------------------------------------------------------
Ran 1 test in 0.004s

FAILED (errors=1)
Destroying test database for alias 'default'...
```

The **next** thing to double-check is:
which test is failing?
Is it definitely the one we expected—
that is, the one we just wrote?
In this case, the answer is **yes**.

The **first place** you look is usually the error itself.
It will let you identify the problem immediately.
But sometimes, like in this case, it's not quite self-evident.

Reading Tracebacks

Then we look for the place in our test code that kicked off the failure.
From the top of the traceback, looking for the filename of the tests file,
to check which test function, and what line of code, the failure is coming from.
In this case it's *the line where we call the resolve function for the "/" URL*.

The traceback is telling us that, **when trying to resolve "/"**, Django raised a **404 error**—in other words,
Django can't find a URL mapping for "/". Let's help it out.

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

superlists/urls.py

```
"""
URL configuration for superlists project.

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/5.0/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""
from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

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

Get rid of the admin URL, because we won't be using the Django admin site for now.

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

superlists/urls.py

```
"""
URL configuration for superlists project.

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/5.0/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""
from django.urls import path
from lists import views

urlpatterns = [
    path('', views.home_page, name='home'),
]
```

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

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

That's progress! No longer getting a 404.

The message is telling us what's going on.
The unit tests have actually **made** the link between the URL "/" and the `home_page = None` in `lists/views.py`, and are now complaining that the `home_page` view is not callable.
That gives us a justification for changing it from being None to being an actual function.
Every single code change is driven by the tests!

list/view.py

```
from django.shortcuts import render

# Create your views here.
def home_page():
    pass
```

```
$ python manage.py test
Creating test database for alias 'default'...
Found 1 test(s).
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
Destroying test database for alias 'default'...

$ git diff # should show changes to urls.py, tests.py, and views.py
$ git commit -am "First unit test and url mapping, dummy view"
```

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

The **a** and **m** flags together
add all changes to tracked files and
uses the **commit message** from the command line.

Unit Testing a View

- Now we will be writing a **test** for our **view**, so that it can be something more than a do-nothing function, and instead be a function that returns a real response with HTML to the browser.
  - Open up `lists/tests.py`, and add a new test method.

lists/tests.py

```
from django.urls import resolve
from django.test import TestCase
from django.http import HttpRequest

from lists.views import home_page

class HomePageTest(TestCase):

    def test_root_url_resolves_to_home_page_view(self):
        found = resolve('/')
        self.assertEqual(found.func, home_page)

    def test_home_page_returns_correct_html(self):
        request = HttpRequest()
        response = home_page(request)
        html = response.content.decode('utf8')
        self.assertTrue(html.startswith('<html>'));
        self.assertIn('<title>To-Do lists</title>', html)
        self.assertTrue(html.endswith('</html>'));
```

Create an **HttpRequest object**, which is what Django will see when a user's browser asks for a page.

Pass it **to** our **`home_page` view**, which gives us a response.
This object is an instance of a class called HttpResponse

Then, **extract** the `.content` of the **response**.
These are the raw bytes, the ones and zeros that would be sent down the wire to the user's browser.
We call `.decode()` to **convert** them **into** the **string** of HTML that's being sent to the user.

We want it to start with an `<html>` tag which gets closed at the end.
And we want a `<title>` tag somewhere in the middle, with the words "To-Do lists" in it—because that's what we specified in our functional test.

Let's run the unit tests.

```
TypeError: home_page() takes 0 positional arguments but 1 was given
```

The Unit-Test/Code Cycle

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

- And repeat!
  - The more nervous we are about getting our code right, the smaller and more minimal we make each code change.
    (The idea is to be absolutely sure that each bit of code is justified by a test.)
- This may seem laborious. But once you get into the swing of things, you'll find yourself coding quickly even if you take microscopic steps.
- **Let's see how fast** we can get this **cycle** going:

The Unit-Test/Code Cycle

- Minimal code change:

lists/views.py
```
def home_page(request):
    pass
```

- Tests:

```
$ python manage.py test
[...]
html = response.content.decode('utf8')
AttributeError: 'NoneType' object has no attribute 'content'
[...]
```

- Code—we use `django.http.HttpResponse`, as predicted:

lists/views.py
```
from django.http import HttpResponse

# Create your views here.
def home_page(request):
    return HttpResponse()
```

- Tests again:

```
[...]
self.assertTrue(html.startswith('<html>'))
AssertionError: False is not true
[...]
```

- Code again:

lists/views.py
```
def home_page(request):
    return HttpResponse('<html>')
```

- Tests:

```
$ python manage.py test
[...]
AssertionError: '<title>To-Do lists</title>' not found in '<html>'
[...]
```

- Code:

lists/views.py
```
def home_page(request):
    return HttpResponse('<html><title>To-Do lists</title>')
```


- Tests—almost there?

```
[...]
 self.assertTrue(html.endswith('</html>'))
AssertionError: False is not true
[...]
```

- Come on, one last effort:

lists/views.py
```
def home_page(request):
    return HttpResponse('<html><title>To-Do lists</title></html>')
```

- Surely?

```
$ python manage.py test
Creating test database for alias 'default'...
Found 2 test(s).
System check identified no issues (0 silenced).
..
----------------------------------------------------------------------
Ran 2 tests in 0.002s

OK
Destroying test database for alias 'default'...
```

Yes!
Now, **let's run our functional tests.** Don't forget to **spin up the dev server** again (if not still running).

- Run functional tests.

```
$ python python functional_tests_firefox.py
======================================================================
FAIL: test_can_start_a_list_and_retrieve_it_later
(__main__.NewVisitorTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File
"D:\somkiat\src\SE\Lecture\superlists\functional_tests_firefox.py",
line 19, in test_can_start_a_list_and_retrieve_it_later
    self.fail('Finish the test!')
AssertionError: Finish the test!

----------------------------------------------------------------------
Ran 1 test in 14.757s

FAILED (failures=1)
```

Failed?
No, it's **just our little reminder**.
Yes! We have a web page!

- Commit:

```
$ git diff # should show our new test in tests.py, and the view in views.py
$ git commit -am "Basic view now returns minimal HTML"
```


The Unit-Test/Code Cycle

- Type `git log –oneline`, for a reminder of what we got up to:

```
$ git log –oneline
ce6363c (HEAD -> master) Basic view now returns minimal HTML
33332ed First unit test and url mapping, dummy view
41dece0 Add app for lists, with deliberately failing unit test
c08183f On branch master
441745d On branch master
```


Useful Commands and Concepts

Running the Django dev server

```
python manage.py runserver
```

Running the functional tests

```
python funtional_tests_firefox.py
```

Running the unit tests

```
python manage.py test
```

The unit-test/code cycle

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

NEXT: Refactoring to Use a Template
What we want to do is **make** our **view function return exactly the same HTML**,
but just using a different process.
That's a **refactor**—when we try to **improve** the **code** *without changing its functionality*.

Not covered here.