Unit Testing Your Web Application

Moshe Zadka – https://cobordism.com

Acknowledgement of Country

Belmont (in San Francisco Bay Area Peninsula)

Ancestral homeland of the Ramaytush Ohlone people

I live in Belmont, in the San Francisco Bay Area Peninsula. I wish to acknowledge it as the ancestral homeland of the Ramaytush Ohlone people.

Outline

Intro to Pyramid

What are Unit tests?

What is httpx?

Unit test examples!

I'll be giving a quick into to Pyramid. Then we'll talk a little about what unit tests are. We will go over httpx, which is a useful tool for writing unit tests for WSGI apps. Finally, the reason we're all here – actually write some unit tests!

0.1 Pyramid Example

Crash Course in Pyramid

Some quick examples!

A full tutorial on Pyramid would be a full semester. There is no chance to do this here. This will cover just enough of Pyramid to give a foundation for the later examples.

0.1.1 JSON Renderer

JSON app: View Function

```
def jsonv(request):
    return {}
```

This function takes a request, and returns an empty dictionary. It is a one-line "view" function.

JSON app: Configurator

```
with pyramid.config.Configurator() as config:
    config.add_route("json", "/json")
    config.add_view(jsonv, route_name="json",
                    renderer="json")
```

The Configurator maps the function to a route. It does so by defining a route based on a path, and then attach the function to the route.

JSON app: WSGI app

```
app = config.make_wsgi_app()
```

Finally, the WSGI application is generated.

This is almost the simplest Pyramid APP that you can write: it has one route and one view.

JSON View Retrieval

```
request = client.get("/json")
request.json()
```

```
{}
```

The app can be run with any WSGI server. After running it, you can grab the URL (/json).

The result is "{}", which the client's .json() method transforms into an empty Python dictionary.

0.1.2 Using Route Matches

Matcher View

```
def matches(request):
    return dict(name=request.matchdict["name"])
with pyramid.config.Configurator() as config:
    config.add_route("matches", "/matches/{name}")
    config.add_view(matches, route_name="matches",
                    renderer="json")
app = config.make_wsgi_app()
```

Up the sophistication ladder a little, this route does something interesting. The route *matches* a part of the path.

The matched part of the path is in the request.matchdict. The return value from the request can use this value.

Matcher View Retrieval

```
request = client.get("/matches/hello")
request.json()
```

```
{'name': 'hello'}
```

When accessing the app, and passing the hello string as part of the path, the response contains the name.

0.1.3 Using Settings

Settings View

```
def setting(request):
    return dict(
        field=request.registry.settings["field"]
    )
with pyramid.config.Configurator(
    settings=dict(field="field_value")
) as config:
    config.add_route("setting", "/setting")
    config.add_view(setting, route_name="setting",
                    renderer="json")
app = config.make_wsgi_app()
```

Another way to make Pyramid applications more sophisiticated is to use the settings parameter to the configurator. This parameter allows passing in settings data: tweaking the configuration, globally-available objects, and the like.

In this example, the settings passes in a field parameter. This parameter can be accessed from a view function via the request.registry.settings parameter.

Settings View Retrieval

```
request = client.get("/setting")
request.json()
```

```
{'field': 'field_value'}
```

Retrieving the relevant URL returns the passed-in setting.

0.1.4 Pyramid crash course review

View Function

```
def lookup(
    request: pyramid.request.Request
) -> str:
#    ^^^
#    Will be jsonified
    matcher = request.registry.settings["matcher"]
#             ^^^^^^^^^^^^^^^^^^^^^^^^^^
#             Access settings through request
    name = request.matchdict["name"]
#          ^^^^^^^^^^^^^^^^^
#          Get parameter from route
    return matcher.get(name.lower(), "default")
#          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
#          Business logic
```

As a way of summary, this is an application that has everything covered so far.

The view uses both the *settings* and the *matchdict* to return a value.

Routing

```
def include_lookup(config):
    config.add_route("lookup", "/lookup/{name}")
#           ^^^^^^^^^|^^^^^^^    ^^^^^^^^^^^^^^^^
#           add a new|route      route path
#           route    |name
    config.add_view(lookup, route_name="lookup",
#           ^^^^^^^^|^^^^^    ^^^^^^^^^^^^^^^^^^^
#           add a   |view     route to attach
#           view to |callable
#           a route |
                    renderer="json")
#                    ^^^^^^^^^^^^^^^
#                    Convert to a JSON response
```

This function adds a route, and a matching view, to a configurator. This has been broken into a function both to keep each piece of code separate to allow over-explaining it, but also because this is not an uncommon pattern in Pyramid.

Settings

```
settings = dict(
#^^
#settings
    matcher=dict(special="hello"),
#    ^^^^^^^ ^^^^^^^^^^^^^^^^^^^^^^^^
#    settings ^^^^^^^^^^^^^^^^^^^^^^^^
#    key       Mapping "special"
#              name to "hello"
)
```

The setting dictionary maps the key matcher to a dictionary with how different paths should be treated. Recall that without "special" handling, the value returned is the default.

Configuration

```
with pyramid.config.Configurator(
#^^^ ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
#Commit at the end       ^^^^^^^^^^
#                        Configurator
#                        class
```

```
    settings=settings,
#    ^^^^^^^^   ^^^^^^^^
#    settings  pre-configured
#              dict
) as config:
#     ^^^^^^
#     configurator
#     object
    include_lookup(config)
```

The Configurator combines the settings and the routing/view configuration into one thing.

App creation

```
app = config.make_wsgi_app(
#                     ^^^^
#          Web Standard Gateway Interface
)
```

Finally, the application is created. It is now possible to serve this application.

0.2 Unit Testing

0.2.1 What is a unit test?

Unit test: rough definition

Runs the code

Might fail on buggy code

Self-contained

Unit tests are defined differently by different people. "What is a unit test" is a question that sparks frequent arguments.

For our purposes, this will be the version we use. The most important part is "self-contained".

This means that it should not do any network traffic, or rely on the machine being in a specific state. This is not the usual definition, which is more specific, but it will be enough for now.

0.2.2 What is a mock?

Mock

Fake object

Configurable behavior

Records access

Often, in order to be *self-contained*, a unit test will use a fake object. Mocks, in Python, are good tools to build such fake objects.

They have configurable behavior: different ways to have them respond to methods. They also record method calls.

0.2.3 Goal of unit test

Good Unit Test

Avoid failing on valid:

Public APIs

Reduce assumptions

A unit test should not fail on valid code. This is not always as possible, and not as easy as it seems!

The unit test might make calls to internal APIs, which can change. This leaves the code valid but breaks the unit test.

The unit test might have assumptions that are not part of the definition of the API. For example, it might be assuming an order of the returned values.

0.2.4 Patching as assumption

Patch Makes Bad Unit Test

Patch: Temporarily replacing a global

Assumption: Global used

Assumption: Used through path

One assumption that unit tests often make is to assume a specific global being used. This is called "patching", and is popular advice.

For example, patching the "random.random" function to return a specific value.

This makes two assumptions:

- The random value is generated by random.random (and not, say, random.uniform).
- The function is accessed as random.random and not via "from random import random".

The test will fail if either of these two assumptions are violated. This includes the case where there are no new bugs being introduced.

0.2.5 Parts of unit test

Unit test anatomy

Set-up

Execution

Post-process (optional)

Verification

All tests, and specifically unit tests, can be thought of as containing four parts:

- Set up: Defining variables or creating objects the unit needs to run.
- Execution: Calling the unit.
- Post-process: Taking the outputs from the unit and doing some processing on it.

- Verification: Checking the post-processed results are as expected.

Some divisions combine the "post process" and "verification" steps. It is useful to separate them to understand unit tests better.

0.3 Python Web Concepts

0.3.1 WSGI

WSGI

Framework

to server

WSGI is a standard for how a *framework*, like Pyramid or Django, can be served by a server, like Gunicorn or uwsgi.

0.3.2 `httpx`

httpx

Requests alternative

Can consume WSGI directly

httpx is a requests alternative. One of the nice features it has is that is can be directed to consume WSGI.

In other words, instead of Framework to server to TCP port to HTTP client, this makes HTTP client to framework. Much easier for testing... hmmmm.....

httpx with WSGI

Creating the client:

```
client = httpx.Client(
    base_url="https://example.com/",
    app=app,
)
```

This creates a client that points to a WSGI application.

httpx with WSGI

Calling the client:

```
response = client.get("/lookup/SPECIAL")
```

Calling a method on the client returns a "response" object.

httpx with WSGI

Using the httpx response object:

```
response.json()
```

```
'hello'
```

Asking the response to get the JSON-parsed version of the body yields the expected output.

0.4 Unit test examples!

0.4.1 System under test

System Under Test: View

```
def lookup(
    request: pyramid.request.Request
) -> str:
    matcher = request.registry.settings["matcher"]
    name = request.matchdict["name"]
    return matcher.get(name, "default")
#                            ^^^^^
#                          Bug: missing .lower()
```

In unit tests, the "system under test" refers to the "unit": the code that needs to be tested. This system has a bug, so unit tests can fail.

System Under Test: App

```
def make_app(special_value):
#^^           ^^^^^^^^^^^^^
#^^          Parameter for
#            the application
#Creating the app from a function,
#not at top level
    settings = dict(
        matcher=dict(special=special_value),
    )
    with pyramid.config.Configurator(
        settings=settings
    ) as config:
        include_lookup(config)
    return config.make_wsgi_app()
```

The system also includes a function to create the app. This makes it easier to write unit tests: they can call the function explicitly to create an app with the right settings.

0.4.2 Mocking Request

Calling function directly

Setup:

```
request = mock.MagicMock()
request.registry.settings = dict(
    matcher=dict(special="hello"),
)
request.matchdict = dict(name="SPECIAL")
```

The first way to test is to call the view function directly. This is sometimes the best way, since it can avoid the routing and app creation. In this example, it does not save too much code.

Calling function directly

Execute:

```
result = lookup(request)
```

The execution is remarkably short: calling the function on the set-up object. This is nice and simple!

Calling function directly

Verification

```
try:
    assert_that(result, equal_to("hello"))
except AssertionError as exc:
    print(exc)

Expected: 'hello'
     but: was 'default'
```

For verification, this test uses hamcrest. It expects the hello response, and gets the default one. The test found the bug, hooray!

0.4.3 Calling WSGI

Calling WSGI

Setup:

```
environ = dict(
    PATH_INFO="/lookup/SPECIAL",
    REQUEST_METHOD="GET",
)
start_response = mock.MagicMock(
    return_value=io.BytesIO()
)
```

The other layer to test this is to go through WSGI. The environment and start_response are part of the WSGI standard.

Calling WSGI

Execute:

```
app = make_app("hello")
base_parts = app(environ, start_response)
```

This time the execution involves calling the make_app function, and then calling the result on the objects from the set up phase.

Calling WSGI

Post-process:

```
parts = [start_response.return_value.getvalue()]
parts.extend(base_parts)
args, kwargs = start_response.call_args
status, headers = args
result = json.loads(b"".join(parts).decode("utf-8"))
```

Post-processing is kind of awkward. There are two ways for WSGI apps to return a response body: using the start_response return value, and returning them directly from the function.

This adds everything up and parses the JSON.

Calling WSGI

Verification

```
try:
    assert_that(result, equal_to("hello"))
except AssertionError as exc:
    print(exc)

Expected: 'hello'
     but: was 'default'
```

The verification is the same: comparing the JSON-decoded result.

0.4.4 Using `httpx`

Using httpx

Setup:

```
app = make_app("hello")
client = httpx.Client(
    base_url="https://example.com/",
    app=app,
)
```

The first version had a problem of calling a non-public API: if the name of lookup is changed, and then is changed in make_app, then the result is valid but the test fails.

The second version had a different problem. Both set-up and post-processing were awkward.

Using httpx can use the best of all worlds. The set-up is more straightforward.

Using httpx

Execute:

```
resp = client.get("/lookup/SPECIAL")
```

The execution is the same as retrieving the result from a real server.

Using httpx

Post-process:

```
result = resp.json()
```

Post-processing is easier. One method call on the response object to get the JSON-parsed body.

Using httpx

Verification:

```
try:
    assert_that(result, equal_to("hello"))
except AssertionError as exc:
    print(exc)
```

```
Expected: 'hello'
     but: was 'default'
```

Verification, once again, is the same.

This tests the *public* API: make_app returns a WSGI app which responds to /location/SPECIAL with hello.

0.5 Summary

0.5.1 Pyramid: Concepts

Summary: Pyramid

View: request, response, route

Route: URL match

Registry: Parameters, shared objects

Configurator: Views, Routes, Registry

WSGI: Configurator to server (or httpx!)

Pyramid maps views to routes. The views can use the registry.

The configurator takes all of these and returns a WSGI app that can be used for HTTP servers or httpx.

0.5.2 Unit Testing

Summary: Unit Test
Verify code, quickly, and safely:
Mock!
Pre-plan
Prefer stable APIs
Unit tests verify code quickly and safely. They use mock to prevent the code from having system dependencies.
Pre-planning the API makes a more testable API

0.6 Bonus material

Bonus
Extra! Extra!

0.6.1 Pyramid Concepts

View

View
Function
Argument: Request
Return value: Response

View Route
Route
to View

View Route Predicates
Route to view can be conditional:
Request type,
Content type,
Arbitrary predicates

Route

Route
Path
to route name

Route: advanced
Path fragment matching
...and more

Registry

Registry: Application Parameters
Settings: ad-hoc dictionary

Configurator

Configurator
Routes
Views
Registry

0.6.2 Routing extras

Static view

Static Value

```
def empty(request):
    return pyramid.response.Response(
        json.dumps({}).encode("ascii"),
        content_type="application/json",
    )
with pyramid.config.Configurator() as config:
    config.add_route("root", "/")
    config.add_view(empty, route_name="root")
app = config.make_wsgi_app()
```

Static View Retrieval

```
request = client.get("/")
request.json()
```

```
{}
```

Using Parameters

Parameter View

```
def params(request):
    return dict(thing=request.params["thing"])
with pyramid.config.Configurator() as config:
    config.add_route("params", "/params")
    config.add_view(params, route_name="params", renderer="json")
app = config.make_wsgi_app()
```

Parameter View Retrieval

```
request = client.get("/params?thing=hello")
request.json()
```

```
{'thing': 'hello'}
```

Using Request Body

Body View

```
def body(request):
    return dict(field=request.json_body["field"])
with pyramid.config.Configurator() as config:
    config.add_route("body", "/body")
    config.add_view(body, route_name="body", renderer="json")
app = config.make_wsgi_app()
```

Body View Retrieval

```
request = client.post("/body", json=dict(field="hello"))
request.json()
```

```
{'field': 'hello'}
```

0.6.3 Registry extras

Registry: Application Parameters
Utility: Scalable configuration

Interfaces and Components

```
class IValueGetter(zope.interface.Interface):
    def get_value(self) -> int:
        ...

@zope.interface.implementer(IValueGetter)
@attrs.frozen
class ValueStorer:
    _value: int

    def get_value(self):
        return self._value
```

Interface-based Registry

```
config = pyramid.config.Configurator()
config.registry.registerUtility(ValueStorer(value=42))
config.registry.getUtility(IValueGetter).get_value()
```

```
42
```

Why Interface-based Registry?

Namespaced
Semantically meaningful
Configuration

0.6.4 Configurator extras

Configurator Inclusion

```
def root_stuff(config):
    config.add_route("root", "/")
    config.add_view(empty, route_name="root")
with pyramid.config.Configurator() as config:
    config.include(root_stuff)
    app = config.make_wsgi_app()

request = client.get("/")
request.json()
```

```
{}
```

Configurator Inclusion

Include allows:

- Callable
- Callable dotted name
- Module (will use "includeme")
- Module dotted name

0.6.5 Scanning

Configurator Scanning

* Scan a package * Find functions decorated "view config"