How to build a simple, lightweight, *custom* test automation framework for functional and end-to-end testing.

A Bit About Me

My name is Derek Sisson, and I've been leading software QA teams for 20 years.

For the last 10 years I've been building custom test automation frameworks for a range of tech companies.

I'm currently the Director of QA at Business Wire, a Berkshire Hathaway company.

Some things I'm *not* going to talk about:

- Test-Driven Development
- Behavior-Driven Development and testing, and the given-when-then syntax
- The reasons why test automation is important
- The dialectic between test automation and exploratory investigation by human beings
- The "checking" vs "testing" labeling dispute
- The various full-featured, do-everything, test automation juggernauts-in-a-box like Tricentis, HP Quality Center, Katalon Studio, etc.

What I *am* going to talk about

How to build a straightforward test automation framework that focuses on functional tests, and that can support end-to-end tests.

This kind of test framework is inherently custom, in that it is closely aligned to the applications under test.

I believe that a test framework should be ephemeral and temporary: the goal should be on learning about the applications under test and on the business logic, and should not be on building the perfect test framework. The goal is on improving the models of business process that support and drive your tests; as your models improve, build better, more effective test frameworks.

A Paradigm Clarification

Developers test differently than specialized testers; we use the same terms, but they mean different things. We also have different aims.

This talk shows how to build a test framework that is external to the code base, and that is focused on deep and broad functional testing. No mocks and no priority on fast-running tests for a CI/CD pipeline. Built correctly, this framework supports end-to-end testing.

Developers often fall prey to the biases of confirmation and congruence, because they are focused on verifying and validating their code to requirements. The specialist tester looks for the non-optimal behavior; this talk is about supporting this attitude in test automation.

A test framework is an experiment.

I want to reinforce this point: a test framework is an experiment.

I'm not going to reveal any deep secrets, but the combination of elements is non-obvious. The concepts are simple, and once you build one test framework in this pattern, you can build subsequent versions very quickly.

Premature concretization is your enemy, and optimization is not your friend, *yet*.

The Vision

- Functional tests are *atomic*, scoped to specific areas of functionality. You need a bunch of these atomic tests to adequately examine the functionality of web sites and APIs.
- End-to-end tests look at paths through areas of functionality. E2E tests play connect-the-dots with the atomic functional tests.
- E2E tests are best when written on a layer of abstraction above the atomic functional tests: combine reasonable sequences of atomic functional test interactions into abstractions that model business processes or “chunks” of flow.
- Combine application wrappers and abstraction models to make E2E tests easier to define and conceptualize.

Our Example Applications

For the purpose of explaining how to hook up a test framework to target applications for functional and end-to-end tests, let's pretend that we have:

- A website front end that provides a commerce interface for selecting products, adding them to a shopping cart, and purchasing them.
- To keep things simple, our products are t-shirts, in a limited range of colors and sizes.
- Our website will allow customers to generate a slogan to be printed on the t-shirts.
- The customization is controlled via an API.
- Shipping is controlled through an API.

The components of a test automation framework:

- Tests
- A test runner
- Test tools
- A scaffolding of framework utilities and supporting code
- Test data and fixtures
- Application wrappers
- Abstraction models
- More, better tests!

Components: Test Runner

The test runner is responsible for

- collecting tests
- executing tests
- reporting on test results
- providing some assertion helpers

With python, the simple default is **unittest**, but I prefer to use **pytest**. The general concepts apply to any *unit test runner.

Components: Test Runner Continued

Pytest doesn't have a specific home in the test framework, because it's imported into your modules and tests where it's needed. However, there are some configuration and supporting files. For example:

```
my_test_framework
    |- my_test_framework
        |- tests
            - conftest.py
            - pytest.ini
            - test_simple.py
     - requirements.txt
```

Pytest supports markers, fixtures, and other tools that are added to files. We'll get to those.

Components: Test Runner Continued

```
$ pytest tests --collect-only
============== test session starts ==============
collected 3 items
<Module ‘test_simple.py'>
  <Class 'SomeTests'>
    <Instance '()'>
      <Function 'test_this'>
      <Function 'test_that'>
      <Function 'test_something_else'>
```

Component: Test Tools

Test tools are libraries that support ways to interact with target applications, or that supply specialized logic for performing some work. Some examples:

- UI drivers (selenium wrapper, appium wrapper)
- HTTP library (requests)
- HTML & XML parsers (Beautiful Soup, requests-html)

```
my_test_framework
    |- my_test_framework
        |- framework
            - selenium_utils.py
        |- tests
            - conftest.py
            - pytest.ini
            - test_simple.py
     - requirements.txt
```

Component: Framework utility code

You will often have code that isn't test code, but that supports the tests, the data used in tests, the reporting of test results, logging, etc. That's the kind of code that goes into the framework utility code.

Examples:

- When dealing with json data, I like to pretty format the json in my logging, so I have a pretty logging method in my framework/utils.py.
- I put custom exceptions in framework/exceptions.py
- If I have code that supports the selenium wrapper, that goes in framework/selenium_utils.py

```
my_test_framework
    |- my_test_framework
        |- framework
            - selenium_utils.py
            - utils.py
            - exceptions.py
        |- tests
            - conftest.py
            - pytest.ini
            - test_simple.py
    - requirements.txt
```

The specifics of your supporting code will be contextually relevant to you; for "housekeeping", it makes sense to keep this code out of your test code.

Component: Test Fixtures

A test may require certain setup and preparatory steps before it can be run. A test may also require certain clean-up and post-execution steps in order to return the system to a ready state.

Test fixtures are where you put these setup and takedown steps, as well as other test-supporting system actions.

Component: Test Fixtures

Let's create a fixture that initiates the Chrome driver to be used in our tests.

tests/conftest.py

```
@pytest.yield_fixture(scope='session')
def driver(request):
    from selenium import webdriver
    driver = webdriver.Chrome()
    driver.implicitly_wait(10)
    yield driver
    driver.quit()
```

To hook this fixture up to a particular test method, just add the fixture name as a parameter. For example:

```
def test_something(self, driver):
    page = driver.get('https://cnn.com')
```

Component: Test Data

The specific data used in your test cases will typically follow a migration route starting inside your test cases, and moving into a central “repository” in your framework.

Method-local test data:

```
def test_add_xl-blue-shirt(self):
    shirt = {‘color’: ‘blue’, ‘size’: ‘xl’}
    # navigate to the product page for xl blue shirts
    # verify that you can find xl blue shirt
    # add xl blue shirt to cart
    # verify that xl blue shirt is in cart
```

Module-local test data:

```
shirt = {‘color’: ‘blue’, ‘size’: ‘xl’}
def test_add_shirt(self, shirt):
    # navigate to the product page for `shirt`
    # verify that you can find `shirt`
    # add `shirt` to cart
    # verify that `shirt` is in cart
```

Component: Test Data Continued

The best place for your test data is where it can be shared among all of your tests, and your models.

```
my_test_framework
   |- my_test_framework
      |- data
         - shirts.py
      |- framework
         - selenium_utils.py
         - utils.py
         - exceptions.py
      |- tests
         - conftest.py
         - pytest.ini
         - test_simple.py
    - requirements.txt
```

Component: Test Data Continued

data/shirts.py

```
shirt = {‘color’: ‘blue’, ‘size’: ‘xl’}
```

tests/test_simple.py

```
from my_test_framework.data import shirts
def test_add_shirt(self, shirt):
    # navigate to the product page for `shirt`
    # verify that you can find `shirt`
    # add `shirt` to cart
    # verify that `shirt` is in cart
```

An important detour into parametrization

Parametrization is a test design superpower. If you have series of tests that vary only in the specific test data, for example what should happen with a blue XL t-shirt vs. a red XL t-shirt, then you should parametrize.

Parametrization works roughly like this:

1. Identify a scenario where you are dealing with essential the same test but with different data or action parameters.
2. Generalize the test steps.
3. Build a data model that can be iterated over.

With pytest, for a test method that is parametrized, during test collection a new test method for each parameter is dynamically created.

So...

So let's create the following data model:

data/shirts.py

```
shirts = {
        'blue_s': {'id': 'blue_s', 'color': 'blue', 'size': 'small'},
        'blue_m': {'id': 'blue_m', 'color': 'blue', 'size': 'medium'},
        'blue_l': {'id': 'blue_l', 'color': 'blue', 'size': 'large'},
        'blue_xl': {'id': 'blue_xl', 'color': 'blue', 'size': 'extra-large'},
        'red_s': {'id': 'red_s', 'color': 'red', 'size': 'small'},
        'red_m': {'id': 'red_m', 'color': 'red', 'size': 'medium'},
        'red_l': {'id': 'red_l', 'color': 'red', 'size': 'large'},
        'red_xl': {'id': 'red_xl', 'color': 'red', 'size': 'extra-large'}
}
```

And parametrize a test method

tests/test_simple.py

```
from my_test_framework.data.shirts import shirts

@pytest.mark.parametrize('shirts_data',  # name of parameter
                         shirts,         # data model
                         ids=[shirts[shirt]['id'] for shirt in shirts])  # ids for tests
def test_add_shirt(self, 'shirts_data'):
    # disambiguate parameter data
    id = shirts_data[‘id’]
    color = shirts_data[‘color’]
    size = shirts_data[‘size’]
    # navigate to the product page for `shirt`
    # verify that you can find `shirt`
    # add `shirt` to cart
    # verify that `shirt` is in cart
```

And generate parametrized test cases, dynamically!

```
$ pytest tests/test_params.py --collect-only
============= test session starts =============
collected 8 items
<Module 'test_params.py'>
  <Class 'SomeTests'>
    <Instance '()'>
      <Function 'test_add_shirt[blue_s]'>
      <Function 'test_add_shirt[blue_m]'>
      <Function 'test_add_shirt[blue_l]'>
      <Function 'test_add_shirt[blue_xl]'>
      <Function 'test_add_shirt[red_s]'>
      <Function 'test_add_shirt[red_m]'>
      <Function 'test_add_shirt[red_l]'>
      <Function 'test_add_shirt[red_xl]'>
```

Component: Application Wrappers

The standard application wrapper is the Page Object Model, where pages are described in a class structure. The conceit is that these classes “know about themselves and their affordances”.

The goal is to abstract out the specifics of the interactions with these pages from the test cases to supporting code, making it easier to maintain the interface to the application under test.

Pro Tip: Use iPython for Selenium Exploration

```
In [1]: from selenium import webdriver

In [2]: d = webdriver.Chrome()

In [3]: d.get('https://www.youtube.com')

In [4]: vid = '1JM90JmrBfU'

In [5]: sel_search = 'form input#search'

In [6]: search = d.find_element_by_css_selector(sel_search)

In [7]: search.click()

In [8]: search.send_keys(vid)

In [9]: search.submit()

In [10]: video = d.find_element_by_css_selector('a[href="/watch?v=%s"]' % vid)

In [11]: video.get_attribute('href')
Out[11]: 'https://www.youtube.com/watch?v=1JM90JmrBfU'
```

Component: Application Wrappers Continued

```
class Mytests(object):

    def test_search_by_id(self):
        d = webdriver.Chrome()
        d.implicitly_wait(10)

        d.get('https://www.youtube.com')

        # use search to find target video
        search = d.find_element_by_css_selector('form input#search')
        search.click()
        search.send_keys('1JM90JmrBfU')
        search.submit()
        time.sleep(2)

        # verify that the correct video was returned
        video = d.find_element_by_css_selector('a[href="/watch?v=1JM90JmrBfU"]')
        assert video.get_attribute('href') ==
                        'https://www.youtube.com/watch?v=1JM90JmrBfU'
        d.quit()
```

An important detour to discuss test automation architecture.

Test automation design is a learning exercise. There's a natural evolution of naive, simple test instructions being revised with more concise instructions, more abstractions, and more supporting code, so that the test cases focus less on setup and more on the actual test logic.

Don't get hung up on the idea of perfection, optimization, or long-term performance. Focus on returning some value from test automation right now. As you write more tests and gain better experience on how to test your apps, you will see the opportunities to abstract and improve your models.

If in doubt, put the logic in the test method/test file. Further experience will show where to move it to.

Adding application wrappers

```
my_test_framework
    |- my_test_framework
        |- apps
            |- website
                - base_page.py
                - home.py
                - cart.py
        |- data
            - shirts.py
        |- framework
            - selenium_utils.py
            - utils.py
            - exceptions.py
        |- tests
            - conftest.py
            - pytest.ini
            - test_simple.py
    - requirements.txt
```

Base Object for shared stuff

```
import pytest
class BaseObject(object):

    domain = 'https://www.youtube.com'
    # selectors
    sel_search = 'form input#search'

    def search(self, driver, vid):
        path = '/watch?v=%s' % vid
        sel_vid = 'a[href="%s"]' % path

    # use search to find target video
    search = driver.find_element_by_css_selector(self.sel_search)
    search.click()
    search.send_keys(vid)
    search.submit()

    # verify that the correct video was returned
    video = driver.find_element_by_css_selector(sel_vid)
    try:
        video.get_attribute('href') == self.domain + path
        return True
    except:
        ValueError, 'ERROR: actual path "%s" not equal to expected path "%s"'
            % (video.get_attribute('href'), path)
```

Specific Objects for specifics

```
import pytest
from my_test_framework.apps.base import BaseObject


class HomePageObject(BaseObject):

    def __init__(self, driver):
        driver.get(self.domain)
```

Page Objects allow greater abstraction in test cases

```
class AdvancedYoutubeTests(object):

    @pytest.mark.parametrize('video', [vids[v] for v in vids], ids=[v for v in vids])
    def test_search_by_vid(self, driver, video):
        """ For a given video, see if you can find it by search. """

# setup
id = video['vid']  # extract the id from the parametrized data input
this_video = VideoDetailObject(id)  # load the page object model

# load the home page
page = HomePageObject(driver)

# perform the search
assert page.search(driver, id)
```

Component: Application Wrappers continued

You can treat an API in pretty much the same way, using an Endpoint Object Model build on top of an HTTP library like requests.

Let's build an EOM for our pretend custom lettering API.

```
my_test_framework
    |- my_test_framework
        |- apps
            |- website
                - base.py
                - home.py
                - cart.py
            |- api
                - base.py
                - letters.py
        |- data
            - shirts.py
        |- framework
            - selenium_utils.py
            - utils.py
            - exceptions.py
        |- tests
            - conftest.py
            - pytest.ini
            - test_simple.py
    - requirements.txt
```

Base Object for API

```
class BaseEndpoint(object):
    """ Common ancestor for all endpoints. """
    base_url = 'http://www.colourlovers.com/api/'

    def get(self, url, ex=200, **kwargs):
        res = requests.get(url, params=kwargs)

        if not res.status_code == 200:
            raise UnexpectedStatusCodeException(response=res)

        return res

    def verify_keys_in_response(self, response_keys):
        # be strict about checking keys
        expected = self.expected_keys
        actual = list(response_keys)
        actual.sort()

        if sorted(expected) == sorted(actual):
            return True
        else:
            raise JsonPayloadException('Actual keys != expected keys.')
```

Specific Object for an Endpoint

```
class ColorEndpoint(base_endpoint.BaseEndpoint):

    def __init__(self):
        self.name = 'color'
         self.endpoint = 'color/'
         self.endpoint_url = self.base_url + self.endpoint
         self.expected_keys = [‘apiUrl’, ‘badgeUrl', ‘dateCreated']
        self.expected_keys.sort()


def get_color(self, hex, format='json', verbose=True, **kwargs):
    kwargs['format'] = format
    url = self.endpoint_url + hex

    # pass to the base endpoints *requests* wrapper
    res = self.get(url, **kwargs)

    if verbose:
        logger.info('Response json: \n%s' % plog(res.json()))
    return res
```

Endpoint Objects allow greater abstraction in test cases

```
class ExampleApiTests(object):

    # create instances of endpoint
    color_endpoint = color.ColorEndpoint()

    def test_get_color(self):
       # setup: make API request
        data = '000000'
        res = self.color_endpoint.get_color(data)

        # test point: verify the correct response for a correct api call
        assert res.status_code == 200

        # test point: verify that we get back the same hex that we requested
        assert res.json()[0]['hex'] == data

        # test point: verify that the json keys are correct
        assert self.color_endpoint.verify_keys_in_response(res.json()[0].keys())
```

Custom Exceptions Support Functional Tests

```
class UnexpectedStatusCodeException(Exception):
    """
        Raise this exception when the server response code from an http request is not
        a success code.
        Capture the requests response object and make that available.

        # pass the response object to the exception
        >>> raise UnexpectedStatusCodeException(response=res)

        # then catch and introspect the exception's property
        >>> except UnexpectedStatusCodeException as e:
            ... print(e.response.status_code)
        400
    """
    def __init__(self, response):
        Exception.__init__(self, response)
        self.response = response


class PageIdentityException(Exception):
    """
        Raise this exception when a page fails its self validation of identity.
    """
    pass
```

Plumbing

Everything we've discussed so far has been the basic structure setup for the meat of test automation: business process abstractions that support end-to-end automation.

Our app wrappers support extensive deep functional tests. Our business abstractions will let us define happy paths through our functional affordances so that we can focus on user journeys.

To Recap...

We have discussed most of the core elements of a custom test automation framework:

- A testrunner to collect, run, and report on tests. Pytest is the glue that holds together the test automation framework, and calls on pytest are scattered throughout the framework. Pytest is how the framework is invoked and run. When you integrate your framework in a build system or CI pipeline, you are calling pytest from the command line.
- The framework contains some specialized tools that are used by the tests to interact with applications under test or perform other test-related logic.
- The framework has wrappers for the application to act as a sort of internal interface for the test cases.
- We have data models that keep data out of the test case files. These data models parametrization of the test cases, so that a few general test cases can be dynamically expanded into instances for each data member.
- We've tried hard to refine the test cases to the core test-specific logic and abstract out anything that doesn't belong in the test cases.

All of this has been the foundation of the most important part of a custom framework: the business models.

Component: Abstraction Models

An abstraction model is a way of rolling up individual interactions, actions, or touch-points into a form that can support the simplification of end-to-end testing.

In our example, we have the following business processes that need to be abstracted:

- The definition of products. The set of products is likely to change over time, so we need a robust way of extending our tests.
- The definition of orders, or the combination of products.
- The pricing model. The data model for products can cover individual pricing, but the pricing of combinations of products, discounting, etc. is definitely something we want to do algorithmically and outside of test case logic.

Component: Abstraction Models continued

```
my_test_framework
    |- my_test_framework
        |- apps
            |- website
                - base.py
                - home.py
                - cart.py
            |- api
                - base.py
                - letters.py
      |- data
            - shirts.py
      |- framework
            - selenium_utils.py
            - utils.py
            - exceptions.py
      |- models
            - products.py
            - carts.py
            - pricing.py
      |- tests
            - conftest.py
            - pytest.ini
            - test_simple.py
   - requirements.txt
```

End-to-end tests rely on abstractions

Example end-to-end test case for our pretend t-shirt commerce site:

```
def test_e2e(self, driver, orders):
    # disambiguate the order

    # for each item in the order:
    # find it
    # add it to the cart

    # instantiate the pricing model
    # validate the expected price with the actual price from the website

    # add the custom lettering

    # place the order
    # check the DB, verity order details and status

    # instantiate the API EOM
    # query the API for the custom lettering, verify it matches
```

Next Steps

We have discussed:

- The architecture of a simple test automation framework.
- Wrappers for applications under test.
- Business process abstractions (“models”).
- Writing end-to-end tests.

Where do you go from here?

- Add more applications.
- Wrappers for integrations and systems to support deeper checks, for example database validations.
- Containerize the test framework.
- Hook the test framework up to a CI pipeline like Travis or Jenkins.
- Improve your models! Move models into their own projects.
- Plan the next iteration of the framework.

Links

- This talk's deck: https://github.com/dsisson/talk-framework
- The example teaching tool framework this talk was based on: https://github.com/dsisson/welkin
- My old, old writings on QA: http://philosophe.com/