You Vs. The Real World: Testing With Fixtures

Kumar McMillan

http://flickr.com/photos/loredenizen/14435744/

Overview

- Why testing the "real world" is important
  - Upsides/ downsides
- Testing with the "fixture" module
  - How to use it
  - Why it was created, some history

What is a real world test?

- Integration test, behavior test, functional test, black box test
- Runs your program in the most “real” way possible
- Doesn’t know about implementation
- Not a unit test

How would you test the real world?

- GET/ POST request
- main()
- assert output rendered something
- assert new artifacts exist

Why is this the first test you should write?

- Shopping cart as counter-example
- You've unit tested the Cart
- You've unit tested the Order object
- still, the front page doesn't load
- Money lost

Why is this the first test you should write?

- An integration test might ...
  - Load the product page, add a product to a cart, and place an order
  - Assert that money was collected
- Doesn't care about the Cart or Order objects

Integration Tests vs. Unit Tests

An integration test proves the system works	A unit test proves "how" the system works
...lets the implementation evolve freely	...ensures the implementation is solid

An integration test needs fixtures

- Environment of your program
- Inputs to your program
- Data your program consumes
- The test's "setup" function

Use cases for testing with fixtures...

- Test a user login
- Test a data import script
- Test a billing calculation process
- Data logic / triggers / stored procedures
- re-produce a bug

Mock objects or fixtures?

- Fixtures are not mock objects
- Real objects and real data result in more accurate tests, better coverage
- Good reasons to use mock objects:
  - speed
  - a reliable resource
- “switchable” mock objects

Enter “fixture”

- A python module for loading and referencing test data
  - provides an interface for loading tabular data into storage media
    - designed primarily for databases
- easy_install fixture
- code.google.com/p/fixture

The idea

- A subclass of DataSet defines the data
- A fixture object knows how to load data
- A data instance can be used to reference loaded data
- setup / teardown

A DataSet

```
class ClientData(DataSet):
    class joe:
        company=”Joe, Inc.”
        contact=”Joe The Client”
```

Referencing DataSet values

```
class SiteData(DataSet):
    class joes_site:
        url=”joe.com”
        client_id = \
            ClientData.joe.ref(‘id’)
```

Inheriting rows

```
class SiteData(DataSet):
    class joes_site:
        url=”joe.com”
        client_id = \
            ClientData.joe.ref(‘id’)
    class bobs_site(joes_site):
        url=”joesbrotherbob.com”
```

Defining a fixture

```
db = SQLAlchemyFixture(
                env=mapped.classes.module,
                session=session,
                style=NamedDataStyle())
```

Behind the scenes (mapped class example)

```
name = style.guess_storable_name(
                                ‘ClientData’)
Client = getattr(env, name)
row = Client()
row.company = “Joe, Inc.”
row.contact = ”Joe The Client”
session.save(row)
session.delete(row)
```

But... the magic!

```
class AnythingData(DataSet):
    class Meta:
        storable=Client
        primary_key=[‘email’]
    class joe:
        email=”joe@joe.com”
        company=”Joe, Inc.”

db = SQLAlchemyFixture(session=s)
```

DataTestCase

```
class TestSite(DataTestCase,
               unittest.TestCase):
    fixture = db
    datasets = [SiteData]

    def test_joes_site(self):
        Client.get(
          self.data.ClientData.joe.id)
```

@db.with_data (for nose)

```
@db.with_data(ClientData, SiteData)
    def test_joes_site(data):
        Client.get(data.ClientData.joe.id)
```

nose

- not required
- easy_install nose
- `nosetests`
- discovers tests and runs them
- code.google.com/p/python-nose/

with db.data() as d

```
with db.data(SiteData) as d:
    assert Site.get(
                d.SiteData.joes_site.id)
```

Accessing Data

```
with db.data(SiteData,
             ClientData) as d:
    d.ClientData.joe.id
    d.ClientData.joe.company
    d.SiteData.joes_site.url
    d.SiteData.bobs_site.url
```

What media is supported?

- from fixture import SQLAlchemyFixture
- from fixture import SQLObjectFixture
- CSV?
- Django?

Regression testing with generated data

```
fixture my_sqlalchemy.table.foo \
        --dsn=”postgres://...” \
        --query=”id=1234”
```

The “fixture” command

- Send it a “path” to an object
  - sqlalchemy: a Table, mapped class
  - SQLObject class
- configure query parameters
- you get DataSet code, foreign keys expanded
- A complete “snapshot” of the query

Where fixture came from

- inspired by Ruby on Rails' fixtures
- python code, not YAML
- in 2005 created python module testtools.fixtures
- found many problems with the testtools.fixtures interface
- foreign keys, oh my

The new fixture

- fixture is a 2nd generation interface
- fixture attempts to be even more pythonic
- both testtools.fixtures and fixture developed for a large ETL test suite
- fixture still has little real world experience

Where fixture is going

- In need of brave souls to incorporate fixture into their test suites
  - submit issues to code.google.com/p/fixture
  - In need of end-user documentation!
  - Questions?