test-helpers

Release 1.5.4

May 11, 2015

Contents

The Test Helpers library exists to consolidate some of the repetition that has arisen in our tests while providing a useful set of generic testing utilities.

This library offers a set of base test cases, testing mixins, and testing utilities to make interacting with third party services, testing metrics generation, and creating mocks or patches easier.

CHAPTER 1

Examples

The library is designed to be simple and modular. By using mixins to extend the test cases functionality we can write more expressive tests in fewer lines of code.

```
>>> from test_helpers import mixins, bases
>>> class WhenFooingBar(mixins.PatchMixin, bases.BaseTest):
...
...     patch_prefix = 'module.submodule'
...
...     @classmethod
...     def configure(cls):
...         cls.foo = cls.create_patch('foo', return_value=True)
...
...     @classmethod
...     def execute(cls):
...         function_under_test()
...
...     def should_have_called_foo(cls):
...         self.foo.assert_called_once_with()
```

CHAPTER 2

Documentation

2.1 Base Test Cases

The base test cases module contains useful classes that inherit from the unit test standard library and optionally inherit from an arbitrary number of mixins. The purpose of these classes is to provide an integration point for the mixins and to help promote the usage of the Arrange-Act-Assert testing methodology used here at AWeber.

class test_helpers.bases.**BaseTest**(*methodName='runTest'*)
Base class for the AWeber AAA testing style.

This implements the Arrange-Act-Assert style of unit testing though the names were chosen to match existing convention. New unit tests should use this as a base class and replace the *configure()* and *execute()* methods as necessary.

classmethod annihilate()
Clean up after a test.

Unlike *tearDownClass()*, this method is guaranteed to be called in all cases. It will be called even if *configure()* fails so do not do anything that depends on it having been successful without checking if it was.

classmethod configure()
Extend to configure your test environment.

classmethod execute()
Override to execute your test action.

maxDiff = 100000

classmethod setUpClass()
Arrange the test and do the action.

If you are extending this method, then you are required to call this implementation as the last thing in your version of this method.

classmethod tearDownClass()

2.2 Testing Mixins

Collection of functionality mix-ins.

This module contains standalone classes that can be safely mixed into the *BaseTest* class. Each mixin extends the functionality of the test case by adding behaviors, methods, and attributes - for example, patching well-understood functionality or automatically creating/destroying an in memory UDP server.

When creating a mixin it is important to take not that you should strive to keep the Method Resolution Order (MRO) as clean as possible. Each mixin class should ideally only inherit from `object`.

class `test_helpers.mixins.`**EnvironmentMixin**
Mix this class in if you manipulate environment variables.

A common problem in testing code that uses environment variables is forgetting that they are really globals that persist between tests. This mixin exposes methods that make it easy and safe to set and unset environment variables while ensuring that the environment will be restored when the test has completed.

You need to mix this in over a class that calls the `configure` `annihilate` class methods around the code under test such as *test_helpers.bases.BaseTest*.

classmethod annihilate()

classmethod configure()

classmethod set_environment_variable(*name*, *value*)
Set the value of an environment variable.

classmethod unset_environment_variable(*name*)
Clear an environment variable.

class `test_helpers.mixins.`**PatchMixin**
A mixin to allow inline patching and automatic un-patching.

This mixin adds one new method, `create_patch` that will create and activate patch objects without having to use the decorator.

In order to make use of the patching functionality you need to set the `patch_prefix` class attribute. This attribute should be the python module path whose objects you want to patch. For example, if you wanted to patch the `baz` object in the `foo.bar` module your patch prefix might look like `foo.bar`. When creating a patch you can now just refer to the object name like `cls.create_patch('baz')`.

This usage of this mixin as opposed to the patch decorator results in less pylint errors and not having to think about the order of decorator application.

Example Usage:

```
class MyTest(mixins.PatchMixin, bases.BaseTest):

    patch_prefix = 'my_application.module.submodule'

    @classmethod
    def configure(cls):
        cls.foo_mock = cls.create_patch('foo')
        cls.bar_mock = cls.create_patch('bar', return_value=100)

    @classmethod
    def execute(cls):
        function_under_test()

    def should_call_foo(self):
        self.foo_mock.assert_called_once_with()

    def should_return_100_from_bar(self):
        self.assertEqual(100, self.bar_mock.return_value)
```

classmethod **create_patch**(*target*, ***kwargs*)
Create and apply a patch.

This method calls `mock.patch()` with the keyword parameters and returns the running patch. This approach has the benefit of applying a patch without scoping the patched code which, in turn, lets you apply patches without having to override *setUpClass()* to do it.

Parameters **target** (*str*) – the target of the patch. This is passed as an argument to `cls.patch_prefix.format()` to create the fully-qualified patch target.

patch_prefix = ''

classmethod **setUpClass**()

classmethod **stop_patches**()
Stop any active patches when the class is finished.

classmethod **tearDownClass**()

2.2.1 Tornado Specific Helpers

2.3 MongoDB Helpers

2.4 Postgres Helpers

2.5 Rabbit MQ Helpers

2.6 Testing Utilities

The testing utilities module contains standalone functionality for that might be useful for a select number of test cases. These functions can be selectively applied to a small subset of tests so they might not warrant the full capacity of mixin behavior.

The utilities within this module are typically simple functions or decorators that ease a specific testing task, such as creating patches.

test_helpers.utils.**create_ppatch**(*path*)
Create a partial ppatch object that will only require the object name

test_helpers.utils.**ppatch**(*path*, *object_name*, ***kwargs*)
Creates a fully qualified patch object.

This function will act as a wrapper that will allow us to create a partial function representation. That will remove the need to keep passing the same path to the patch object.

2.7 Indices and tables

- genindex
- search

2.8 Release History

- Next Release
  - Removed Makefile from the development environment.
  - Make overriding `execute` optional.
- 1.5.4
  - Relax the pin on the six library
- 1.5.3
  - `test_helpers.postgres` module added
  - `test_helpers.mongo` module added
- 1.5.2
  - `test_helpers.rabbit` module added
- 1.5.1
  - Allow use of *mixins* module without requiring the `tornado` package
- 1.5.0
  - Initial public release.

t

A

B

C

E

M

P

S

T

U