test-helpers

Release 1.5.4

Dec 07, 2017

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

class test_helpers.mixins.tornado.**JsonMixin**
Mix in over *TornadoMixin* to enable JSON handling.

This mix in extends *TornadoMixin.request()* so that it will automatically serialize request data and deserialize responses as JSON. It does honor the HTTP content type headers so it will not accidentally deserialize a non-JSON response or serialize an already serialized request.

classmethod request(**args*, ***kwargs*)
Send a request and process the response.

Parameters

- **args** – positional parameters to send to `super().request`
- **kwargs** – keyword parameters to send to `super().request`

Returns either a `tornado.httpclient.HTTPResponse` instance or `None` if the request timed out.

class test_helpers.mixins.tornado.**TornadoMixin**
Test tornado applications with AAA testing.

Mix this class in over *test_helpers.bases.BaseTest* or similar classmethod-based testing class and you can directly test tornado-based applications.

Usage

```
from unittest import TestCase
import test_helpers.mixins.tornado
import myproject

class WhenMyApplicationsGets(mixins.tornado.TornadoMixin, TestCase):

    @classmethod
    def setUpClass(cls):
```

```
        super(WhenMyApplicationGets, cls).setUpClass()
        cls.start_tornado(myproject.Application())
        cls.execute()

    @classmethod
    def tearDownClass(cls):
        super(WhenMyApplicationGets, cls).tearDownClass()
        cls.stop_tornado()

    @classmethod
    def execute(cls):
        cls.response = cls.get('/index')

    def should_return_ok(self):
        self.assertEqual(self.response.code, 200)
```

Attributes

This mix-in creates three useful attributes in addition to the methods. Each of the attributes is initialized by *start_tornado()* and will be None until that method is called.

client
The tornado.httpclient.HTTPClient instance used to interact with the application.

io_loop
The tornado.ioloop.IOLoop instance that the application is attached to.

url_root
The URL root used to interact with the application. This refers to host and port that *io_loop* is attached to.

request_timeout
The number of seconds to run the IO loop for before giving up on the request.

classmethod delete(*path*, ***kwargs*)
Issue a DELETE request.

classmethod get(*path*, ***kwargs*)
Issue a GET request.

classmethod head(*path*, ***kwargs*)
Issue a HEAD request.

classmethod options(*path*, ***kwargs*)
Issue a OPTIONS request.

classmethod patch(*path*, *body*, ***kwargs*)
Issue a PATCH request.

classmethod post(*path*, *body*, ***kwargs*)
Issue a POST request.

classmethod put(*path*, *body*, ***kwargs*)
Issue a PUT request.

classmethod request(*method*, *path*, ***kwargs*)
Issue a request to the application.

Parameters

- **method** (*str*) – HTTP method to invoke
- **path** (*str*) – possibly absolute path of the resource to invoke

- **kwargs** – additional arguments passed to the tornado.httpclient.HTTPRequest initializer

Returns either a tornado.httpclient.HTTPResponse instance or None if the request timed out.

The *path* parameter can be a relative path or an absolute URL. It will be joined to *url_root* before the request object is created.

classmethod start_tornado(*application*)
Start up tornado and register *application*.

Parameters **application** – anything suitable as a Tornado *request callback* (see tornado.httpserver.HTTPServer)

This method binds a socket to an arbitrary temporary port, creates a new Tornado IOLoop, and adds the *application*' to it. Calling any of the request-related methods will start the IOLoop and run it until a response is received.

classmethod stop_tornado()
Terminate the tornado IO loop.

class test_helpers.mixins.tornado.**TornadoTest**(*methodName='runTest'*)
Tornado version of *BaseTest*.

This class acts as a replacement for *BaseTest* with the Tornado magic pre-mixed. All that you have to do is:

1. set *tornado_application* as a top-level class attribute *OR*
2. pass the application keyword to *configure()* and make sure that this class is the first one in the __mro__

In either case, the application is the Tornado *request callback* that is being tested. See tornado.httpserver.HTTPServer for a complete description of what constitutes a *"request callback"*.

classmethod annihilate()
Terminates the Tornado application.

classmethod configure(*application=None*)
Configures the Tornado IOLoop.

Parameters **application** – overrides *tornado_application*

tornado_application = None
The Tornado request handler callable.

2.3 MongoDB Helpers

class test_helpers.mongo.**TemporaryDatabase**(***kwargs*)
Creates a temporary MongoDB database that is destroyed automatically.

Parameters

- **host** (*str*) – Database to connect to. This defaults to :envvar: MONGOHOST or localhost if omitted.
- **port** (*int*) – Port number that the database is listening on. This defaults to :envvar: MONGOPORT` or ``27017 if omitted.

Instances of this class will create a bare MongoDB database with a single collection named test_helpers containing a single document with a create date for tracking purposes. When the test process exits all databases

created will be destroyed automatically. Under the hood it uses `pymongo` and registers a single cleanup function with :func‘atexit.register‘.

Usage Example

```
from test_helpers import mongo

_testing_db = mongo.TemporaryDatabase()

def setup_module():
    _testing_db.create()
```

create()
Create the temporary database if it does not exist.

drop()
Drop the temporary database if it was created.

set_environment()
Export MongoDB environment variables for the database.

This exports the `MONGOHOST`, `MONGOPORT`, and `MONGODATABASE` environment variables

2.4 Postgres Helpers

class test_helpers.postgres.**TemporaryDatabase**(***kwargs*)
Creates a temporary database that is destroyed automatically.

Parameters

- **user** (*str*) – the database user to connect with. This defaults to `PGUSER` or `postgres` if unset.
- **password** (*str*) – the database password to connect with. This defaults to `None`.
- **host** (*str*) – the database server to connect to. This defaults to `PGHOST` or `localhost` if omitted.
- **port** (*str*) – port number that the database server is listening on. This defaults to `PGPORT` or `5432` if omitted.
- **kwargs** – additional psycopg2 connection parameters

Instances of this class will create an isolated database from a template and ensure that it is destroyed when the test process exits. Under the hood it issues DDL commands over a psycopg2 connection to manage the database and registers a single cleanup function with `atexit.register()`.

Usage Example

```
from test_helpers import postgres

_testing_db = postgres.TemporaryDatabase()

def setup_module():
    _testing_db.create()
    _testing_db.set_environment()

    # from this point on, the standard Postgres environment
    # variables PGDATABASE, PGHOST, PGPORT, and PGUSER are
    # set to connect to the temporary database
```

By default, this will connect to postgres using the *postgres* database and clone the *template0* database. The starting database is controlled by the *STARTING_DATABASE* class attribute and the template database can be changed by passing it to the *create()* method.

Once a temporary database has been created, you can either call the *set_environment()* method to export the standard set of Postgres environment variables (e.g., PGHOST) or get a copy of the connection parameters from the *connection_parameters* property.

STARTING_DATABASE = 'postgres'
Database to connect to when cloning the template.

connection_parameters
Keyword parameters for psycopg2.connect().

create(*template='template0'*, ***options*)
Create the temporary database if it does not exist.

Parameters

- **template** (*str*) – the name of the database to use as a template for the new database. This defaults to template0 if omitted.
- **options** – additional parameters to use in the CREATE DATABASE command.

drop()
Drop the temporary database if it was created.

set_environment()
Export Postgres environment variables for the database.

This exports the PGUSER, PGHOST, PGPORT, and PGDATABASE environment variables set to match *connection_parameters*.

2.5 Rabbit MQ Helpers

class test_helpers.rabbit.**RabbitMqFixture**(*host*, *user*, *password*)
Manages a Rabbit MQ virtual host.

Create an instance of this class when you need to programmatically manage a Rabbit MQ cluster. Pika gives you the ability to create exchanges and queues and bind them together using the AMQP protocol but there is no way to create a new virtual host. Despite that, running tests on randomly unique virtual hosts is quite convenient.

This class uses Rabbit's HTTP API to manipulate the cluster. It exposes the management functions that have proven useful for writing tests against a shared RabbitMQ cluster.

Usage Example

```
from test_helpers import rabbit

_fixture = rabbit.RabbitMqFixture('localhost', 'guest', 'guest')

def setup_module():
    _fixture.install_virtual_host()
    _fixture.create_binding('accounts', 'my_queue', 'status.added')

    # from this point on, os.environ['AMQP'] points to a
    # newly created, isolated virtual host that will be
    # removed when the test is complete
```

create_binding(*exchange_name*, *queue_name*, *routing_key*)
Create a message binding.

Parameters

- **exchange_name** (*str*) – name of the exchange to create/update
- **queue_name** (*str*) – name of the queue to create/update
- **routing_key** (*str*) – routing key that binds the exchange and the queue

This method creates the specified queue and exchange if they do not already exist and then binds *routing_key* such that messages with it are routed through the exchange and queue.

host
The URL-quoted rabbit server.

install_virtual_host()
Create a new virtual host.

Returns the name of the virtual host

The virtual host will be created and the current user will be granted full permission to it.

This method also sets the AMQP environment variable to the appropriate URL for connecting to the virtual host.

password
The URL-quoted rabbit password.

purge_queue(*queue_name*)
Purge a Rabbit MQ queue.

remove_virtual_host()
Remove the generated virtual host.

user
The URL-quoted rabbit user name.

virtual_host
The URL-quoted virtual host ready to use as-is.

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

- 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.

Python Module Index

t

Index

P

R

S

T

U

V