txfixtures Documentation

Release 0.2.2.dev1

Free Ekanayaka

December 22, 2016

Contents

1 Run asynchronous code from test cases 3

2 Spawn, control and monitor test services 5

3 Setup a phantomjs Selenium driver 7

4 API documentation 9

5 Indices and tables 13

Python Module Index 15

txfixtures hooks into the testtools test fixture interface, so that you can write tests that rely on having an external Twisted daemon.

See:

- https://launchpad.net/txfixtures
- https://launchpad.net/testtools

Contents:

CHAPTER 1

Run asynchronous code from test cases

The *Reactor* fixture can be used to drive asynchronous Twisted code from a regular synchronous Python TestCase.

The approach differs from trial or testtools twisted support: instead of starting the reactor in the main thread and letting it spin for a while waiting for the Deferred returned by the test to fire, this fixture will keep the reactor running in a background thread until cleanup.

When used with testresources's FixtureResource and OptimisingTestSuite, this fixture makes it possible to have full control and monitoring over long-running processes that should be up for the whole test suite run, and maybe produce output useful for the test itself.

The typical use case is integration testing.

```
>>> from testtools import TestCase

>>> from twisted.internet import reactor
>>> from twisted.internet.threads import blockingCallFromThread
>>> from twisted.internet.utils import getProcessOutput

>>> from txfixtures import Reactor

>>> class TestUsingAsyncAPIs(TestCase):
...
...     def setUp(self):
...         super().setUp()
...         self.useFixture(Reactor())
...
...     def test_uptime(self):
...         out = blockingCallFromThread(reactor, getProcessOutput, b"uptime")
...         self.assertIn("load average", out.decode("utf-8"))
...
>>> test = TestUsingAsyncAPIs(methodName="test_uptime")
>>> test.run().wasSuccessful()
True
```

CHAPTER 2

Spawn, control and monitor test services

The *Service* fixture can be used to spawn a background service process (for instance a web application), and leave it running for the duration of the test suite (see testresources-integration).

It supports real-time streaming of the service standard output to Python's `logging` system.

2.1 Spawn a simple service fixture listening to a port

Let's create a test that spawns a dummy HTTP server that listens to port 8080:

```
>>> import socket

>>> from testtools import TestCase
>>> from txfixtures import Reactor, Service

>>> HTTP_SERVER = "python3 -m http.server 8080".split(" ")

>>> class HTTPServerTest(TestCase):
...
...     def setUp(self):
...         super().setUp()
...         self.useFixture(Reactor())
...
...         # Create a service fixture that will spawn the HTTP server
...         # and wait for it to listen to port 8080.
...         self.service = Service(HTTP_SERVER)
...         self.service.expectPort(8080)
...
...         self.useFixture(self.service)
...
...     def test_connect(self):
...         connection = socket.socket()
...         connection.connect(("127.0.0.1", 8080))
...         self.assertEqual(connection.getsockname()[0], "127.0.0.1")

>>> test = HTTPServerTest(methodName="test_connect")
>>> test.run().wasSuccessful()
True
```

2.2 Forward standard output to the Python logging system

Let's spawn a simple HTTP server and have its standard output forwarded to the Python logging system:

```
>>> import requests

>>> from fixtures import FakeLogger

>>> TWIST_COMMAND = "twistd -n web".split(" ")

# This format string will be used to build a regular expression to parse
# each output line of the service, and map it to a Python LogRecord. A
# sample output line from the twistd web command looks like:
#
#   2016-11-17T22:18:36+0000 [-] Site starting on 8080
#
>>> TWIST_FORMAT = "{Y}-{m}-{d}T{H}:{M}:{S}\+0000 \[{name}\] {message}"

# This output string will be used as a "marker" indicating that the service
# has initialized, and should shortly start listening to the expected port (if
# one was given). The fixture.setUp() method will intercept this marker and
# then wait for the service to actually open the port.
>>> TWIST_OUTPUT = "Site starting on 8080"

>>> class TwistedWebTest(TestCase):
...
...     def setUp(self):
...         super().setUp()
...         self.logger = self.useFixture(FakeLogger())
...         self.useFixture(Reactor())
...         self.service = Service(TWIST_COMMAND)
...         self.service.setOutputFormat(TWIST_FORMAT)
...         self.service.expectOutput(TWIST_OUTPUT)
...         self.service.expectPort(8080)
...         self.useFixture(self.service)
...
...     def test_request(self):
...         response = requests.get("http://localhost:8080")
...         self.assertEqual(200, response.status_code)
...         self.assertIn('"GET / HTTP/1.1" 200', self.logger.output)
...
>>> test = TwistedWebTest(methodName="test_request")
>>> test.run().wasSuccessful()
True
```

CHAPTER 3

Setup a phantomjs Selenium driver

The `PhantomJS` fixture starts a phantomjs service in the background and exposes it via its `webdriver` attribute, which can then be used by test cases for Selenium-based assertions:

```
>>> from fixtures import FakeLogger
>>> from testtools import TestCase
>>> from txfixtures import Reactor, Service, PhantomJS

>>> TWIST_COMMAND = "twistd -n web".split(" ")

>>> class HTTPServerTest(TestCase):
...
...     def setUp(self):
...         super().setUp()
...         self.logger = self.useFixture(FakeLogger())
...         self.useFixture(Reactor())
...
...         # Create a sample web server
...         self.service = Service(TWIST_COMMAND)
...         self.service.expectPort(8080)
...         self.useFixture(self.service)
...
...         self.phantomjs = self.useFixture(PhantomJS())
...
...     def test_home_page(self):
...         self.phantomjs.webdriver.get("http://localhost:8080")
...         self.assertEqual("Twisted Web Demo", self.phantomjs.webdriver.title)

>>> test = HTTPServerTest(methodName="test_home_page")
>>> test.run().wasSuccessful()
True
```

CHAPTER 4

API documentation

Generated reference documentation for all the public functionality of txfixtures.

4.1 txfixtures.reactor

class txfixtures.reactor.**Reactor**(*reactor=None*, *timeout=5*)
A fixture to run the Twisted reactor in a separate thread.

This fixture will spawn a new thread in the test process and run the Twisted reactor in it. Test code can then invoke asynchronous APIs by using blockingCallFromThread().

Parameters

- **reactor** – The Twisted reactor to run.
- **timeout** – Raise an exception if the reactor or the thread is it runs in doesn't start (at setUp time) or doesn't stop (at cleanUp time) in this amount of seconds.

Variables **thread** – The Thread that the reactor runs in.

call(*timeout*, *f*, **a*, ***kw*)
Convenience around blockingCallFromThread, with timeout support.

The function f will be invoked in the reactor's thread with the given arguments and keyword arguments. If f returns a Deferred, the calling code will block until it has fired.

Returns The value returned by f or the value fired by the Deferred it returned. If f traces back or the Deferred it returned errbacks, the relevant exception will be propagated to the caller of this method.

Raises **CallFromThreadTimeout** – If timeout seconds have elapsed and the Deferred returned by f hasn't fired yet.

reset()
Make sure that the reactor is still running.

If the reactor and its thread have died, this method will try to recover them by creating a new thread and starting the reactor again.

4.2 txfixtures.service

class txfixtures.service.**Service**(*command*, *reactor=None*, *timeout=15*, *env=None*)
Spawn, control and monitor a background service.

allocatePort()
Allocate an unused port.

This method can be used by subclasses to allocate a random ports for the service they spawn.

There is a small race condition here (between the time we allocate the port, and the time it actually gets used), but for the purposes for which this method gets used it isn't a problem in practice.

class txfixtures.service.**ServiceOutputParser**(*service*, *logger=None*, *pattern=None*)
Parse the standard output stream of a service and forward it to the Python logging system.

The stream is assumed to be a UTF-8 sequence of lines each delimited by a (configurable) delimiter character.

Each received line is tested against the RegEx pattern provided in the constructor. If a match is found, a LogRecord is built using the information from the groups of the match object, otherwise default values will be used.

The record is then passed to the Logger provided in the constructor.

Match objects that result from the RegEx pattern are supposed to provide groups named after the substitutions below.

Parameters **service** – A string identifying the service whose output is being parsed. It will be attached as 'service' attribute to all log records emitted.

delimiter = '\n'
The delimiter character identifying the end of a line.

lineLengthExceeded(*line*)
Simply truncate the line.

lineReceived(*line*)
Foward the received line to the Python logging system.

substitutions = {'M': '(?P<M>\\d{2})', 'S': '(?P<S>\\d{2})', 'd': '(?P<d>\\d{2})', 'Y': '(?P<Y>\\d{4})', 'H': '(?P<H>
Substitutions for commonly used groups in line match patterns. For example, this allows you to use "{Y}-{m}-{S}" as pattern snippet, as

whenLineContains(*text*, *callback*)
Fire the given callback when a line contains the given text.

The callback will be fired only once when and if a match is found.

class txfixtures.service.**ServiceProtocol**(*reactor=None*, *parser=None*, *timeout=15*)
Start and stop a background service process.

This ProcessProtocol manages the start up and termination phases of a background service process. The process is considered 'running' when it has stayed up for at least 0.1 seconds (or any other non default value which *minUptime* is set too), and optionally when it has emitted a certain string and/or it has started listening to a certain port.

expectedOutput = None
Optional text that we expect the process to emit in standard output before we consider it ready.

expectedPort = None
Optional port number that we expect the service process to listen, before we consider it ready.

minUptime = 0.1
The service process must stay up at least this amount of seconds, before it's considered running. This allows to catch common issues like the service process executable not being in PATH or not being executable.

ready = None
Deferred that will fire when the service is considered ready, i.e. it has stayed up for at least minUptime

seconds, has produced the expected output (if any), and is listening to the expected port (if any). Upon cancellation, any waiting activity will be stopped.

`terminated` = None
Deferred that will fire when the service has fully terminated, i.e. it has exited and we parent process have read any outstanding data in the pipes and have closed them.

`timeout` = None
Maximum amount of seconds to wait for the service to be ready. After that, the 'ready' deferred will errback with a TimeoutError.

4.3 txfixtures.phantomjs

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

t

txfixtures.reactor, 9
txfixtures.service, 9

A

allocatePort() (txfixtures.service.Service method), 9

C

call() (txfixtures.reactor.Reactor method), 9

D

delimiter (txfixtures.service.ServiceOutputParser attribute), 10

E

expectedOutput (txfixtures.service.ServiceProtocol attribute), 10
expectedPort (txfixtures.service.ServiceProtocol attribute), 10

L

lineLengthExceeded() (txfixtures.service.ServiceOutputParser method), 10
lineReceived() (txfixtures.service.ServiceOutputParser method), 10

M

minUptime (txfixtures.service.ServiceProtocol attribute), 10

R

Reactor (class in txfixtures.reactor), 9
ready (txfixtures.service.ServiceProtocol attribute), 10
reset() (txfixtures.reactor.Reactor method), 9

S

Service (class in txfixtures.service), 9
ServiceOutputParser (class in txfixtures.service), 10
ServiceProtocol (class in txfixtures.service), 10
substitutions (txfixtures.service.ServiceOutputParser attribute), 10

T

terminated (txfixtures.service.ServiceProtocol attribute), 11
timeout (txfixtures.service.ServiceProtocol attribute), 11
txfixtures.reactor (module), 9
txfixtures.service (module), 9

W

whenLineContains() (txfixtures.service.ServiceOutputParser method), 10