Glusto Documentation

Release 0.1.0

Jonathan Holloway

Sep 17, 2018

Contents

Glusto is a framework designed to provide features commonly used in a remote/distributed environment via a single and easy-to-access object.

It started out as a port of some shell ssh functions I had written and was meant for use in PyUnit* tests and config scripts for Gluster.

I've removed the Gluster specifics from this package. Feel free to give it a go, and please let me know how it works out.

Some of the key concepts and features of Glusto:

- Glusto inherits from multiple classes providing configuration (yaml, json, ini), remote connection (SSH, SCP, RPyC), ANSI color output, logging, and unit test functionality (PyUnit, PyTest, Nose)–presenting them in a single global Class object.
- Glusto also acts as a global class for maintaining state and configuration data across multiple modules and classes.
- Glusto provides a wrapper utility (`/usr/bin/glusto`) to help make configuration files available to test cases from the command-line.

Adding Glusto utilities to a Python module is as simple as an import.

Example: To use Glusto in a module:

```
from glusto.core import Glusto as g
```

Note: It is no longer necessary to say "Glusto Importo!" out loud before executing scripts using the Glusto module. The import statement is more than sufficient.

CHAPTER 1

User Guide

1.1 Table of Contents

1.1.1 Installing Glusto

There is more than one way to install Glusto.

- Installing the package directly from the github repo via the `pip` command.
- Installing a docker image from Docker Hub.
- Cloning from github and installing via setuptools.

Installing Glusto Directly from Git via Pip

The `pip` command can install directly from the Glusto project repo on github.com.

```
# pip install --upgrade git+git://github.com/loadtheaccumulator/glusto.git
```

Uninstalling

What?! But, why?!

To uninstall glusto, use the `pip` command.

```
# pip uninstall glusto
```

Using Glusto via Docker

A minimal Docker image is available to download and experiment with Glusto.

Note: The image has currently been tested on Fedora 23 and Mac OS X (El Capitan) running *Docker for Mac* without issues.

To use the Glusto Docker image, pull the image from Docker Hub and go.

```
docker pull loadtheaccumulator/glusto
docker run -it --rm loadtheaccumulator/glusto /bin/bash
```

This takes you into the running container as root. Please reference the documentation on docker.com (or available all over the web now) for more information on using Docker.

Note: You will need to pay particular attention to keys and configs when using the docker image. It might be useful to create a Dockerfile to build a new image, based on the Glusto image, that makes your own custom config, keys, and tests available. The Dockerfile used to create the Glusto image is available in the GitHub repo, so you can also just roll your own on the distro image of your choice. More on Docker later, but for now... experiment.

Cloning the Glusto Github Repo

On the system where Glusto is to be installed, clone the repo...

```
# git clone https://github.com/loadtheaccumulator/glusto.git
```

Installing Glusto from a Git Clone

To install the Glusto package via setuptools.

1. Change directory into the glusto directory.

```
# cd glusto
```

2. Run the setuptools script.

```
# python setup.py
```

1.1.2 Configuring Glusto

Glusto currently reads configuration files in yaml, json, or ini format. It looks in `/etc/glusto` for `defaults.yml`, `defaults.yaml`, `defaults.json` and `defaults.ini`. You can provide any or all at the same time.

Note: It is currently necessary to create the `/etc/glusto` directory manually and populate it with `defaults.` files. Automatic creation of the defaults directory, a default `defaults.yml`, and sample configs is upcoming.

defaults.yml or defaults.yaml:

```
keyfile: "~/ssh/id_rsa"
use_ssh: True
use_controlpersist: True
log_color: True
```

defaults.ini:

```
[defaults]
this = yada1
that = yada2
the_other = %(this)s and %(that)s

[globals]
some_default = yada yada
```

defaults.json:

```
{"things": {
  "thing_one": "yada",
  "thing_two": "yada yada",
  "thing_three": {
    "combo_thing": [
      {"combo_thing_one": "yada", "combo_thing_two": "yada yada"}
    ]
  }
}}
```

The ini format provides some simple variable capability.

For example, this line from the above defaults.ini config:

```
the_other = %(this)s and %(that)s
```

. . . will populate the_other variable in your Python script as "yada1 and yada2":

```
defaults: {that: yada2, this: yada1, this_and_that: yada1 and yada2}
```

Note: It is also possible to pass additional configuration files at the command-line, in IDLE, or from within script. via the `-c` option. See Using Config Files with Glusto and Using the Glusto CLI Utility for more information.

1.1.3 Using Glusto

Using Glusto in a Module

To use Glusto in a module, import the Glusto class at the top of each module leveraging the glusto tools.

```
from glusto.core import Glusto as g
```

This provides access to all of the functionality of Glusto via the `g` object created by the import statement.

Using Glusto via the Python Interactive Interpreter

One of the primary objectives of Glusto is to maintain feature support from the Python Interactive Interpreter. Most, if not all, features should be easily referenced via the interpreter to ease use and reduce time during development.

To use Glusto via the Python Interactive Interpreter, enter the interpreter via the `python` command.

```
$ python
>>> from glusto.core import Glusto as g
```

This provides access to all of the functionality of Glusto via the `g.` object created by the import statement–in the same way as imported in a script.

For example:

```
$ python
>>> from glusto.core import Glusto as g

>>> g.run_local('uname -a')
(0, 'Linux mylaptop 4.4.9-300.fc23.x86_64 #1 SMP Wed May 4 23:56:27 UTC 2016␣
↪x86_64 x86_64 x86_64 GNU/Linux\n', '')

>>> config = g.load_config('examples/systems.yml')
>>> config
{'nodes': ['192.168.1.221', '192.168.1.222', '192.168.1.223', '192.168.1.224
↪'], 'clients': ['192.168.1.225'], 'masternode': '192.168.1.221'}

>>> g.run(config['nodes'][0], 'uname -a')
(0, 'Linux rhserver1 2.6.32-431.29.2.el6.x86_64 #1 SMP Sun Jul 27 15:55:46␣
↪EDT 2014 x86_64 x86_64 x86_64 GNU/Linux\n', '')
>>> g.list_ssh_connections()
root@192.168.1.221

>>> g.run_serial(config['nodes'], 'hostname')
{'192.168.1.224': (0, 'rhserver4\n', ''), \
'192.168.1.221': (0, 'rhserver1\n', ''), \
'192.168.1.223': (0, 'rhserver3\n', ''), \
'192.168.1.222': (0, 'rhserver2\n', '')}
>>> g.list_ssh_connections()
root@192.168.1.222
root@192.168.1.223
root@192.168.1.221
root@192.168.1.224
```

Typing `from glusto.core import Glusto as g` each time you start the interpreter can become tedious. To automatically run commands, the PYTHONSTARTUP environment variable can be pointed to a python script containing common setup commands.

```
$ cat examples/pythonstartup_script.py
from glusto.core import Glusto as g

print "Python startup starting"
config = g.load_config('examples/systems.yml')
rcode, rout, rerr = g.run(config['nodes'][0], 'uname -a')
print ('The uname info is: %s' % rout)
print "Python startup complete"
```

```
$ export PYTHONSTARTUP=examples/pythonstartup_script.py

$ python
Python 2.7.11 (default, Mar 31 2016, 20:46:51)
[GCC 5.3.1 20151207 (Red Hat 5.3.1-2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Python startup starting
Python startup complete
>>> g
<class 'glusto.core.Glusto'>
>>> config
```

(continues on next page)

(continued from previous page)

```
{'nodes': ['192.168.1.221', '192.168.1.222', '192.168.1.223', '192.168.1.224
↪'], 'clients': ['192.168.1.225'], 'masternode': '192.168.1.221'}
>>> uname_info
'Linux rhserver1 2.6.32-431.29.2.el6.x86_64 #1 SMP Sun Jul 27 15:55:46 EDT␣
↪2014 x86_64 x86_64 x86_64 GNU/Linux\n'
```

Using the Glusto CLI Utility

Glusto provides a wrapper utility for features like unit test support, etc. Currently, the Glusto CLI allows leveraging the PyUnit, PyTest, and Nose module features in an easily configurable and callable wrapper.

To see the options available, use the `--help` option.

```
$ /usr/bin/glusto --help
Starting glusto via main()
usage: glusto [-h] [-c CONFIG_LIST] [-u] [-d DISCOVER_DIR]

Glusto CLI wrapper

optional arguments:
  -h, --help            show this help message and exit
  -c CONFIG_LIST, --config CONFIG_LIST
                        Config file(s) to read.
  -u, --unittest        Run unittests per provided config file.
  -d DISCOVER_DIR, --discover DISCOVER_DIR
                        Discover unittests from directory
  -t RUN_PYTEST, --pytest RUN_PYTEST
                        Run tests using the pytest framework
  -n RUN_NOSETESTS, --nosetests RUN_NOSETESTS
                        Run tests using the nose framework
```

By default, the `glusto` command will read the default config files in the `/etc/glusto/` directory

For example, this run of the command reads the `defaults.yml` and `defaults.ini` files in `/etc/glusto/`:

```
$ glusto
Starting glusto via main()
defaults: {that: yada2, the_other: yada1 and yada2, this: yada1}
globals: {some_default: yada yada}
keyfile: ~/ssh/id_rsa
log_color: true
that: yada2
the_other: yada1 and yada2
this: yada1
use_controlpersist: true
use_ssh: true
Ending glusto via main()
```

Options for Running Unit Tests

To run unit tests via the Glusto CLI Utility, see the examples and links to additional documentation below.

Running PyUnit Tests

Example:

```
$ glusto -c 'examples/systems.yml' -u -d 'tests'
$ glusto -c 'examples/unittests/unittest.yml examples/unittests/unittest_
↪list.yml examples/systems.yml' -u
```

For more information on working with unit tests, see Unittests and Glusto

Running PyTest Tests

Example:

```
$ glusto -c 'examples/systems.yml' --pytest='-v -x tests -m response'
```

For more information on working with unit tests, see PyTest and Glusto

Running Nose Tests

Example:

```
$ glusto -c 'examples/systems.yml' --nosetests='-v -w tests'
```

For more information on working with unit tests, see Nose and Glusto

Running Different Frameworks in a Single Run

Not that the need would arise, but the capability to run all three in a single command is there.

Example running tests with `--pytest=` and `nosetests=` options:

```
$ glusto -c 'examples/systems.yml examples/unittests/unittest.yml' -u --
↪nosetests='-w tests' --pytest='-x tests -m response'
Starting glusto via main()
clients: [192.168.1.225]
masternode: 192.168.1.221
nodes: [192.168.1.221, 192.168.1.222, 192.168.1.223, 192.168.1.224]
unittest:
  load_tests_from_module: {module_name: tests.test_glusto, use_load_tests:␣
↪true}
  output_junit: false

clients: [192.168.1.225]
masternode: 192.168.1.221
nodes: [192.168.1.221, 192.168.1.222, 192.168.1.223, 192.168.1.224]
unittest:
  load_tests_from_module: {module_name: tests.test_glusto, use_load_tests:␣
↪true}
  output_junit: false

PREFIX: tests.test_glusto.TestGlustoBasics
Setting Up Class: TestGlustoBasics
```

(continues on next page)

(continued from previous page)

```
test_return_code (tests.test_glusto.TestGlustoBasics)
Testing the return code ... Setting Up: tests.test_glusto.TestGlustoBasics.
↪test_return_code
Running: tests.test_glusto.TestGlustoBasics.test_return_code - Testing the␣
↪return code
Tearing Down: tests.test_glusto.TestGlustoBasics.test_return_code
ok
test_stdout (tests.test_glusto.TestGlustoBasics)
Testing output to stdout ... Setting Up: tests.test_glusto.TestGlustoBasics.
↪test_stdout
Running: tests.test_glusto.TestGlustoBasics.test_stdout - Testing output to␣
↪stdout
Tearing Down: tests.test_glusto.TestGlustoBasics.test_stdout
Cleaning up after setup on fail or after teardown
ok
test_stderr (tests.test_glusto.TestGlustoBasics)
Testing output to stderr ... Setting Up: tests.test_glusto.TestGlustoBasics.
↪test_stderr
Running: tests.test_glusto.TestGlustoBasics.test_stderr - Testing output to␣
↪stderr
Tearing Down: tests.test_glusto.TestGlustoBasics.test_stderr
ok
test_expected_fail (tests.test_glusto.TestGlustoBasics)
Testing an expected failure. This test should fail ... Setting Up: tests.
↪test_glusto.TestGlustoBasics.test_expected_fail
Running: tests.test_glusto.TestGlustoBasics.test_expected_fail - Testing an␣
↪expected failure. This test should fail
expected failure
Tearing Down: tests.test_glusto.TestGlustoBasics.test_expected_fail
test_negative_test (tests.test_glusto.TestGlustoBasics)
Testing an expected failure as negative test ... Setting Up: tests.test_
↪glusto.TestGlustoBasics.test_negative_test
Running: tests.test_glusto.TestGlustoBasics.test_negative_test - Testing an␣
↪expected failure as negative test
Tearing Down: tests.test_glusto.TestGlustoBasics.test_negative_test
ok
test_skip_me (tests.test_glusto.TestGlustoBasics)
Testing the unittest skip feature ... skipped 'Example test skip'
Tearing Down Class: TestGlustoBasics

----------------------------------------------------------------------
Ran 6 tests in 0.585s

OK (skipped=1, expected failures=1)
pytest: -x tests -m response
============================================================================␣
↪test session starts␣
↪============================================================================
platform linux2 -- Python 2.7.11, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: glusto, inifile:
collected 21 items

tests/test_glusto_pytest.py ...

=============================================================================␣
↪18 tests deselected by "-m 'response'"␣
↪=============================================================================
```

(continues on next page)

(continued from previous page)

```
====================================================================== 3␣
↪passed, 18 deselected in 0.32 seconds␣
↪======================================================================
nosetests: -w tests
/usr/lib64/python2.7/unittest/case.py:378: RuntimeWarning: TestResult has no␣
↪addExpectedFailure method, reporting as passes
  RuntimeWarning)
...S..E...F.....E......
======================================================================
ERROR: Load tests in a specific order.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/usr/lib/python2.7/site-packages/nose/case.py", line 197, in runTest
    self.test(*self.arg)
TypeError: load_tests() takes exactly 3 arguments (0 given)

======================================================================
ERROR: Load tests in a specific order.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/usr/lib/python2.7/site-packages/nose/case.py", line 197, in runTest
    self.test(*self.arg)
TypeError: load_tests() takes exactly 3 arguments (0 given)

======================================================================
FAIL: Testing an expected failure. This test should fail
----------------------------------------------------------------------
Traceback (most recent call last):
  File "glusto/tests/test_glusto_pytest.py", line 98, in test_expected_fail
    self.assertEqual(rcode, 0)
AssertionError: 1 != 0
-------------------- >> begin captured stdout << ---------------------
Setting Up: tests.test_glusto_pytest.TestGlustoBasicsPyTest.test_expected_
↪fail
Running: tests.test_glusto_pytest.TestGlustoBasicsPyTest.test_expected_fail -
↪ Testing an expected failure. This test should fail

--------------------- >> end captured stdout << ----------------------
-------------------- >> begin captured logging << --------------------
plumbum.local: DEBUG: Running ['/usr/bin/ssh', '-T', '-
↪oPasswordAuthentication=no', '-oStrictHostKeyChecking=no', '-oPort=22', '-
↪oConnectTimeout=10', '-oControlMaster=auto', '-oControlPersist=4h', '-
↪oControlPath=~/.ssh/glusto-ssh-%r@%h:%p', 'root@192.168.1.221', 'cd', '/
↪root', '&&', 'false']
-------------------- >> end captured logging << --------------------

----------------------------------------------------------------------
Ran 23 tests in 1.964s

FAILED (SKIP=1, errors=2, failures=1)
Ending glusto via main()
```

Note: I'll be able to demonstrate this better when I have PyTest example test scripts written. The above command runs the same PyUnit-based test scripts against the PyUnit, PyTest and Nose frameworks.

1.1.4 Working With Remote Systems

Glusto provides functions for running commands on remote systems, as well as sending and retrieving files.

Passwordless SSH with Keys

Glusto relies on existing SSH keys. Please consult the docs for your specific platform for more information on how to setup passwordless ssh.

Configuring Glusto to Use Specific Keys

Add a specific SSH key to the /etc/glusto defaults configs...

For example, add the following line to `/etc/glusto/defaults.yml`:

```
keyfile: "~/ssh/id_rsa"
```

Run a Single Command via SSH

To run a command on a remote system via SSH, use the "run" command:

```
>>> g.run('server01.example.com', 'uname -a')
(0, 'Linux server01 2.6.32-431.29.2.el6.x86_64 #1 SMP Sun Jul 27 15:55:46 EDT 2014␣
↪x86_64 x86_64 x86_64 GNU/Linux\n', '')
```

It is also easy enough to assign the return code, stdout, and stderr to variables:

```
>>> retcode, stdout, stderr = g.run('server01.example.com', 'uname -a')
>>> retcode
0
>>> stdout
'Linux server01 2.6.32-431.29.2.el6.x86_64 #1 SMP Sun Jul 27 15:55:46 EDT 2014 x86_64␣
↪x86_64 x86_64 GNU/Linux\n'
>>> stderr
''
```

Run a Single Command on the Localhost

A command can be run on the localhost via SSH by simply passing 'localhost' as the hostname. As a convenience, and to save some overhead, Glusto provides a method to run a command locally and get the return code, stdout, and stderr like the remote run command.

To run a command on the local system:

```
>>> g.run_local('uname -a')
(0, 'Linux localhost 4.4.9-300.fc23.x86_64 #1 SMP Wed May 4 23:56:27 UTC 2016 x86_64␣
↪x86_64 x86_64 GNU/Linux\n', '')
```

Run a Command on More than One Host

Glusto provides convenience methods to run commands against multiple hosts.

Run a Command Serially on Multiple Servers

To run a command against a list of hosts, use the `run_serial()` method. The command will be run against the hosts one after another.

```
>>> hosts = ["breedshill.example.com", "bunkerhill.example.com"]
>>> results = g.run_serial(hosts, 'uname -a')
```

Run a Command in Parallel

To run a command against a list of hosts in parallel, use the `run_parallel()` method. The command will be run against the hosts at the same time.

```
>>> command = "uname -a"
>>> results = g.run_parallel(hosts, 'uname -a')
    {'192.168.1.221':
             (0, 'Linux rhserver1 2.6.32-431.29.2.el6.x86_64 #1 SMP Sun Jul␣
↪27 15:55:46 EDT 2014 x86_64 x86_64 x86_64 GNU/Linux\n', ''),
    '192.168.1.222':
             (0, 'Linux rhserver2 2.6.32-431.29.2.el6.x86_64 #1 SMP Sun Jul␣
↪27 15:55:46 EDT 2014 x86_64 x86_64 x86_64 GNU/Linux\n', '')}
```

Run a Command Asynchronously

The `run_parallel` method is a convenience method and runs the same command against a list of systems using the same user. It is possible to use the underlying `run_async` command directly to run a variety of combinations asynchronously.

An example of how `run_parallel` uses `run_async`:

```
>>> command = "uname -a"
>>> proc1 = g.run_async("bunkerhill", command)
>>> proc2 = g.run_async("breedshill", command)

>>> results1 = proc1.async_communicate()
>>> results2 = proc2.async_communicate()
```

To asynchronously run the same command against the same server as a different user:

```
>>> command = "uname -a; echo $USER"
>>> proc1 = g.run_async("breedshill", command, user="howe")
>>> proc2 = g.run_async("breedshill", command, user="pigot")

>>> results1 = proc1.async_communicate()
>>> results2 = proc2.async_communicate()
```

Note: run_async() runs commands asynchronously, but blocks on async_communicate() and reads output sequentially. This might not be a good fit for run-and-forget commands.

Transferring Files To and From Remote Systems

Glusto provides methods to call SShMachine's upload and download commands, as well as a method to transfer a file directly between remote systems.

Uploading a File

To upload a file to a remote system, use the `upload()` method.

```
>>> g.upload('server01.example.com', '/etc/localfile.txt', '/tmp/localfile_
↪remotecopy.txt')
```

Downloading a File

To download a file from a remote system, use the `download()` method.

```
>>> g.download('server01.examples.com', '/etc/remotefile.txt', '/tmp/
↪remotefile_localcopy.txt')
```

Transferring a File from Remote to Remote

To transfer a file directly from a remote system to another remote system, without having to first download to the local system and then upload to the remote, use the `transfer` method.

```
>>> g.transfer('server01.example.com', '/etc/remote1file.txt', 'server02.
↪example.com', '/tmp/remote1file_remote2copy.txt')
```

Listing SSH Connections

To see a list of the current SSH connections, use the `ssh_list_connections()` method.

```
>>> g.ssh_list_connections()
root@192.168.1.222
root@192.168.1.223
root@192.168.1.221
root@192.168.1.224
```

Closing Connections

It is typically not necessary to close a connection. Connections are cached for quick re-use and SSH connections should close at program exit. Should the need arise...

Closing a Connection

To close a connection use the `ssh_close_connection()` method.

```
>>> g.ssh_close_connection('192.168.1.221')
>>> g.ssh_close_connection('192.168.1.221', user='george')
```

Close All Connections

To close all connections use the `ssh_close_connections()` method.

```
>>> g.ssh_close_connections()
```

1.1.5 Using RPyC

Let's start with this…

Warning: Per the install documentation for RPyC, it is not possible to connect to a Python 3.x remote from a Python 2.x system and vice-versa.

See the RPyC Install documentation[1] for more information. That's not necessarily a show-stopper for everyone, but certainly worth consideration depending on your environment.

Passwordless Connections

Glusto's implementation of RPyC leverages the same SSH connections described in the previous section. See Passwordless SSH with Keys for more information on configuring Glusto for specific SSH keys.

Setting up Connections

Unlike the SSH connections that are created automatically when you use `run()` or the other SSH methods, the RPyC connection needs to be created before it can be used. After the connection is made, it is cached for use by subsequent RPyC calls.

Setting up a Single Connection

To setup an RPyC connection to a remote server, use the `rpyc_get_connection()` method.

```
>>> g.rpyc_get_connection('192.168.1.221')
```

Listing Connections

To see the list of connections, use the `rpyc_list_connections()` method.

```
>>> g.rpyc_list_connections()
root@192.168.1.221:1
```

Setting up a Connection with a Specific User

The default user is root. To setup a connection with a user other than the default, add the `user` parameter.

[1] https://rpyc.readthedocs.io/en/latest/install.html#cross-interpreter-compatibility

```
>>> g.rpyc_get_connection('192.168.1.221', user='george')
>>> g.rpyc_list_connections()
george@192.168.1.221:1
```

Setting up Multiple Connections with Different Users

Sometimes it is necessary to run commands as different users at the same time. With RPyC, it is possible to setup multiple connections to the same server with different users.

```
>>> g.rpyc_get_connection('192.168.1.221', user='george')
>>> g.rpyc_get_connection('192.168.1.221', user='alexander')

>>> g.rpyc_list_connections()
alexander@192.168.1.221:1
george@192.168.1.221:1
```

On the remote server, multiple instances of the rpyc server are run:

```
$ ps -ef | grep deployed
george    7504  5456  0 18:13 ?        00:00:00 bash -c cd /home/george && /
↪usr/bin/python2 /tmp/tmp.XuDwqQkXVq/deployed-rpyc.py
george    7511  7504  1 18:13 ?        00:00:00 /usr/bin/python2 /tmp/tmp.
↪XuDwqQkXVq/deployed-rpyc.py
root      7579  3041  0 18:13 ?        00:00:00 bash -c cd /root && /usr/bin/
↪python2 /tmp/tmp.xAfvVdtmjg/deployed-rpyc.py
root      7582  7579  4 18:13 ?        00:00:00 /usr/bin/python2 /tmp/tmp.
↪xAfvVdtmjg/deployed-rpyc.py
```

Setting up Multiple Connections with the Same User

There are also times when it is helpful to be able to run commands at the same time, but as the same user. For example, to run a long running command while checking another command running at the same time. With RPyC, it is possible to setup multiple connections to the same server with the the same user.

To setup a connection to the same server as the same user, use the `instance` parameter to specificy an instance number.

```
>>> g.rpyc_get_connection('192.168.1.221')
>>> g.rpyc_get_connection('192.168.1.221', instance=2)
>>> g.rpyc_get_connection('192.168.1.221', user='george')
>>> g.rpyc_get_connection('192.168.1.221', user='george', instance=2)

>>> g.rpyc_list_connections()
george@192.168.1.221:2
root@192.168.1.221:2
root@192.168.1.221:1
george@192.168.1.221:1
```

Note: Glusto doesn't automatically increment the instance number. Specifying the same instance number will return the cached connection and not a new instance.

Making RPyC Calls

Note: Rather than cover RPyC in-depth here, below are some examples of using RPyC with Glusto. Please refer to the RPyC documentation[2] for more information.

Using the Connection

Once an RPyC connection is made, it can be referenced to make RPyC calls against the remote system.

```
conn1 = g.rpyc_get_connection('192.168.1.221')
>>> conn1.modules.sys.platform
'linux2'
```

Asynchronous RPyC Calls

RPyC provides an asynchronous mechanism to allow for running remote calls in the background.

Backgrounding an RPyC Call

Sometimes you just want to kick off a process and let it run without needing to wait for it to finish or caring about the result.

To run a command in the background without waiting for a result.

```
>>> import rpyc
>>> conn1 = g.rpyc_get_connection('192.168.1.221')
>>> async_sleep1 = rpyc.async(conn1.modules.time.sleep)
>>> async_sleep1(10)
<AsyncResult object (pending) at 0x7f3382bc6f50>
```

Waiting for a Backgrounded Call

Other times you want to wait for the processes to finish before continuing.

To wait for a backgrounded process, use the rpyc `wait()` method.

```
>>> import rpyc
>>> conn1 = g.rpyc_get_connection('192.168.1.221')
>>> async_sleep1 = rpyc.async(conn1.modules.time.sleep)
>>> res1 = async_sleep1(10)
<AsyncResult object (pending) at 0x7f3382bc6530>
>>> res1.wait()
```

Running a Second Call Against the Same System

When it is necessary to run a background command against a system and run another command against the same system, you can use `wait()` to wait for a return for each call made.

[2] http://rpyc.readthedocs.io/en/latest/index.html

```
>>> res1 = async_sleep(60)
>>> res2 = async_sleep(10)
>>> res2.wait()
>>> res1.wait()
```

Note: Because the backgrounded calls are made against the same connection, the first call blocks the connection until complete. In the above example, the `res2.wait()` will block for 70 seconds. The `res1.wait()` returns instantly.

To run multiple background calls against the same system, you can create a second connection and run the second background call against it.

```
>>> import rpyc

>>> conn1 = g.rpyc_get_connection('192.168.1.221')
>>> async_sleep1 = rpyc.async(conn1.modules.time.sleep)

>>> conn2 = g.rpyc_get_connection('192.168.1.221', instance=2)
>>> async_sleep2 = rpyc.async(conn2.modules.time.sleep)

>>> res1 = async_sleep(60)
>>> res = async_sleep(10)
>>> res.wait()
>>> res1 = async_sleep(60)
>>> res2 = async_sleep2(10)
>>> res2.wait()
>>> res1.wait()
```

The first call will block on the first connection, while the second call runs in parallel on the other connection.

Running Local Code on the Remote System

Normally, a module already needs to reside on the remote system or be transferred at runtime to be called. Glusto leverages a feature of RPyC to define a local module on the remote system without the extra step of transferring a file into the remote PYTHONPATH.

This feature makes it simple to create module files of commonly used function, class, and method snippets for use on remote servers without the need to package, distribute, and install on each remote server ahead of time.

To define a local module on the remote system, use the `rpyc_define_module()` method.

Local module script named `mymodule` with a function called `get_uname`:

```
>>> import mymodule
>>> connection = g.rpyc_get_connection('192.168.1.221')
>>> r = g.rpyc_define_module(connection, mymodule)
>>> r.get_uname()
('Linux', 'rhserver1', '2.6.32-431.29.2.el6.x86_64', '#1 SMP Sun Jul 27␣
↪15:55:46 EDT 2014', 'x86_64')
```

Going Ape with Monkey-Patching

Monkey-patching with RPyC can be a useful feature.

Monkey-patching Standard Out

While using the Python interpreter, it is sometimes helpful to be able to see the output of a call that is normally directed to stdout on the remote.

To wire the remote stdout to the local stdout...

```
>>> import sys
>>> conn = g.rpyc_get_connection('192.168.1.221')
>>> conn.modules.sys.stdout = sys.stdout
>>> conn.execute("print 'Hello, World!'")
Hello, World!
```

Re-wiring Local and Remote

Monkey-patching can be used to make lengthy or often-used remote calls appear local.

An oversimplified example:

```
# Monkey-patching the remote to a local object
>>> conn = g.rpyc_get_connection('192.168.1.221')
>>> r_uname = conn.modules.os.uname
>>> r_uname()
('Linux', 'rhserver1', '2.6.32-431.29.2.el6.x86_64', '#1 SMP Sun Jul 27␣
↪15:55:46 EDT 2014', 'x86_64')

# Calling the local uname method
>>> import os
>>> os.uname()
('Linux', 'mylaptop', '4.4.9-300.fc23.x86_64', '#1 SMP Wed May 4 23:56:27␣
↪UTC 2016', 'x86_64')
```

A slightly better example:

```
>>> # create a function unaware of remote vs local
>>> def collect_os_data(os_object):
...     print os_object.uname()
...     print os_object.getlogin()
...

>>> # pass it the local object
>>> collect_os_data(os)
('Linux', 'mylaptop', '4.4.9-300.fc23.x86_64', '#1 SMP Wed May 4 23:56:27␣
↪UTC 2016', 'x86_64')
loadtheaccumulator

>>> # pass it the remote object
>>> collect_os_data(ros)
('Linux', 'rhserver1', '2.6.32-431.29.2.el6.x86_64', '#1 SMP Sun Jul 27␣
↪15:55:46 EDT 2014', 'x86_64')
root
```

Checking Connections

To check a connection is still available, use the `rpyc_ping_connection()` method.

```
>>> g.rpyc_ping_connection()
connection is alive
```

Note: Pinging a connection gets the connection from cache, but if the connection was not established before the ping, it will be opened–followed by the ping.

Closing Connections

On occasion, it might be necessary to remove a connection from the cache (e.g., when a cached connection is no longer needed or when looping through connections to execute the same command against all connections and an unwanted connection is in the list).

Warning: Closing a connection directly without using the methods discussed in this section will leave a connection definition in the connection dictionary. You will want to close rpyc connections via these methods to avoid unnecessary cleanup. It will also guarantee any future features are handled correctly upon close.

Closing a Single Connection

To remove a cached connection, close it with the `rpyc_close_connection()` method.

```
>>> g.rpyc_close_connection('192.168.1.221')

>>> g.rpyc_list_connections()
george@192.168.1.221:2
root@192.168.1.221:2
george@192.168.1.221:1

>>> g.rpyc_close_connection('192.168.1.221', user='george')

>>> g.rpyc_list_connections()
george@192.168.1.221:2
root@192.168.1.221:2

>>> g.rpyc_close_connection('192.168.1.221', user='george', instance=2)

>>> g.rpyc_list_connections()
root@192.168.1.221:2
```

Closing All Connections

To remove all cached connections, use the `rpyc_close_connections()` method.

```
>>> g.rpyc_close_connections()
```

Undeploying the RPyC Server

With the RPyC Zero-Deploy automated setup, the RPyC server process running on the remote system does not stop when a connection is closed. To stop that process, it is necessary to close the deployed server connection setup by

Zero-Deploy.

To list the deployed servers, use the `rpyc_list_deployed_servers()` method.

```
>>> g.rpyc_list_deployed_servers()
george@192.168.1.221
root@192.168.1.221
alexander@192.168.1.221
```

Note: When multiple connection instances to the same server with the same user exist, they share the same deployed server, so only one deployed server will appear in the list.

To close a deployed server connection, use the `rpyc_close_deployed_server()` method.

```
>>> g.rpyc_list_deployed_servers()
george@192.168.1.221
root@192.168.1.221

>>> g.rpyc_close_deployed_server('192.168.1.221', user='george')

>>> g.rpyc_list_deployed_servers()
root@192.168.1.221
```

Note: Glusto will automatically close all of the connection instances related to the deployed server being closed. However, it does not dispose of the cached SSH connection.

To close all deployed servers, use the `rpyc_close_deployed_servers()` method.

```
>>> g.rpyc_close_deployed_servers()
```

Note: Glusto leverages the RPyC Zero-Deploy methodology which copies the RPyC server files to the remote and sets up the SSH tunnel automatically. This can add overhead when the first `g.rpyc_get_connection()` call to a remote server is made. The time lag is negligible on the LAN or short distances across the WAN, but when dealing with a large number of systems across the globe, especially on a slow link (DSL, etc), there may be lengthy "go get something to drink" periods of time. Try it out and adjust according to your taste.

1.1.6 Using Config Files with Glusto

Glusto currently supports loading and storing configs in YAML, INI, and JSON format. File format can be specified explicitly or Glusto can determine the format based on file extension.

Loading Config Files

Config files can be loaded from the local filesystem as well as URLs.

Loading Config Files From the Local Filesystem

To load configuration from a file, use the `load_config()` method.

Example config file `examples/systems.yml`:

```
$ cat examples/systems.yml
clients: [192.168.1.225]
masternode: 192.168.1.221
nodes: [192.168.1.221, 192.168.1.222, 192.168.1.223, 192.168.1.224]
```

Example `load_config()`:

```
>>> config = g.load_config('examples/systems.yml')
>>> config
{'nodes': ['192.168.1.221', '192.168.1.222', '192.168.1.223', '192.168.1.224
↪'], 'clients': ['192.168.1.225'], 'masternode': '192.168.1.221'}
```

The `config` dictionary object now contains Python object representations of the config in the file.

Loading Config Files from a URL

To load configuration from a URL, pass the `load_config()` method a filename beginning with `http://`, `https://`, or `file://`.

```
>>> config = g.load_config('http://myserver.com/example.yaml')
```

Setting the Glusto Config Dictionary with a Config File

Glusto stores configs in a dictionary object named `config` at the root of the Glusto class. Using the `set_config()` method will assign a loaded configuration to the Glusto `config` class attribute.

The config will be available in any module where the Glusto class is imported.

Adding some data to demonstrate the effects of using `set_config()`:

```
>>> g.config['this'] = 'yada'
>>> g.config
{'this': 'yada'}
```

Example of using the `set_config()` method:

```
>>> config = g.load_config('examples/systems.yml')

>>> g.set_config(config)
>>> g.config
{'nodes': ['192.168.1.221', '192.168.1.222', '192.168.1.223', '192.168.1.224
↪'], 'clients': ['192.168.1.225'], 'masternode': '192.168.1.221'}
```

The Glusto class attribute `g.config` is now populated with the configuration loaded from file, and the `this` dictionary item is no longer there.

Warning: This is destructive. Any existing data in the `g.config` attribute will be overwritten by the data passed to `set_config()`.

Updating the Glusto Config Dictionary with a Config File

Updating with the `update_config` method is similar to using `set_config`, but will add to the config and not overwrite everything in the `config` class attribute automatically.

Adding some data to demonstrate the effects of using `update_config()`:

```
>>> g.config['this'] = 'yada'
>>> g.config
{'this': 'yada'}
```

Example of using the `update_config()` method:

```
>>> config = g.load_config('examples/systems.yml')

>>> g.update_config(config)
>>> g.config
{'this': 'yada','nodes': ['192.168.1.221', '192.168.1.222', '192.168.1.223',
↪'192.168.1.224'], 'clients': ['192.168.1.225'], 'masternode': '192.168.1.
↪221'}
```

With `update_config()`, the `this` dictionary item is still there.

To organize different configs in the `g.config` dictionary, you can leverage Python's ability to have nested dictionaries.

Example:

```
g.config['systems'] = {}
g.config['myapp'] = {}
```

Warning: When using nested dictionaries to separate different configs under the same `g.config` dictionary, as mentioned above, you will need to use update_config() instead of set_config() as described in the *Setting the Glusto Config Dictionary with a Config File* section.

Displaying Objects in Config File Format

To output objects to stdout in config file format, use the `show_config()` method.

```
>>> g.show_config(g.config)
clients: [192.168.1.225]
masternode: 192.168.1.221
nodes: [192.168.1.221, 192.168.1.222, 192.168.1.223, 192.168.1.224]
```

Storing Objects in Config File Format

Glusto provides a simple interface for formatting objects and storing them in a config file.

To format and store an object in a file, use the `store_config()` method.

```
>>> g.config
{'this': 'yada', 'nodes': ['192.168.1.221', '192.168.1.222', '192.168.1.223',
↪ '192.168.1.224'], 'clients': ['192.168.1.225'], 'masternode': '192.168.1.
↪221'}
```

(continues on next page)

(continued from previous page)

```
>>> g.store_config(g.config, filename='/tmp/glusto_config.yml')
```

```
$ cat /tmp/glusto_config.yml
clients: [192.168.1.225]
masternode: 192.168.1.221
nodes: [192.168.1.221, 192.168.1.222, 192.168.1.223, 192.168.1.224]
this: yada
```

The `store_config()` method will determine the config format based on the filename extension passed to it. If a format needs to be specified (maybe the extension does not represent the format), the format can be specified with the `config_type` parameter.

```
>>> g.store_config(g.config, filename='/tmp/glusto_config.conf, config_type=
↪'ini')
```

Note: Glusto currently defaults to yaml format.

Creating an INI Config Format Compatible Object

The INI format is simple in layout with a section header followed by key=value pairs. For that reason, an object being stored in INI format needs to be a dictionary (or dictionaries) of key:value dictionaries.

```
>>> config = {'section1': {'this': 'yada', 'that': 'yada yada'}, 'section2':
↪{'the_other': 'yada yada yada'}}
>>> config
{'section2': {'the_other': 'yada yada yada'}, 'section1': {'this': 'yada',
↪'that': 'yada yada'}}
```

Storing the INI Formatted Config

To store the INI formatted object, pass it to the `store_config()` method.

```
>>> g.store_config(config, filename='/tmp/config.ini')
```

```
$ cat /tmp/config.ini
[section2]
the_other = yada yada yada

[section1]
this = yada
that = yada yada
```

Note: Due to the nature of Python not maintaining order in certain objects, the order of the sections may not be the order in the dictionary being passed. To maintain section order, you will need to use an OrderedDict.

Storing the INI Formatted Config in a Specific Order

To store the INI formatted object with the sections in a specific order, pass it to the `store_config()` method as an OrderedDict object.

```
>>> from collections import OrderedDict
>>> config = OrderedDict()
>>> config.update('section1': {'this': 'yada'})
>>> config.update('section2': {'that': 'yada yada'})
>>> config.update('section3': {'the_other': 'yada yada yada'})
>>> g.store_config(config, '/tmp/ordered.ini')
```

```
$ cat /tmp/ordered.ini
[section1]
this = yada

[section2]
that = yada yada

[section3]
the_other = yada yada yada
```

Loading Config from a String

YAML formatted text can be converted into a dictionary object using the `load_yaml_string()` method.

```
>>> g.load_yaml_string(yaml_string)
{'clusters': ['e2effa75a5a50560c3250b67cf71b465']}
```

JSON formatted text can be converted into a dictionary object using the `load_json_string()` method.

```
>>> config = g.load_json_string(json_string)
>>> config
{u'clusters': [u'e2effa75a5a50560c3250b67cf71b465']}
```

Note: There is not a current method for loading an INI formatted string.

Adding Simple Configuration Capability to Your Own Class

Glusto provides an inheritable class (`Intraconfig`) that can add basic introspection and config functionality to classes in your scripts.

Making a Class Configurable

Making a class configurable is as simple as making it inherit from the Intraconfig class.

To inherit from the Intraconfig, add `Intraconfig` to the class definition.

Example making the class MyClass configurable:

```
>>> from glusto.configurable import Intraconfig
>>> class MyClass(Intraconfig):
>>>     def __init__(self):
>>>         self.this = 'yada1'
>>>         self.that = 'yada2'
```

Displaying the Class Config

To output attributes of the `myinst` instance of `MyClass`, use the inherited `show_config()` method.

Example with myinst as an instance of class MyClass:

```
>>> myinst = MyClass()
>>> myinst.show_config()
{that: yada2, this: yada1}
```

Loading Config from a File into Class Attributes

To load a config file into a dictionary attribute of a class instance, use the inherited `load_config()` method.

Example loading a config from `examples/systems.yml` into class instance `myinst`:

```
>>> myinst.load_config('examples/systems.yml')
>>> myinst.show_config()
clients: [192.168.1.225]
masternode: 192.168.1.221
nodes: [192.168.1.221, 192.168.1.222, 192.168.1.223, 192.168.1.224]
that: yada2
this: yada1
```

Storing Attributes of an Instance to File

To store the attributes of a class instance, use the inherited `store_config()` method.

Example storing the attributes from the `myinst` instance of `MyClass` to file `/tmp/myinst.yml`:

```
>>> myinst.store_config('/tmp/myinst.yml')
```

Looking at the contents of the resulting config file:

```
$ cat /tmp/myinst.yml
clients: [192.168.1.225]
masternode: 192.168.1.221
nodes: [192.168.1.221, 192.168.1.222, 192.168.1.223, 192.168.1.224]
that: yada
this: yada
```

Warning: Glusto will currently throw errors when using Instaconfig to store INI formatted config to file. Currently, the best way to store in INI format would be to form your config data, and then use `g.store_config()`.

1.1.7 Handling Logging with Glusto

Default Logging

By default, a logging object (`glustolog`) is setup by Glusto. It writes to `/tmp/glusto.log`

The `glustolog` object is designed for use by Glusto itself. If you need a log that is independent of the events logged by Glusto, you can create a new log object for use in your scripts.

Setting up Logging

To create a new logging object, use the `create_log` command:

```
>>> mylog = g.create_log(name='mylog', filename='/tmp/my.log')
```

The default severity level is `INFO`.

To create a logging object with a severity level other than the default:

```
>>> mylog = g.create_log(name='mylog', filename='/tmp/my.log', level='WARNING
↪')
```

The options are `INFO`, `WARNING`, `ERROR`, `CRITICAL`, and `DEBUG`

Sending a Log Event

To write a message to a log, use the standard Python logging methods with your log object.

An example sending a warning to `mylog`:

```
>>> mylog.warning('this is a test to my log')
>>> g.show_file('/tmp/my.log')
2016-07-10 09:07:11,591 WARNING (<module>) this is a test to my log
```

Available options are:

- <log>.debug
- <log>.info
- <log>.error
- <log>.warning
- <log>.critical

Sending Log Events to Multiple Logfiles

If you need to log to multiple files at the same time, you can add additional log handlers to an existing log object.

To add an additional logfile to an existing log object:

```
>>> g.add_log(g.mylog, filename='/tmp/my_other.log', level='CRITICAL')
```

If 'STDOUT' is passed as the filename, a log will be added that prints to stdout.

```
>>> g.add_log(g.mylog, filename='STDOUT')
>>> g.mylog.info('This is a test log entry to stdout.')
2016-06-05 10:27:24,175 INFO (<module>) This is a test log entry to stdout.
```

Show the Logfiles Attached to a Specific Logger

To show a list of the logfiles attached to a logger, use the `show_logs()` command.

```
>>> g.show_logs(g.mylog)
Log:  mylog
- mylog1: /tmp/my.log (WARNING)
- mylog2: /tmp/my_other.log (CRITICAL)
```

Removing a Log

If a logfile is no longer needed, remove the logfile from the logger with the `remove_log()` command.

```
>>> g.show_logs(g.mylog)
Log: mylog
- mylog1: /tmp/my.log
- mylog2: sys.stdout
>>> g.remove_log(g.mylog, 'mylog1')

>>> g.show_logs(g.mylog)
Log: mylog
- mylog1: /tmp/my.log
```

To remove all logfiles from a logger, use the `remove_log` command without passing a name.

```
>>> g.remove_log(g.mylog)
```

Changing the Level of an Existing Log Handler

To change the level of an existing log, use the `set_log_level()` method.

```
>>> g.show_logs(g.log)
Log:  glustolog
- glustolog1: /tmp/glusto.log (DEBUG)
- glustolog2: /tmp/testtrunc.log (INFO)

>>> g.set_log_level('glustolog', 'glustolog2', 'WARNING')

>>> g.show_logs(g.log)
Log:  glustolog
- glustolog1: /tmp/glusto.log (DEBUG)
- glustolog2: /tmp/testtrunc.log (WARNING)
```

Changing the Filename of an Existing Log Handler

To change the level of an existing log, use the `set_log_filename()` method.

```
>>> g.show_logs(g.log)
Log:  glustolog
- glustolog1: /tmp/glusto.log (DEBUG)
- glustolog2: /tmp/testtrunc.log (INFO)

>>> g.set_log_filename('glustolog', 'glustolog2', '/tmp/my.log')

>>> g.show_logs(g.log)
Log:  glustolog
- glustolog1: /tmp/glusto.log (DEBUG)
- glustolog2: /tmp/my.log (WARNING)
```

Clearing a Log

To empty a logfile, use the `clear_log()` method.

```
>>> g.show_logs(g.log)
Log:  glustolog
- glustolog1: /tmp/glusto.log (DEBUG)
- glustolog2: /tmp/testtrunc.log (INFO)
>>> g.clear_log('glustolog', 'glustolog2')
```

Temporarily Disable Logging

There might be times when suspending logging at a certain level is necessary. For example, if a particular function tends to spam the log.

To suspend logging at a specific level, use the `disable_log_levels()` method.

```
>>> g.disable_log_levels('WARNING')
```

Note: This will suspend logging for the specific level and all levels below it across all logs.

To resume logging at the previously defined levels, use the `reset_log_levels()` method.

```
>>> g.reset_log_levels()
```

Logging with Color Text

With the simple ANSI color capability built into Glusto, it is possible to add color text in logs or other output.

Changing the Color of a String

To wrap a string in color, use the colorfy command.

```
>>> print g.colorfy(g.RED, 'This string is RED')
```

The printed string will be output in the color red and any following text will return to default color.

See the "Available Color Values" below for the full list of Foreground Colors.

Changing the Background Color of a String

It is possible to change the background color of a string.

```
>>> print g.colorfy(g.BG_YELLOW, 'This string has a YELLOW background')
```

See the "Available Color Values" below for the full list of Background Colors.

Changing an ANSI Attribute of a String

It is also possible to make a string bold.

```
>>> print g.colorfy(g.BOLD, 'This string is BOLD')
```

Warning: Mileage may vary depending on the output device.

See the "Available Color Values" below for the full list of Attributes.

Combining Colors and Attributes

Glusto allows multiple combinations of color and attributes to be used at the same time.

To combine colors and attributes, pass a Bitwise Or'd list to `colorfy()`.

```
>>> print g.colorfy(g.BOLD | g. RED | g.BG_YELLOW, 'This string is BOLD and␣
↪RED on a YELLOW BACKGROUND.')
```

Tip: Create your own combinations ahead of time for re-use throughout your script.

```
>>> COLOR_ALERT =  g.BOLD | g.RED | g.REVERSE
>>> COLOR_WARNING =  g.BOLD | g.RED
>>> print '%s %s' %(g.colorfy(COLOR_ALERT, 'WARNING:'), g.colorfy(COLOR_
↪WARNING, 'This is a warning!'))
```

Send Color Text to the Log

Any of the previously discussed print commands can be replaced with logging commands to send the color text to logfiles.

```
>>> g.log.debug(g.colorfy(g.BOLD | g.RED | g.BG_YELLOW, 'This string is BOLD␣
↪and RED on a YELLOW BACKGROUND.'))
```

Enabling Color Logging for Built-In Commands

Color output is enabled by default and some of the Glusto internal commands (e.g., `g.run()`) already use color output for logging.

To disable the built-in color logging, add a line to the Glusto `/etc/glusto/defaults.yml` file.

```
log_color: False
```

To enable the built-in color logging, add a line to the Glusto `/etc/glusto/defaults.yml` file.

```
log_color: True
```

Color output can also be disabled by adding the log_color option to a config file loaded via `/usr/bin/glusto -c`.

Another method is to set `g.config['log_color'] = False` directly in your code.

Available Color Values

When using the color values listed in the table below, remember to add the Glusto `g.` reference in front of each color value.

For example, `g.BG_LTMAGENTA`

BACKGROUND	FOREGROUND	ATTRIBUTES
BG_DEFAULT	DEFAULT	NORMAL
BG_BLACK	BLACK	BOLD
BG_RED	RED	DIM
BG_GREEN	GREEN	UNDERLINE
BG_YELLOW	YELLOW	BLINK
BG_BLUE	BLUE	REVERSE
BG_MAGENTA	MAGENTA	HIDDEN
BG_CYAN	CYAN	
BG_LTGRAY	LTGRAY	
BG_DKGRAY	DKGRAY	
BG_LTRED	LTRED	
BG_LTGREEN	LTGREEN	
BG_LTYELLOW	LTYELLOW	
BG_LTBLUE	LTBLUE	
BG_LTMAGENTA	LTMAGENTA	
BG_LTCYAN	LTCYAN	
BG_WHITE	WHITE	

1.1.8 Handling Templates with Glusto

Create a File from a Template

```
>>> g.render_template("templates/template_testcase.jinja",
... template_vars, "/tmp/testcase.py", searchpath='examples/')
```

The Template File

Template Variables

Providing Vars Programmatically

Providing Vars in a Config File

A Bit About Search Path

1.1.9 PyUnit (unittest) and Glusto

Glusto plugs into the Python unittest module methodology and can run test modules with the Glusto features imported or existing scripts without Glusto imported in the test module. This makes it possible to combine tests that do not require any of the Glusto functionality with Glusto-savvy test cases.

Running Unittests with Glusto

Glusto supports running test cases in a number of ways.

- Using the Python interpreter interactive mode to execute commands manually.
- Running an individual script at the command-line via `if __name__ == '__main__':` logic.
- Running an individual script via an IDE (e.g., Eclipse PyDev plug-in[1] test runner).
- Running testcases directly at the command-line via the `python -m unittest` module.
- Using the glusto cli command with CLI options or config files.

Each of the above options is documented below.

Note: All of the examples are based on the samples provided in the `examples` directory installed with the glusto package.

Running Unittests via the Python Interpreter Interactive Mode

Running the tests via the Python interpreter can come in handy while developing and debugging tests scripts.

To run tests via the Python interpreter...

1. Enter the Python Interpreter.

```
# python
```

2. Type commands...

   For example:

```
>>> import unittest
>>> loader = unittest.TestLoader()
>>> suite = loader.loadTestsFromName('tests.test_glusto.TestGlustoBasics.
↪test_stdout')
```

(continues on next page)

[1] http://www.pydev.org/

(continued from previous page)

```
>>> runner = unittest.TextTestRunner()
>>> runner.run(suite)
Setting Up Class: TestGlustoBasics
Setting Up: tests.test_glusto.TestGlustoBasics.test_stdout
Running: tests.test_glusto.TestGlustoBasics.test_stdout - Testing output␣
↪to stdout
Tearing Down: tests.test_glusto.TestGlustoBasics.test_stdout
Cleaning up after setup on fail or after teardown
.Tearing Down Class: TestGlustoBasics

----------------------------------------------------------------------
Ran 1 test in 0.242s

OK
<unittest.runner.TextTestResult run=1 errors=0 failures=0>
```

Running Unittests with the CLI Option

The simplest and most direct way to run unittests with Glusto is via the glusto cli discover option.

The discover (*-d,–discover*) option accepts a directory name and the unittest module discover feature[2] searches the directory and subdirectories for modules with a name matching `` `test*.py` ``.

Note: It is possible to fine-tune the discover options with a specific pattern and top_level_directory via the configuration option described below.

Example:

```
# glusto -d 'tests' -c 'examples/systems.yml'
```

Running Tests in a Specific Order

One of the most argued elements of unit testing is creating a relationship between test cases by specifying an order where one has to be run before the other to satisfy a dependency, etc. Some will say test cases should never be related and always standalone. Some say there is a need in integration testing where it makes sense to leverage one test case to setup another without jumping through programmatic hoops to prevent a dependency. Regardless of which camp you might side with, the Python unittest module can be leveraged for a variety of use cases, so Glusto provides a convenient interface to the capability added in Python 2.7.

Add the following example to the test module containing a standard unittest.TestCase class.

```
1 def load_tests(loader, standard_tests, pattern):
2             '''Load tests in specified order'''
3     testcases_ordered = ['test_return_code',
4                          'test_stdout',
5                          'test_stderr']
6
7     suite = g.load_tests(TestGlustoBasics, loader, testcases_ordered)
8
9     return suite
```

[2] https://docs.python.org/2.7/library/unittest.html#unittest.TestLoader.discover

See `tests/test_glusto.py` for a full example of a unittest.TestCase using Glusto and running tests in order.

Running Unittests with the Configuration Options

If more control over discovery options, or the ability to select tests at the module or test case level is required, you can use config files to specify those requirements.

To run tests with information from config files, use the -u option:

```
# glusto -u -c 'examples/systems.yml examples/unittests/unittest.yml examples/
↪unittests/unittest_list.yml'
```

Configuring Glusto for Unittests

Along with the simple discovery method at the CLI, Glusto supports more granular control over Unittests via configuration files.

Base Unittest Options

Configuration items that control options Glusto-wide can be configured.

output_junit The `output_junit` option writes the test results in junit xml format.

```
unittest:
    output_junit: false
```

test_method_prefix The `test_method_prefix` option changes the name prefix used by unittest to discover tests.

```
unittest:
    test_method_prefix: 'rhgs'
```

Discover Tests from a Directory

Discovery via config is similar to the CLI, but offers additional options.

Config:

```
# DISCOVER TESTS FROM DIRECTORY
  discover_tests:
    start_dir: 'tests'
    # optional
    pattern: 'test*.py'
    top_level_dir: 'tests'
```

Load Tests from a List

To run a specific set of tests, Glusto supports configuring a list.

Config (unittest.yml):

```
# LOAD TESTS FROM LIST (SEE unittest_list.yml)
  load_tests_from_list: true
```

Config (unittest_list.yml):

```
unittest_list:
  module_name: 'tests.test_glusto'
  list: [
    'TestGlustoBasics.test_stdout',
    'TestGlustoBasics.test_return_code',
    'TestGlustoBasics.test_stderr',
    'TestGlustoBasics.test_expected_fail',
    ]
```

Load Tests from a Module

To limit test list to only those in a specific module, use the `load_tests_from_module` option. Tests are discovered automatically and run in alphabetical order.

Config:

```
# LOAD TESTS FROM MODULE w/ TEST_LOAD ORDERED TESTS
  load_tests_from_module:
    module_name: 'tests.test_glusto'
    use_load_tests: false
```

Load Tests from a Module with Ordered Test List

To limit the test list to a specific module and specify an order, set the `use_load_test` option to `true`.

Config:

```
# LOAD TESTS FROM MODULE w/o TEST_LOAD ORDERED TESTS
  load_tests_from_module:
    module_name: 'tests.test_glusto'
    use_load_tests: true
```

Note: When setting `use_load_tests: true` it is necessary to add a `load_tests()` method to your test script. For more information on the load_tests() method, please see the "*Running Tests in a Specific Order*" section earlier in this doc.

Load a Test Using a Name

To limit the test to a specific test module, class, or method, use the `load_tests_from_name` option.

Config:

```
# LOAD TESTS FROM NAME
  load_tests_from_name: 'tests.test_glusto.TestGlustoBasics.test_stdout'
```

When providing a module, the list is created from all tests in the module.

```
load_tests_from_name: 'tests.test_glusto'
```

When providing a class, the list is created from all tests in the class.

```
load_tests_from_name: 'tests.test_glusto_configs'
```

When providing a method, only that method is run.

```
load_tests_from_name: 'tests.test_glusto.TestGlustoBasics.test_stdout'
```

Load Tests from a List of Names

To limit the test to a list of names described above, use the `load_tests_from_names` option.

Config:

```
# LOAD TESTS FROM LIST OF NAMES
  load_tests_from_names: ['tests.test_glusto',
                          'tests.test_glusto_configs',
                          'tests.test_glusto.TestGlustoBasics.test_stdout']
```

The list will be composed of all tests combined.

Writing Unittests

Glusto's unit test features are based on the Python unittest module. The unittest module provides a simple class structure that makes testcase development rather robust without modification.

To use the `unittest` module for creating a testcase, import the `unittest` module and create a subclass of `unittest.TestCase`.

```
1  import unittest
2
3  class MyTestClass(unittest.TestCase)
```

The base class for `unittest` is `unittest.TestCase`. It consists of several automatically called methods that are designed to be overridden to provide your own functionality.

Note: In the future, I will look at integrating some PyTest or other frameworks, but don't have an immediate need.

Example Using setUp and tearDown

`test_glusto_configs.py`

Example Using setUpClass and tearDownClass

`test_glusto_templates.py`

To Do

- Expand the Writing Test Cases section with more examples.

1.1.10 PyTest and Glusto

Running PyTests from the CLI

Use the `-t=` or `--pytest=` parameter followed by the options normally passed to `py.test`

```
$ glusto -c 'examples/systems.yml' --pytest='-v -x tests -m response'
Starting glusto via main()
...
pytest: -v -x tests -m response
========================== test session starts␣
↪============================================
platform linux2 -- Python 2.7.11, pytest-2.9.2, py-1.4.31, pluggy-0.3.1 -- /
↪usr/bin/python
cachedir: .cache
rootdir: glusto, inifile:
collected 21 items

tests/test_glusto_pytest.py::TestGlustoBasicsPyTest::test_return_code PASSED
tests/test_glusto_pytest.py::TestGlustoBasicsPyTest::test_stderr PASSED
tests/test_glusto_pytest.py::TestGlustoBasicsPyTest::test_stdout PASSED

======================== 18 tests deselected by "-m 'response'"␣
↪=========================
======================== 3 passed, 18 deselected in 0.62 seconds␣
↪=========================
Ending glusto via main()
```

For a list of available options, pass `--help` to the `pytest` parameter or use the `py.test` command itself.

```
$ glusto --pytest='--help'
$ py.test --help
```

Running PyTest from Python Interactive Interpreter

```
>>> import pytest
>>> pytest.main('-v -x tests -m response')
========================== test session starts␣
↪============================================
platform linux2 -- Python 2.7.11, pytest-2.9.2, py-1.4.31, pluggy-0.3.1 -- /
↪usr/bin/python
cachedir: .cache
rootdir: glusto, inifile:
collected 21 items

tests/test_glusto_pytest.py::TestGlustoBasicsPyTest::test_return_code PASSED
tests/test_glusto_pytest.py::TestGlustoBasicsPyTest::test_stderr PASSED
tests/test_glusto_pytest.py::TestGlustoBasicsPyTest::test_stdout PASSED

======================== 18 tests deselected by "-m 'response'"␣
↪=========================
```

(continues on next page)

(continued from previous page)

```
======================== 3 passed, 18 deselected in 0.55 seconds␣
↪==========================
0
```

To make config files available to test cases when running interactively, use the `load_config` and `update_config` methods.

```
>>> from glusto.core import Glusto as g
>>> config = g.load_config('examples/systems.yml')
>>> g.update_config(config)
>>> import pytest
>>> pytest.main('-v -x tests -m response')
```

To Do

- Expand text and examples

1.1.11 Nose and Glusto

Running Nosetests from the CLI

Use the `-n=` or `--nostests=` parameter followed by the options normally passed to `nosetests`

```
$ glusto -c 'examples/systems.yml' --nosetests='-v -w tests'
Starting glusto via main()
...
nosetests: -v -w tests
Testing an expected failure. This test should fail ... /usr/lib64/python2.7/
↪unittest/case.py:378: RuntimeWarning: TestResult has no addExpectedFailure␣
↪method, reporting as passes
  RuntimeWarning)
ok
Testing an expected failure as negative test ... ok
Testing the return code ... ok
Testing the unittest skip feature ... SKIP: Example test skip
Testing output to stderr ... ok
Testing output to stdout ... ok
Load tests in a specific order. ... ERROR
Testing ini config file(s) ... ok
Testing ordered ini config file(s) ... ok
Testing yaml config file ... ok
Testing an expected failure. This test should fail ... FAIL
Testing an expected failure as negative test ... ok
Testing the return code ... ok
Testing the unittest skip feature ... ok
Testing output to stderr ... ok
Testing output to stdout ... ok
Load tests in a specific order. ... ERROR
Testing rpyc connection ... ok
Testing local module definition on remote system ... ok
test_remote_call (tests.test_glusto_rpyc.TestGlustoRpyc) ... ok
Testing template for loop ... ok
Testing template include ... ok
```

(continues on next page)

(continued from previous page)

```
Testing template scalar ... ok

...

======================================================================
FAIL: Testing an expected failure. This test should fail
----------------------------------------------------------------------
Traceback (most recent call last):
  File "glusto/tests/test_glusto_pytest.py", line 98, in test_expected_fail
    self.assertEqual(rcode, 0)
AssertionError: 1 != 0
-------------------- >> begin captured stdout << ---------------------
Setting Up: tests.test_glusto_pytest.TestGlustoBasicsPyTest.test_expected_
↪fail
Running: tests.test_glusto_pytest.TestGlustoBasicsPyTest.test_expected_fail -
↪ Testing an expected failure. This test should fail

--------------------- >> end captured stdout << ----------------------
-------------------- >> begin captured logging << --------------------
plumbum.local: DEBUG: Running ['/usr/bin/ssh', '-T', '-
↪oPasswordAuthentication=no', '-oStrictHostKeyChecking=no', '-oPort=22', '-
↪oConnectTimeout=10', '-oControlMaster=auto', '-oControlPersist=4h', '-
↪oControlPath=~/.ssh/glusto-ssh-%r@%h:%p', 'root@192.168.1.221', 'cd', '/
↪root', '&&', 'false']
--------------------- >> end captured logging << ---------------------

----------------------------------------------------------------------
Ran 23 tests in 2.791s

FAILED (SKIP=1, errors=2, failures=1)
Ending glusto via main()
```

For a list of available options, pass `--help` to the `nosetests` parameter or use the `nosetests` command itself.

```
$ glusto --noseests='--help'
$ nosetests --help
```

Running Nosetests from Python Interpreter

```
>>> import nose
>>> nose.run(argv=['-v', '-w', 'tests'])
```

To Do

- Expand text and examples

1.1.12 Working with PyTest and Nose

It is easy to integrate Glusto capability into a PyTest or Nose formatted test script. Simply add the import to the script and all of the Glusto methods are available.

```
import pytest
from glusto.core import Glusto as g
```

The Glusto command-line utility currently wraps the PyUnit mechanism for discovering and running tests. PyUnit tests using Glusto can also run under PyTest and, with some exceptions, Nose.

Note: Except where noted, the examples provided here are currently only written with the Python `unittest` module in mind. This section discusses running those scripts under PyTest and Nose. At some point, I will write a set of specific examples for each. For now, this discussion is intended to demonstrate the flexibility Glusto as a library of utilities can provide across the three popular test frameworks.

The following examples use the tests included in the `tests` directory.

PyTest

With the provided examples, PyTest runs the tests without issue and even handles skipped tests and expected failures. There are some notable exceptions they make (e.g., ignoring test order with load_tests).

A simple py.test run:

```
$ py.test
==================== test session starts ============================
platform linux2 -- Python 2.7.11, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /home/loadtheaccumulator/glusto, inifile:
collected 21 items

test_glusto.py x..s..
test_glusto_configs.py ...
test_glusto_pytest.py x..s..
test_glusto_rpyc.py ...
test_glusto_templates.py ...

============= 17 passed, 2 skipped, 2 xfailed in 2.63 seconds ========
```

The same run with verbose output:

```
$ py.test -v
================== test session starts ==============================
platform linux2 -- Python 2.7.11, pytest-2.9.2, py-1.4.31, pluggy-0.3.1 -- /
↪usr/bin/python
cachedir: ../.cache
rootdir: /home/loadtheaccumulator/Dropbox/glusto, inifile:
collected 21 items

test_glusto.py::TestGlustoBasics::test_expected_fail <- ../../../../usr/
↪lib64/python2.7/unittest/case.py xfail
test_glusto.py::TestGlustoBasics::test_negative_test PASSED
test_glusto.py::TestGlustoBasics::test_return_code PASSED
test_glusto.py::TestGlustoBasics::test_skip_me <- ../../../../usr/lib64/
↪python2.7/unittest/case.py SKIPPED
test_glusto.py::TestGlustoBasics::test_stderr PASSED
test_glusto.py::TestGlustoBasics::test_stdout PASSED
test_glusto_configs.py::TestGlustoConfigs::test_ini PASSED
test_glusto_configs.py::TestGlustoConfigs::test_ini_ordered PASSED
test_glusto_configs.py::TestGlustoConfigs::test_yaml PASSED
```

(continues on next page)

(continued from previous page)

```
test_glusto_pytest.py::TestGlustoBasicsPyTest::test_expected_fail <- ../../..
↪/../usr/lib64/python2.7/unittest/case.py xfail
test_glusto_pytest.py::TestGlustoBasicsPyTest::test_negative_test PASSED
test_glusto_pytest.py::TestGlustoBasicsPyTest::test_return_code PASSED
test_glusto_pytest.py::TestGlustoBasicsPyTest::test_skip_me SKIPPED
test_glusto_pytest.py::TestGlustoBasicsPyTest::test_stderr PASSED
test_glusto_pytest.py::TestGlustoBasicsPyTest::test_stdout PASSED
test_glusto_rpyc.py::TestGlustoRpyc::test_connection PASSED
test_glusto_rpyc.py::TestGlustoRpyc::test_local_module_on_remote PASSED
test_glusto_rpyc.py::TestGlustoRpyc::test_remote_call PASSED
test_glusto_templates.py::TestGlustoTemplates::test_template_forloop PASSED
test_glusto_templates.py::TestGlustoTemplates::test_template_include PASSED
test_glusto_templates.py::TestGlustoTemplates::test_template_scalar PASSED

============== 17 passed, 2 skipped, 2 xfailed in 2.85 seconds ========
```

PyTest supports running PyUnit and Nose tests, so it's simple to leverage PyTest features, such as markers, in a PyUnit script. See the `test_glusto_pytest.py` script for an example of combining PyTest marker features in a unittest.

A test run of only tests with a py.test formatted marker *response*:

```
$ py.test -v -m "response"
===================== test session starts ============================
platform linux2 -- Python 2.7.11, pytest-2.9.2, py-1.4.31, pluggy-0.3.1 -- /
↪usr/bin/python
cachedir: ../.cache
rootdir: /home/loadtheaccumulator/Dropbox/glusto, inifile:
collected 21 items

test_glusto_pytest.py::TestGlustoBasicsPyTest::test_return_code PASSED
test_glusto_pytest.py::TestGlustoBasicsPyTest::test_stderr PASSED
test_glusto_pytest.py::TestGlustoBasicsPyTest::test_stdout PASSED

============== 18 tests deselected by "-m 'response'" ================
============== 3 passed, 18 deselected in 0.69 seconds ================
```

Nose

Nose successfully runs the test methods, but does not handle the load_tests function as easily as PyTest.

```
$ nosetests
/usr/lib64/python2.7/unittest/case.py:378: RuntimeWarning: TestResult has no␣
↪addExpectedFailure method, reporting as passes
  RuntimeWarning)
...S..E.........E......
======================================================================
ERROR: Load tests in a specific order.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/usr/lib/python2.7/site-packages/nose/case.py", line 197, in runTest
    self.test(*self.arg)
TypeError: load_tests() takes exactly 3 arguments (0 given)

======================================================================
ERROR: Load tests in a specific order.
```

(continues on next page)

(continued from previous page)

```
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/usr/lib/python2.7/site-packages/nose/case.py", line 197, in runTest
    self.test(*self.arg)
TypeError: load_tests() takes exactly 3 arguments (0 given)

----------------------------------------------------------------------
Ran 23 tests in 2.253s

FAILED (SKIP=1, errors=2)
```

Twenty-three tests run successfully, two with errors (expected though), and 1 skipped. Apparently Nose ignores the py.test markers and did not skip a test in the py.test example.

A Nose test run with results written to an xunit xml file:

```
$ nosetests --with-xunit --xunit-file=/tmp/nosetests.xml
```

Glusto for Good Measure

```
$ glusto -d 'tests'
Starting glusto via main()

...

----------------------------------------------------------------------
Ran 21 tests in 2.522s

OK (skipped=1, expected failures=2)
```

Not surprisingly, the `unittest` module does not recognize the PyTest skip marker, so it is currently necessary to run PyTest-savvy scripts with the `py.test` command.

To Do

- split this out and merge into individual test framework pages
- Add examples of PyTest and Nose specific test scripts using Glusto calls.

more on this subject later...

1.1.13 Dynamic Cartesian Product Test Case Creation

Glusto provides a decorator class that, given some variables by overriding the __init__ method and passing data in a decorator, will create test cases on the fly based on cartesian product combinations.

Currently, the only decorator available in Carteplex is the CarteTestClass. It was written with `unittest.TestCase` in mind and includes some wiring specific to unittest (leveraging load_tests, test class-based, etc.). The Cartetestclass is available with the same Glusto import that provides the other functionality.

Note: Carteplex is a new feature to meet a very specific use case. Currently, the unittest and PyTest loaders/runners work without issue. The Nose runner currently chokes on the load_test function. The goal, of course, is to keep it

generic and make it work across the test frameworks. Working on it.

Using Carteplex

To make the CarteTestClass decorator available to your test class, import Glusto.

```
from glusto.core import Glusto as g
```

An Example Config File

Test case scripts accept dynamic configuration variables from config files. In this case, a configuration state is added to a config file for the base class to use when creating the test case classes on the fly.

Listing 1: gluster_config.yml

```
run_on_volumes : 'ALL'
run_on_mounts : 'ALL'
```

Subclassing the CarteTestClass Decorator Class

Providing information specific to the tests being run is necessary for the decorator to work correctly. This is accomplished by subclassing the decorator class and overriding the `__init__()` method to provide the specific data.

There are three attributes in particular that the CarteTestClass uses to create the resulting array of test cases.

axis_names This is a list of names used to automatically create class attributes in the resulting test classes. For example, adding the name *volume* results in the class attribute named `volume` being added to the test class with the value resulting from the cartesian product process. In this way, the test methods have access to the values of that specific combination.

selections This is a list of lists containing the options passed in via config file. For example, in the case of the gluster examples below, the desired values for the configurations to be run are passed in via a config file. The decorator in this case would be used to *limit* the configurations each test class can be run on. The CarteTestClass intersects the *selections* lists with the *limits* list to create the resulting list of used to create a test class for each combination.

limits Limits are automatically passed in via the decorator and are used to limit the *selections* the test case can/will run on.

available_options This is the full list of all values allowed for a specific attribute and is used to populate the *selections* value when 'ALL' is specified for the *selections* attribute.

Listing 2: gluster_base_class.py

```
import unittest

from glusto.core import Glusto as g


class runs_on(g.CarteTestClass):
    """Decorator providing runs_on capability for standard unittest script"""

    def __init__(self, value):
```

(continues on next page)

(continued from previous page)

```
        self.axis_names = ['volume', 'mount']

        self.available_options = [['distributed', 'replicated',
                                   'distributed-replicated',
                                   'disperse', 'distributed-disperse'],
                                  ['glusterfs', 'nfs', 'cifs']]

        config = g.load_config('gluster_config.yml')
        if config:
            g.update_config(config)
        run_on_volumes = g.config.get('run_on_volumes',
                                      self.available_options[0])
        run_on_mounts = g.config.get('run_on_mounts',
                                     self.available_options[1])
        self.selections = [run_on_volumes, run_on_mounts]
        self.limits = value
```

Creating a Custom unittest.TestCase Subclass

Listing 3: gluster_base_class.py

```
class GlusterBaseClass(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        print "setUpClass: %s" % cls.__name__
        print "SETUP GLUSTER VOLUME: %s on %s" % (cls.volume, cls.mount)

    def setUp(self):
        """Setting this up"""
        print "\tsetUp: %s - %s" % (self.id(), self.shortDescription())

    def tearDown(self):
        print "\ttearDown: %s - %s" % (self.id(), self.shortDescription())

    @classmethod
    def tearDownClass(cls):
        print "tearDownClass: %s" % cls.__name__
        print "TEARDOWN GLUSTER VOLUME: %s on %s" % (cls.volume, cls.mount)
```

Using the Decorator

Listing 4: test_gluster_runsauto.py

```
from glusto.core import Glusto as g

from gluster_base_class import GlusterBaseClass
from gluster_base_class import runs_on

import pytest
import unittest

```

(continues on next page)

(continued from previous page)

```

volumes = ['distributed', 'replicated', 'disperse']
mounts = ['glusterfs', 'nfs']


@runs_on([volumes, mounts])
class MyGlusterTest(GlusterBaseClass):
    def test_gluster1(self):
        """Test 1"""
        print "\t\tRunning: %s - %s" % (self.id(), self.shortDescription())
        print "\t\t%s on mount %s" % (self.volume, self.mount)

    @pytest.mark.test2
    def test_gluster2(self):
        """Test 2"""
        print "\t\tRunning: %s - %s" % (self.id(), self.shortDescription())
        print "\t\t%s on mount %s" % (self.volume, self.mount)

    @pytest.mark.skip
    def test_gluster3(self):
        """Test 3"""
        print "\t\tRunning: %s - %s" % (self.id(), self.shortDescription())
        print "\t\t%s on mount %s" % (self.volume, self.mount)
```

Run the Tests

```
$ glusto -c 'examples/systems.yml tests_gluster/gluster_conf.yml' --pytest='-vv -q␣
↪tests_gluster/test_gluster_runsauto.py'
Starting glusto via main()
...
pytest: -vvv -q tests_gluster/test_gluster_runsauto.py
=============================================================================␣
↪test session starts␣
↪=============================================================================
platform linux2 -- Python 2.7.11, pytest-2.9.2, py-1.4.31, pluggy-0.3.1 -- /usr/bin/
↪python
cachedir: .cache
rootdir: glusto, inifile:
collected 18 items

tests_gluster/test_gluster_runsauto.py::MyGlusterTest_distributed_nfs::test_gluster1␣
↪PASSED
tests_gluster/test_gluster_runsauto.py::MyGlusterTest_distributed_nfs::test_gluster2␣
↪PASSED
tests_gluster/test_gluster_runsauto.py::MyGlusterTest_distributed_nfs::test_gluster3␣
↪SKIPPED
tests_gluster/test_gluster_runsauto.py::MyGlusterTest_replicated_glusterfs::test_
↪gluster1 PASSED
tests_gluster/test_gluster_runsauto.py::MyGlusterTest_replicated_glusterfs::test_
↪gluster2 PASSED
tests_gluster/test_gluster_runsauto.py::MyGlusterTest_replicated_glusterfs::test_
↪gluster3 SKIPPED
tests_gluster/test_gluster_runsauto.py::MyGlusterTest_disperse_glusterfs::test_
↪gluster1 PASSED
```

(continues on next page)

(continued from previous page)

```
tests_gluster/test_gluster_runsauto.py::MyGlusterTest_disperse_glusterfs::test_gluster2 PASSED
tests_gluster/test_gluster_runsauto.py::MyGlusterTest_disperse_glusterfs::test_gluster3 SKIPPED
tests_gluster/test_gluster_runsauto.py::MyGlusterTest_disperse_nfs::test_gluster1 PASSED
tests_gluster/test_gluster_runsauto.py::MyGlusterTest_disperse_nfs::test_gluster2 PASSED
tests_gluster/test_gluster_runsauto.py::MyGlusterTest_disperse_nfs::test_gluster3 SKIPPED
tests_gluster/test_gluster_runsauto.py::MyGlusterTest_distributed_glusterfs::test_gluster1 PASSED
tests_gluster/test_gluster_runsauto.py::MyGlusterTest_distributed_glusterfs::test_gluster2 PASSED
tests_gluster/test_gluster_runsauto.py::MyGlusterTest_distributed_glusterfs::test_gluster3 SKIPPED
tests_gluster/test_gluster_runsauto.py::MyGlusterTest_replicated_nfs::test_gluster1 PASSED
tests_gluster/test_gluster_runsauto.py::MyGlusterTest_replicated_nfs::test_gluster2 PASSED
tests_gluster/test_gluster_runsauto.py::MyGlusterTest_replicated_nfs::test_gluster3 SKIPPED

========================================================================== 12 passed, 6 skipped in 0.04 seconds ==========================================================================
Ending glusto via main()
```

ToDo

- Add decorator for function and method.
- Finish this document.

1.1.14 Glusto Simple REST Client

Glusto provides simple methods for basic REST API get, post, put, and delete functionality.

Making REST API Requests

Glusto supports the four basic REST API request types.

- GET
- POST
- PUT
- DELETE

Making a GET Request

To submit a GET request to a url, use the `rest_get()` method.

```
>>> g.rest_get('http://httpbin.org/get')
```

Making a POST Request

To submit a POST request to a url, use the `rest_post()` method.

```
>>> g.rest_post('http://httpbin.org/post', data={'this': 'yada1', 'that':
↪'yada2'})
```

Making a PUT Request

To submit a PUT request to a url, use the `rest_put()` method.

```
>>> g.rest_put('http://httpbin.org/put', data={'this': 'yada1', 'that':
↪'yada2'})
```

Making a DELETE Request

To submit a DELETE request to a url, use the `rest_delete()` method.

```
>>> g.rest_delete('http://httpbin.org/delete', data={'this': 'yada1', 'that
↪': 'yada2'})
```

Handling the Request Response

Glusto provides the return in a tuple similar to the SSH calls in the `glusto.Connectible` class.

```
>>> g.rest_get('http://192.168.1.112:8081/hello')
(0, 'HelloWorld from GlusterFS Application', None)
```

The returned tuple consists of the return code, the response output, and the response error.

Note: The return code is the standard HTTP code returned by the web server on server response error (e.g., 404), otherwise returns zero for a successful request.

Using the Request Response as Config

The output of the response will be in string format.

YAML Formatted Text

If the string is yaml formatted text, it can be converted into a dictionary object using the `load_yaml_string()` method.

```
>>> g.rest_get('http://192.168.1.112:8081/clusters')
(0, '{"clusters":["e2effa75a5a50560c3250b67cf71b465"]}\n', None)

>>> rcode, rout, rerr = g.rest_get('http://192.168.1.112:8081/clusters')[1]
>>> g.load_yaml_string(rout)
{'clusters': ['e2effa75a5a50560c3250b67cf71b465']}
```

JSON Formatted Text

If the string is json formatted text, it can be converted into a dictionary object using the `load_json_string()` method.

```
>>> g.rest_get('http://192.168.1.112:8081/clusters')
(0, '{"clusters":["e2effa75a5a50560c3250b67cf71b465"]}\n', None)

>>> rcode, rout, rerr = g.rest_get('http://192.168.1.112:8081/clusters')[1]
>>> g.load_json_string(out)
{u'clusters': [u'e2effa75a5a50560c3250b67cf71b465']}
```

1.1.15 To Do

- better docs and more examples
- move print noise to log info and debug

CHAPTER 2

API

2.1 glusto package

2.1.1 Submodules

glusto.carteplex module

Cartesian product decorator class for on-the-fly creation of testcases based on a matrix of lists.

- **Handles automatic generation of cartesian product combinations based on any** number of lists.
- Provides an 'ALL' default to populate a var with a predefined full list.
- **Compares variables passed in decorator to a list of configuration options** (e.g., specify what underlying configuration is available to the test via config file and specify the configurations the test is limited to... Glusto will only create the possible combinations based on the intersection).
- **Passes "axis names" into test class as class attributes to make them** available to test methods.
- Provided via the Glusto import "from glusto.core import Glusto as g"

To use, 1) Subclass CarteTestClass, 2) Override __init__ to define variables, 3) Import your subclass and add a decorator of the same name to your tests.

Carteplex is "Cartesian Multiplexing". And that's simplified from the original Carteprodplexorator (Cartesian Product Multiplexing Decorator).

See docs for more information and use-case examples.

class glusto.carteplex.**Carteplex**
Bases: object

Main class for cartesian product classes

class CarteTestClass(*value*)
Bases: object

Decorator providing cartesian product parameter-like capability for unittest class

Override to provide data specific to your tests.

Parameters **value** (*object*) – data automatically provided by the decorator.

__call__(*obj*)

The engine behind the cartesian product multiplexing (carteplex) goodness. Do not override.

Parameters **obj** (*object*) – object automatically passed by the decorator.

Returns An empty object. (removes original testclass from run)

glusto.colorfiable module

All things ANSI color text output.

Note: Colorfiable is inherited by the Glusto class and not designed to be instantiated.

class glusto.colorfiable.**Colorfiable**

Bases: object

Defines and displays ANSI-compatible colors for string formatting.

BG_DEFAULT = 1

BG_BLACK = 2

BG_RED = 4

BG_GREEN = 8

BG_YELLOW = 16

BG_BLUE = 32

BG_MAGENTA = 64

BG_CYAN = 128

BG_LTGRAY = 256

BG_DKGRAY = 512

BG_LTRED = 1024

BG_LTGREEN = 2048

BG_LTYELLOW = 4096

BG_LTBLUE = 8192

BG_LTMAGENTA = 16384

BG_LTCYAN = 32768

BG_WHITE = 65536

DEFAULT = 131072

BLACK = 262144

RED = 524288

GREEN = 1048576

YELLOW = 2097152

BLUE = 4194304

MAGENTA = 8388608

CYAN = 16777216

LTGRAY = 33554432

DKGRAY = 67108864

LTRED = 134217728

LTGREEN = 268435456

LTYELLOW = 536870912

LTBLUE = 1073741824

LTMAGENTA = 2147483648

LTCYAN = 4294967296

WHITE = 8589934592

NORMAL = 0

BOLD = 17179869184

DIM = 34359738368

UNDERLINE = 68719476736

BLINK = 137438953472

REVERSE = 274877906944

HIDDEN = 549755813888

COLOR_COMMAND = 17246978048
Constant for command strings (BOLD | DKGRAY)

COLOR_STDOUT = 17180131584
Constant for stdout (BOLD | BG_LTGRAY | BLACK)

COLOR_STDERR = 17180393472
Constant for stderr (BOLD | RED)

COLOR_RCODE = 17184063488
Constant for command return code (BOLD | BLUE)

classmethod colorfy(*color*, *message*)
Applies ANSI terminal colors and attributes to strings.

Parameters

- **color** (*int*) – Bitwise value(s) for color settings.
- **message** (*str*) – String to wrap in the specified color.

Returns A color formatted string.

Example

```
>>> g.colorfy(g.BG_CYAN | g.RED | g.BOLD, 'Bold red text on cyan')
```

glusto.configurable module

All things configuration.

Note: Configurable is inherited by the Glusto class and not designed to be instantiated.

class glusto.configurable.**Configurable**
Bases: object

The class providing all things configuration.

config = {}
The default class attribute for storing configurations.

static store_config(*obj*, *filename*, *config_type=None*, ***kwargs*)

Writes an object to a file format. Automatically detects format based on filename extension.

Parameters

- **obj** (*object*) – The Python object to store in file.
- **filename** (*str*) – Filename for output of configuration.
- **config_type** (*optional[str]*) – The type of config file. Use when extension needs to differ from actual type. (e.g., .conf instead of .yml)

Returns Nothing

Note: Uses custom GDumper class to strip Python object formatting. This is not a utility function for serialization.

static load_config(*filename*, *config_type=None*, ***kwargs*)
Reads a config from file. Defaults to yaml, but will detect other config formats based on filename extension. Currently reads yaml, json, ini, csv, and text files.

Parameters

- **filename** (*str*) – Filename of configuration to be read.
- **config_type** (*optional[str]*) – The type of config file. Use when extension needs to differ from actual type. (e.g., .conf instead of .yml)
- ****kwargs** (*optional[dict]*) – keyword arguments specific to formats

Returns Dict of configuration items.

static load_yaml_string(*yaml_string*)
Reads a yaml formatted string into a dictionary

Parameters **yaml_string** (*str*) – A string containing yaml formatted text.

Returns Dictionary on success.

static load_json_string(*json_string*)
Reads a json formatted string into a dictionary

Parameters **json_string** (*str*) – A string containing json formatted text.

Returns Dictionary on success.

static load_configs(*filelist*)
Reads multiple configs from a list of filenames into a single configuration.

Parameters **filelist** (*list*) – List of configuration filenames to read.

Returns Dict of configuration items.

classmethod load_config_defaults()

classmethod set_config(*config*)
Assigns a config to the config class attribute.

Parameters **config** (*dict*) – A dictionary of configuration objects.

Returns Nothing

> **Warning:** DESTRUCTIVE. This will assign a new dictionary on top of an existing config. See update_config().

classmethod update_config(*config*)
Adds a config to the config class attribute.

Parameters **config** (*dict*) – A dictionary of configuration objects.

Returns Nothing

> **Warning:** SOMEWHAT DESTRUCTIVE. This will overwrite any previously existing objects.
>
> For example, config['thisandthat'] will overwrite cls.config['thisandthat'], but config['subconfig']['thisandthat'] will add the subconfig dictionary without overwriting cls.config['thisandthat'].

classmethod log_config(*obj*)
Writes a yaml formatted configuration to the log.

Parameters **obj** (*dict*) – The configuration object to write to log.

Returns Nothing

static get_config(*obj*)
Retrieves an object in yaml format.

Parameters **obj** (*object*) – A Python object to be converted to yaml.

Returns A yaml formatted string.

static show_config(*obj*)
Outputs a yaml formatted representation of an object on stdout.

Parameters **obj** (*object*) – A Python object to be converted to yaml.

Returns Nothing

classmethod clear_config()
Clears the config class attribute with an empty dictionary.

Returns Nothing

static show_file(*filename*)
Reads a file and prints the output.

Parameters **filename** (*str*) – Name of the file to display.

Returns Nothing

class glusto.configurable.**GDumper**(*stream*, *default_style=None*, *default_flow_style=None*, *canonical=None*, *indent=None*, *width=None*, *allow_unicode=None*, *line_break=None*, *encoding=None*, *explicit_start=None*, *explicit_end=None*, *version=None*, *tags=None*)

Bases: yaml.dumper.Dumper

Override the alias junk normally output by Dumper. This is necessary because PyYaml doesn't give a simple option to modify the output and ignore tags, aliases, etc.

ignore_aliases(*data*)

Overriding to skip aliases.

prepare_tag(*tag*)

Overriding to skip tags. e.g., !!python/object:glusto.cluster.Cluster

class glusto.configurable.**Intraconfig**

Bases: object

Class to provide instances with simple configuration utility and introspection in yaml config format.

Intended to be inherited.

Example

To inherit Intraconfig in your custom class:

```
>>> from glusto.configurable import Intraconfig
>>> class MyClass(Intraconfig):
>>>     def __init__(self):
>>>         self.myattribute = "this and that"
```

To use Intraconfig to output MyClass as yaml:

```
>>> myinstance = MyClass()
>>> myinstance.show_config()
```

update_config(*config*)

Adds a config to the config class attribute.

Parameters **config** (*dict*) – A dictionary of configuration objects.

Returns Nothing

> **Warning:** SOMEWHAT DESTRUCTIVE. This will overwrite any previously existing objects.
>
> For example, config['thisandthat'] will overwrite cls.config['thisandthat'], but config['subconfig']['thisandthat'] will add the subconfig dictionary without overwriting cls.config['thisandthat'].

show_config()

Outputs a yaml formatted representation of an instance on stdout.

Returns Nothing

get_config()

Retrieves an instance object in yaml format.

Returns A yaml formatted string.

load_config(*filename*)
Reads a yaml config from file and assigns to the config instance attribute.

Parameters **filename** (*str*) – Filename of configuration to be read.

Returns Nothing

store_config(*filename*, *config_type=None*)

Writes attributes of a class instance to a file in a config format. Automatically detects format based on filename extension.

Parameters

- **filename** (*str*) – Filename for output of configuration.
- **config_type** (*optional[str]*) – The type of config file. Use when extension needs to differ from actual type. (e.g., .conf instead of .yml)

Returns Nothing

Note: Uses custom GDumper class to strip Python object formatting. This is not a utility function for serialization.

---

glusto.connectible module

All things remote connection and local shell.

Note: Connectible is inherited by the Glusto class and not designed to be instantiated.

---

class glusto.connectible.**Connectible**
Bases: object

The class provding remote connections and local commands.

use_controlpersist = True

user = 'root'

classmethod run(*host*, *command*, *user=None*, *log_level=None*)
Run a command on a remote host via ssh.

Parameters

- **host** (*str*) – The hostname of the system.
- **command** (*str*) – The command to run on the system.
- **user** (*optional[str]*) – The user to use for connection.
- **log_level** (*optional[str]*) – only log stdout/stderr at this level.

Returns A tuple consisting of the command return code, stdout, and stderr. None on error.

Example

To run the uname command on a remote host named "bunkerhill"...

```
>>> from glusto.core import Glusto as g
>>> results = g.run("bunkerhill", "uname -a")
```

classmethod run_async(*host*, *command*, *user=None*, *log_level=None*)
Run remote commands asynchronously.

Parameters

- **host** (*str*) – The hostname of the system.
- **command** (*str*) – The command to run on the system.
- **user** (*optional[str]*) – The user to use for connection.
- **log_level** (*optional[str]*) – only log stdout/stderr at this level.

Returns An open connection descriptor to be used by the calling function. None on error.

Example

To run a command asynchronously on remote hosts named "bunkerhill" and "breedshill"...

```
>>> from glusto.core import Glusto as g
```

```
>>> command = "ls -R /etc"
>>> proc1 = g.run_async("bunkerhill", command)
>>> proc2 = g.run_async("breedshill", command)
```

```
>>> results1 = proc1.async_communicate()
>>> results2 = proc2.async_communicate()
```

This can also be used to run a command against the same system asynchronously as different users...

```
>>> command = "ls -R /etc"
>>> proc1 = g.run_async("breedshill", command, user="howe")
>>> proc2 = g.run_async("breedshill", command, user="pigot")
```

```
>>> results1 = proc1.async_communicate()
>>> results2 = proc2.async_communicate()
```

Note: run_async() runs commands asynchronously, but blocks on async_communicate() and reads output sequentially. This might not be a good fit for run-and-forget commands.

---

classmethod run_local(*command*, *log_level=None*)
Run a command on the local management system.

Parameters

- **command** (*str*) – Command to run locally.
- **log_level** (*optional[str]*) – only log stdout/stderr at this level.

Returns A tuple consisting of the command return code, stdout, and stderr.

Example

To run a command locally...

```
>>> from glusto.core import Glusto as g
>>> retcode, stdout, stderr = g.run_local("uname -a")
```

classmethod run_serial(*hosts*, *command*, *user=None*, *log_level=None*)
Sequentially runs a command against a list of hosts.

Parameters

- **hosts** (*list*) – A list of hostnames to run command against.
- **command** (*str*) – The command to run on the system.
- **user** (*optional[str]*) – The user to use for connection.
- **log_level** (*optional[str]*) – only log stdout/stderr at this level.

Returns A dictionary of tuples containing returncode, stdout, and stderr. Labeled by the host.

Example

To run a command against a list of hosts...

```
>>> from glusto.core import Glusto as g
>>> hosts = ["bunkerhill", "breedshill"]
>>> results = g.run_serial(hosts, "ls -Rail /etc")
```

classmethod run_parallel(*hosts*, *command*, *user=None*, *log_level=None*)
Runs a command against a list of hosts in parallel.

Parameters

- **hosts** (*list*) – A list of hostnames to run command against.
- **command** (*str*) – The command to run on the system.
- **user** (*optional[str]*) – The user to use for connection.
- **log_level** (*optional[str]*) – only log stdout/stderr at this level.

Returns A dictionary of tuples containing returncode, stdout, and stderr. Labeled by the host.

Example

To run a command against a list of hosts in parallel...

```
>>> from glusto.core import Glusto as g
>>> hosts = ["bunkerhill", "breedshill"]
>>> results = g.run_serial(hosts, "ls -Rail /etc")
```

classmethod upload(*host*, *localpath*, *remotepath*, *user=None*)
Uploads a file to a remote system.

Parameters

- **host** (*str*) – Hostname of the remote system.
- **localpath** (*str*) – The source path for the file on the local system.

- **remotepath** (*str*) – The target path on the remote server.
- **user** (*optional[str]*) – The user to use for the remote connection.

Returns None on failure.

classmethod download(*host*, *remotepath*, *localpath*, *user=None*)
Downloads a file from a remote system.

Parameters

- **host** (*str*) – Hostname of the remote system.
- **remotepath** (*str*) – The source path on the remote server.
- **localpath** (*str*) – The target path for the file on the local system.
- **user** (*optional[str]*) – The user to use for the remote connection.

Returns None on failure.

classmethod transfer(*sourcehost*, *sourcefile*, *targethost*, *targetfile*, *user=None*)
Transfer a file between remote systems (scp) Requires keys to be set up between remote systems.

Parameters

- **sourcehost** (*str*) – Hostname of the remote system copying from.
- **sourcefile** (*str*) – The source path on a remote system.
- **targethost** (*str*) – Hostname of the remote system copying to.
- **targetfile** (*str*) – The target path for the file on a remote system.
- **user** (*optional[str]*) – The user to use for the remote connection.

Returns Nothing

classmethod ssh_list_connections()
Display the list of existing ssh connections on stdout.

classmethod ssh_get_connections()
Retrieves the dictionary of ssh connections.

Returns A dictionary of ssh connections.

classmethod ssh_close_connection(*host*, *user=None*)
Close an SshMachine connection.

Parameters

- **host** (*str*) – Hostname of the system.
- **user** (*optional[str]*) – User to use for connection.

Returns Nothing

classmethod ssh_close_connections()
Close all ssh connections.

Parameters **None** –

Returns Nothing

classmethod ssh_set_keyfile(*keyfile*)

classmethod ssh_get_keyfile()

glusto.core module

The brains of the Glusto toolset.

Glusto inherits from multiple classes providing configuration, remote connection, and logging functionality and presents them in a single global Class object. Glusto also acts a global class for maintaining state across multiple modules and classes.

Example

To use Glusto in a module, import at the top of each module leveraging the glusto tools.:

```
from glusto.core import Glusto as g
```

class glusto.core.**Glusto**

Bases: *glusto.configurable.Configurable*, *glusto.connectible.Connectible*, *glusto.colorfiable.Colorfiable*, *glusto.loggable.Loggable*, *glusto.templatable.Templatable*, *glusto.unittestable.Unittestable*, *glusto.restable.Restable*, *glusto.rpycable.Rpycable*, *glusto.carteplex.Carteplex*

The locker for all things Glusto.

log = <logging.Logger object>

glusto.loggable module

All things logging.

Note: Loggable is inherited by the Glusto class and not designed to be instantiated.

---

class glusto.loggable.**Loggable**

Bases: object

The class providing logging functionality.

handler_counter = 1

classmethod create_log(*name='glustolog'*, *filename='/tmp/glusto.log'*, *level='DEBUG'*, *log_format=None*, *allow_multiple=False*)

Creates a log object using the Python "logging" module.

Parameters

- **name** (*optional[str]*) – The reference name for the logging object. Defaults to "glustolog".
- **filename** (*optional[str]*) – Fully-qualified path and filename. Defaults to "/tmp/glusto.log".
- **level** (*optional[str]*) – The minimum log level. Defaults to "INFO".
- **allow_multiple** (*bool*) – Can multiple logfiles exist? Tells method whether to create a new handler or wipe existing before creating a new handler.

Returns A logging object.

classmethod add_log(*logobj*, *filename='/tmp/glusto.log'*, *level='INFO'*, *log_format=None*)

Add a logfile to the logobj

Parameters

- **logobj** (*object*) – A logging object.
- **filename** (*optional[str]*) – Fully-qualified path and filename. Defaults to "/tmp/glusto.log".
- **level** (*optional[str]*) – The minimum log level. Defaults to "INFO".

classmethod remove_log(*logobj*, *name=None*)
Remove a log handler from a logger object.

Parameters

- **logobj** (*object*) – A logging object.
- **name** (*optional[str]*) – The name of the log handler to remove. If None, will remove all log handlers from the logger.

classmethod show_logs(*logobj*)
Show a list of log handlers attached to a logging object

Parameters **logobj** (*object*) – A logging object.

classmethod disable_log_levels(*level*)
Disable level (and lower) across all logs and handlers. Handy if a method continually spams the logs. Use reset_log_level() to return to normal logging.

Note: See Python logging module docs for more information.

Parameters **level** (*str*) – String name for the top log level to disable.

Returns Nothing

classmethod reset_log_levels()
Reset logs to current handler levels. Convenience method to undo disable_log_level()

Parameters **None** –

Returns Nothing

classmethod set_log_level(*log_name*, *handler_name*, *level*)
Set the log level for a specific handler. Use show_logs() to get the list of log and handler names.

Parameters

- **log_name** (*str*) – The name of the log.
- **handler_name** (*str*) – The name of the specific log handler.
- **level** (*str*) – The string representation of the log level.

Returns Nothing

classmethod set_log_filename(*log_name*, *handler_name*, *filename*)
Change the logfile name for a specific handler. Use show_logs() to get the list of log and handler names.

Parameters

- **log_name** (*str*) – The name of the log.
- **handler_name** (*str*) – The name of the specific log handler.
- **filename** (*str*) – The path/filename to log to.

Returns Nothing

Note: Nothing in logging docs mentions this method (close and set baseFilename) over removing the handler and creating a new handler with the new filename. Research and correct if needed. Caveat emptor.

---

classmethod clear_log(*log_name*, *handler_name*)
Empties an existing log file

Parameters

- **log_name** (*str*) – The name of the log.
- **handler_name** (*str*) – The name of the specific log handler.

Returns Nothing

glusto.main module

Glusto CLI wrapper

glusto.main.**handle_configs**(*config_list*)
Load default and user-specified configuration files

glusto.main.**main**()
Entry point console script for setuptools.

Provides a command-line interface to Glusto.

Currently does nothing useful, but plan to wrap Glusto functionality in a CLI interface that can be injected into shell scripts, etc.

Example

glusto run hostname.example.com "uname -a"

glusto.restable module

All things REST API related.

Note: Restable is inherited by the Glusto class and not designed to be instantiated.

---

class glusto.restable.**Restable**
Bases: object

The class providing REST API functionality.

classmethod rest_get(*url*)
Submit a REST API GET request.

Parameters **url** (*str*) – The HTTP protocol standard url for the request.

Returns The HTTP protocol standard return code. Zero on success. response output (str): The output text from the response. response error (str): The error text on failure.

Return type returncode (int)

classmethod rest_post(*url*, *data*)
Submit a REST API POST request.

Parameters

- **url** (*str*) – The HTTP protocol standard url for the request.
- **data** (*dict*) – A dictionary of key:value pairs

Returns The HTTP protocol standard return code. Zero on success. response output (str): The output text from the response. response error (str): The error text on failure.

Return type returncode (int)

classmethod rest_put(*url*, *data*)
Submit a REST API PUT request.

Parameters

- **url** (*str*) – The HTTP protocol standard url for the request.
- **data** (*dict*) – A dictionary of key:value pairs

Returns The HTTP protocol standard return code. Zero on success. response output (str): The output text from the response. response error (str): The error text on failure.

Return type returncode (int)

classmethod rest_delete(*url*, *data*)
Submit a REST API DELETE request.

Parameters

- **url** (*str*) – The HTTP protocol standard url for the request.
- **data** (*dict*) – A dictionary of key:value pairs

Returns The HTTP protocol standard return code. Zero on success. response output (str): The output text from the response. response error (str): The error text on failure.

Return type returncode (int)

glusto.rpycable module

All things rpyc connection.

Note: Rpycable is inherited by the Glusto class and not designed to be instantiated.

Warning: Rpyc breaks in mixed Python 2.x/3.x environments. When using rpyc, you will only be able to successfully make rpyc calls against a system running the same version of Python. (see rpyc module install docs for more information)

class glusto.rpycable.**Rpycable**
Bases: object

classmethod rpyc_get_connection(*host*, *user=None*, *instance=1*)
Setup and cache a connection via rpyc.

Parameters

- **host** (*str*) – The hostname or IP of the remote system.
- **user** (*str*) – A user on the remote system. Default: root
- **instance** (*int*) – The number of the instance when multiple connections are used.

Returns A new or cached rpyc connection object.

classmethod rpyc_create_connections(*hosts*, *user=None*, *num_instances=1*)
Setup and cache multiple connections via rpyc.

Parameters

- **host** (*str*) – The hostname or IP of the remote system.
- **user** (*str*) – A user on the remote system. Default: root
- **num_instances** (*int*) – The number of the instances to create.

Returns Nothing.

classmethod rpyc_get_connections()
Get the connection dictionary.

Parameters **None** –

Returns The dictionary of rpyc connections.

classmethod rpyc_list_connections()
Display the list of existing ssh connections on stdout.

Parameters **None** –

Returns Nothing

classmethod rpyc_list_deployed_servers()

classmethod rpyc_check_connection(*host*, *user=None*, *instance=1*)
Check whether a connection is open or closed.

Parameters

- **host** (*str*) – The hostname or IP of the remote system.
- **user** (*str*) – A user on the remote system. Default: root
- **instance** (*int*) – The number of the instance when multiple connections are used.

Returns Nothing

classmethod rpyc_ping_connection(*host*, *user=None*, *instance=1*)
Ping an rpyc connection.

Parameters

- **host** (*str*) – The hostname or IP of the remote system.
- **user** (*str*) – A user on the remote system. Default: root
- **instance** (*int*) – The number of the instance when multiple connections are used.

Returns True if pingable. False if does not ping.

classmethod rpyc_close_connection(*host=None*, *user=None*, *instance=1*)
Close an rpyc connection.

Parameters

- **host** (*str*) – The hostname or IP of the remote system.

- **user** (*str*) – A user on the remote system. Default: root
- **instance** (*int*) – The number of the instance when multiple connections are used.

Returns Nothing.

classmethod rpyc_close_connections()
Close all rpyc connections.

Parameters **None** –

Returns Nothing

classmethod rpyc_close_deployed_servers()
Close all deployed server connections.

Parameters **None** –

Returns Nothing

classmethod rpyc_close_deployed_server(*host=None*, *user=None*)
Close a deployed server connection.

Parameters

- **host** (*str*) – The hostname or IP of the remote system.
- **user** (*str*) – A user on the remote system. Default: root

Returns Nothing.

classmethod rpyc_define_module(*connection*, *local_module*)
Define a local module on the remote system

Parameters

- **connection** (*obj*) – An rpyc connection object.
- **local_module** (*obj*) – The module object being defined on the remote.

Returns A module object representing the local module defined on remote

glusto.templatable module

All things Jinja templates.

Note: Templatable is inherited by the Glusto class and not designed to be instantiated.

class glusto.templatable.**Templatable**
Bases: object

The class providing Jinja template functionality.

static render_template(*template_filename*, *template_vars={}*, *output_file=None*, *search-path=None*)
Render a template into text file

Parameters

- **template_filename** (*str*) – Fully qualified template filename.
- **template_vars** (*dict*) – A dictionary of variables.
- **output_file** (*str*) – Fully qualified output filename.

- **searchpath** (*str*) – The root path to begin file searches. Default is the current path.

Returns String containing template rendering result if successful.

glusto.unittestable module

All things unittest.

Note: Unittestable is inherited by the Glusto class and not designed to be instantiated.

class glusto.unittestable.**Unittestable**
Bases: object

The class providing unittest functionality.

static load_tests(*test_class*, *loader*, *ordered_testcases*)

Load specified tests in a order followed by the remaining tests in the test_class.

Parameters

- **test_class** (*object*) – The TestCase class object with test methods.
- **loader** (*object*) – The loader object passed from unittests to calling load_tests function.
- **ordered_testcases** (*list*) – list of testcase method names in order to be run.

Returns Returns a unittest.TestSuite() containing loaded tests.

Note: This feature requires Python2.7 or higher

2.1.2 Module contents

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

Symbols

A

B

C

D

G

H

I

L

M

N

P

R

S

T

U

W

Y