gluetool Documentation

Release 1.25

mvadkerti@redhat.com

Apr 15, 2021

Contents

Gluetool 1.25 (1.25)

The `gluetool` command-line tool is an automation tool constructing a sequential pipeline on the command-line. It is able to implement any sequential process when it's divided into modules with minimal interaction between them, gluing them together on the command-line to form a pipeline. It is implemented as an open *gluetool framework*.

The cool thing about having the pipeline on the command-line is that it can be easily copy-pasted to your local shell for debugging/development, and such pipeline can be easily customized by changing options of the modules when needed.

The tool optionally integrates with Sentry.io error logging platform for reporting issues, very useful when running `gluetool` pipelines at larger scales.

CHAPTER 1

Installation

If you want to install `gluetool` on your machine, you have two options:

For just using `gluetool`, you can install package from pip:

```
pip install gluetool
```

If you want to change the code of `gluetool` and use your copy, follow our *Development* readme in the project root folder.

CHAPTER 2

Table of contents

2.1 The `gluetool` framework

The `gluetool` framework is a command-line centric, modular Python framework. It allows to quickly develop modules which can be executed on the command-line by the `gluetool` command, forming a sequential pipeline.

2.1.1 Architecture

The `gluetool` framework was created with various features in mind. These should provide an easy and straightforward way how to write, configure and execute modules.

Generic core

The *core of the framework* is completely decoupled from the modules and their specific functionality. It could be used for implementation of other tools.

Modules

See How to: gluetool modules for more information about `gluetool` modules.

Configuration

The framework provides an easy way how to define configuration for the framework and the modules. The configuration files use the ConfigParser format. The configuration option key in the config file is the long option string as defined in the *`options`* dictionary.

`gluetool` searches for configuration files in several directories, in specific order, and it opens relevant files in all of them if they are present there, with, given the order in which directories are being examined, values from later files replace those from the former ones.

Following directories are examined (the list is defined as `MODULE_CONFIG_PATHS`):

- `/etc/gluetool.d`
- `~/.gluetool.d`
- `./.gluetool.d`

The configuration directory layout is:

```
~/.gluetool.d/
├── gluetool                          - gluetool configuration
└── config                         - per module configurations, one file per␣
↪unique module name
    ├── <module_config_file>
    └── ...
```

Note: Note that this directory can be used for storing additional configuration files and directories according to the needs of the modules.

Configuration of `gluetool`

The *gluetool* file defines the default options for the `gluetool` command itself. It can be used to define defaults for any of the supported options. The options need to be defined in the `[default]` section. You can view all the supported options of `gluetool` by running the command `gluetool -h`. For example, to enable the debug mode by default, we can use this configuration

```
$ cat ~/.gluetool.d/gluetool
[default]
debug = True
```

Modules Configuration

The *config* subdirectory can define a default configuration for each module. The configuration filename must be the same as the module's `name`. All options must be defined in the `[default]` section of the configuration file. You can view the module available options by running `gluetool <module> -h`, e.g. `gluetool openstack -h` for the (hypothetical) `openstack` module.

Below is an example of configuration for this `openstack` module.

```
$ cat ~/.gluetool.d/config/openstack
[default]
auth-url = https://our-instance.openstack.com:13000/v2.0
username = batman
password = YOUR_SECRET_PASSWORD
project-name = gotham_ci
ssh-key = ~/.ssh/id_rsa
ssh-user = root
key-name = id_rsa
ip-pool-name = 10.8.240.0
```

Shared Functions

Shared functions are the only way how modules can share data to the subsequent modules on the command-line pipeline. Each module can define a shared function via the *shared_functions* list. The available shared functions then can be easily called from any subsequent module being executed via the *shared* method.

To list all shared functions provided by the available modules, use the gluetool's *-L* option

```
$ gluetool -L
```

Shared function names are unique, but different modules can expose the same shared function. This is useful for generalization, where for example different modules can provide a `provision` - or any other name your team agreed on - shared function returning a list of provisioned machines from different virtualization providers.

Shared functions can have arguments and they behave the same way as ordinary Python functions.

> **Note:** The documentation of the shared function is generated automatically from the docstring of the method and displayed in the help of the module. As an example, see the help of the `dep-list` module by running `gluetool dep-list -h`. You'll see the module provides `prepare_dependencies` shared function.

Uniform Logging

The `gluetool` framework provides uniform logging. Modules can use their own `info`, `warn`, `debug` and `verbose` methods to log messages on different log levels. The log level can be changed using the `-d/--debug` and `-v/--verbose` options of the `gluetool` command.

> **Note:** Note that the `-d/--debug` and `-v/--verbose` options will enable the logging after parsing the command line and configuration. To enable the debug mode as early as possible, i.e. during the initialization of the logging system, export the variable `GLUETOOL_DEBUG` to any value.

2.2 `gluetool` modules

All application specific functionality should be placed into modules. Modules are defined in one Python file and the module class must inherit from the class *gluetool.glue.Module*.

2.2.1 Importing of modules

The framework searches for modules in the module path(s). By default the module path is `gluetool/modules` in the project's root directory. You can override the module path with the `--module-path` option on the command line or via the *gluetool configuration*. The search algorithm tries to be clever about the import. It firstly parses the syntax tree of all `*.py` files it finds in the modules path(s) and imports it only if it finds a class definition which inherits from the *gluetool.glue.Module* class.

> **Note:** The module importing logic requires that you always inherit from the `gluetool.glue.Modules` class or your module will not be imported. So for example, if you want to extend an existing module `Koji` to `MyKoji`, you need to use:

```
class MyKoji(Koji, Module):
...
```

2.2.2 Basic attributes

Name and description

Module must define one or more unique names with the class variable `name`. This name identifies the module on the command line. For more information about modules providing multiple names see the section *Modules with multiple names*.

Module should also define **description** with the class variable *`description`*, which will be displayed in the module listing, i.e. `gluetool -l`.

Options

Modules can define an *`options`* dictionary, which defines their command line arguments and also the *module configuration* at once. Modules can use their *option method* <*gluetool.glue.Module.option*> to access the option value. The method returns `None` if option does not exist or it's value is not defined.

> **Note:** The `gluetool` framework currently provides support only for named options/arguments. It is strongly advised to use named options only.

A module option value can be specified in 3 ways and in this precedence (later replaces the previously defined value):

- value defined by the `default` key in the option's dictionary
- value read from the *module configuration* <*modules_configuration*>
- value read from the module's command line argument

The first two possibilities are used to define the option defaults. The command line argument value is used to override these if needed.

Modules can define a list of required options using the `required_options` class variable. The required options specify which options need to be specified when executing the module.

> **Note:** It is advised to use `required_options` list instead of argparse's required option because the latter will only require the option specified on the command line, while the `required_options` list also takes into account values read from the *module configuration* <*modules_configuration*>.

2.2.3 Basic methods

Modules usually want to implement three main *`Module`* methods - *`sanity`*, *`execute`* and *`destroy`*.

The *`sanity method`* is called after parsing the command line options and the configuration files before any module is executed. The usual use-case for using the sanity method is to do additional actions before any module is executed.

The *`execute method`* is the main entrypoint for the module. This method usually implements the module's main functionality.

The *destroy method* is called after the execution of all the modules specified in the pipeline. The destroy methods are called in the opposite direction as the modules are executed and the methods are called also if the execution of the pipeline did not finish (e.g. a module aborted the execution).

2.2.4 Shared functions

See the *framework's documentation* for introduction into shared functions.

A module can define any number of shared functions by listing their name as a string in the `gluetool.glue.Module.shared_functions` list. The shared functions are made available to other modules after the module has been executed. This makes it possible for the module to redefine the previously defined shared functions with their own version.

Here is an example of a simple module that exposes myapi shared function and takes one optional argument specifying the api version.

```
import gluetool

class MyApiModule(gluetool.Module):
    name = 'myapi'

    shared_functions = ['myapi']

    def myapi(self, api_version=1):
        return 'My Api version: {}'.format(api_version)

    def execute(self):
        self.info('hello world')
```

If you want to call a shared function from an other module, just use the *shared* method and provide the name of the function as a string, for example in the above example, you would call:

```
self.shared('myapi')
```

Note: `shared()` actually **calls** the shared function `myapi` from the MyApiModule in this case.

If you would like to pass additional arguments to the called shared function, just pass it as an argument to the shared function, e.g.:

```
self.shared('myapi', api_version=2)
```

By design, more recently registered shared function replaces older ones of the same name, making them inaccessible. When calling shared function `foo`, the one added by the module further in the pipeline gets called. Should you need to call the older version of `foo`, the one replaced by the current instance, you can use the *overloaded_shared* method. It can be used to simulate a chain of `super()` calls in Python classes, giving "parent"-ish modules, listed sooner in the pipeline, a say.

For example, imagine two "publishing" modules - one sends messages to "alpha", the other one to "omega". Both "implement the interface" by providing a shared function with the same name, `publish`, and both call older version of `publish` shared function when they're done with their own work, to give modules listed sooner in the pipeline a chance to "publish" as well. With this cooperation, it does not matter how many publishing modules you have in the pipeline or what's their order as long as each of them calls older version of `publish`. User of such modules, `publish-message`, then calls `publish` shared function, leaving the rest to them.

```
import gluetool

class PublishAlpha(gluetool.Module):
    name = 'publish-alpha'
    shared_functions = ['publish']

    def publish(self, message):
        self.info("publishing to alpha '{}'".format(message))
        self.overloaded_shared('publish', message)
```

```
import gluetool

class PublishOmega(gluetool.Module):
    name = 'publish-omega'
    shared_functions = ['publish']

    def publish(self, message):
        self.info("publishing to omega '{}'".format(message))
        self.overloaded_shared('publish', message)
```

```
import gluetool

class Publish(gluetool.Module):
    name = 'publish-message'
    options = {
        'message': {
            'help': 'Message to publish'
        }
    }
    required_options = ['message']

    def execute(self):
        self.shared('publish', self.option('message'))
```

Here is an example of the execution of the above modules:

```
$ gluetool publish-alpha publish-omega publish-message --message test
[14:05:11] [+] [publish-omega] publishing to omega 'test'
[14:05:11] [+] [publish-alpha] publishing to alpha 'test'
```

2.2.5 Examples

A minimal module

Adding a new gluetool module is very simple. This is a minimal module that just prints 'hello world':

```
from gluetool import Module

class MinimalModule(Module):
    name = 'minimal'
    description = 'A minimal module'

    def execute(self):
        self.info('hello world')
```

Drop this module into the module path and try to run the module via:

```
$ gluetool minimal
```

2.2.6 Advanced development techniques

Modules with multiple names

Modules can actually define multiple names under which they can be called on the command line. This is very useful, if you have the same plugin providing access to various instances of the same system, or a system that can be used using the same API. An example can be a postgresql module, that can be also used to connect to an Teiid instance. The benefit from having the same module appearing with different name is that you can define specific configuration for each module incarnation.

```
from gluetool import Module

class Posgresql(Module):
    name = ('postgresql', 'teiid')
    ....
```

2.3 How to: `gluetool` tests

This text is a (hopefully complete) list of best practices, dos and don'ts and tips when it comes to writing tests for `gluetool` APIs, modules and other code. When writing - or reviewing - `gluetool` tests, please adhere to these rules whenever possible.

Note: These rules are not cast in stone - when we find out some are standing in our way to the most readable and usable documentation, let's just discuss the change and change what must be changed.

2.3.1 py.test

`gluetool` uses `py.test` framework for its test and tox to automate the running of the tests. If you're not familiar with these tools, please see following links to get some idea:

- py.test
- tox

Also inspecting existing tests and `tox.ini` is a good way to find out how to do something, e.g. add new coverage for your module.

2.3.2 How to run tests?

Static analysis is using coala in docker, so for full test, you need to have docker daemon running.

You can run all tests using `tox`:

```
tox -e py27
```

If you want to skip coala analysis so you don't need docker, you can run

```
tox -e 'py-{unit-tests,static-analysis,doctest}'
```

Tox also accept additional options:

```
python setup.py test -a "--option1 --option2=value"

tox -e py27 -- --option1 --option2=value
```

2.3.3 How to see code coverage?

By default, coverage measurement is disabled. To enable it, pass following options to the test runner of your choice:

```
--cov=gluetool --cov-report=html:coverage-report
```

With these options, coverage will be enabled and when test run finishes, the coverage report (in HTML) will be created in `coverage-report` directory. Simply open `coverage-report/index.html` in your browser then.

Note: Coverage data are stored in `.coverage` file - if you'd like to use `coverage` utility to create additional reports or filter the output to better suit your needs, feel free to do so, nothing stands in your way :)

2.3.4 Module tests should be in the same file

Tests dealing with a single module should be packed in the same file.

2.3.5 Test function tests one thing/code path

Avoid the temptation to put more different tests into a single test function. Test function should test a single feature or a code path. If you're concerned about repeating setup/teardown code a lot, learn about fixtures bellow.

2.3.6 Use `assert`

`py.test` prefers to use `assert` keyword to actually test values, and it promotes its use by providing really nice and helpful formatting of failures, with pointers to places where the actual values differ from expected ones.

Sometimes it's very useful to create a helper function that checks complex response, data or object state, using multiple lower-level `assert` instances.

2.3.7 Use fixtures

> The purpose of test fixtures is to provide a fixed baseline upon which tests can reliably and repeatedly execute. pytest fixtures offer dramatic improvements over the classic xUnit style of setup/teardown functions.
>
> —py.test documentation

They don't lie, it's definitely worth the effort. Pretty much every test of a module's code begins with "get a fresh instance of a module-under-test". You can call some function to create this instance, or you can use a fixture and simply accept this instance as a argument of your test function. And so on.

```
# every test function gets its own instance of gluetool.glue and the module it's␣
↪testing
from . import create_module

@pytest.fixture(name='module')
def fixture_module():
    return create_module(gluetool.modules.helpers.ansible.Ansible)

def test_sanity(module, tmpdir):
    glue, _ = module

    assert glue.has_shared('run_playbook') is True
```

Session fixtures belong to `tests/conftest.py`.

2.3.8 Check exception messages with `match`

Use `pytest.raises()` parameter `match` to assert exception messages whenever possible:

```
with pytest.raises(Exception, match=r'dummy exception'):
    foo()
```

Be aware that `match` value is actually a regular expression used to match exception's message, therefore use Python's raw strings, prefixed with `r`.

2.3.9 Don't be afraid of monkeypatching

It helps a lot with failure injection, with observing whether your code calls other functions it's expected to call, and other useful tricks. And all patches are undone when your test function returns.

```
# If OSEror pops up, run_command should raise GlueError and re-use message from the␣
↪original exception
def faulty_popen_enoent(*args, **kwargs):
    raise OSError(errno.ENOENT, '')

monkeypatch.setattr(subprocess, 'Popen', faulty_popen_enoent)

with pytest.raises(gluetool.GlueError, match=r"^Command '/bin/ls' not found$"):
    run_command(['/bin/ls'])
```

2.3.10 When your attempts lead to messy tests, cosider refactoring of the tested code

This can happen very often - you'd like to test a method which is way too complex, and the result is huge pile of setup/teardown code, unreadable asserts and even more complicated ways to convince the tested function to take different path, e.g. when it comes to injecting errors into its flow. In such case, consider refactoring the tested code - it's possible it could be rewritten to more separate pieces of code (main function & several helpers) which could greatly improve the list of options you have, and it may even lead to more readable code.

2.3.11 `MagicMock` is very handy tool

Don't be afraid to use `MagicMock` - its `return_value` and `side_effect` parameters can help a lot when it comes to mocking mocking functions returning prepared values or raising exceptions. E.g.

```
monkeypatch.setattr(library, 'library_function', MagicMock(side_effect=Exception))
```

when `library.library_function` gets called, it will raise an exception. If you need to raise an exception with specific arguments, pass a helper function as a side effect:

```
def throw(*args, **kwargs):
    # pylint: disable=unused-argument

    raise Exception('simply bad request')

monkeypath.setattr(library, 'library_function', MagicMock(side_effect=throw))
```

Instead of mocking a whole function, use `MagicMock`'s `return_value`:

```
monkeypatch.setattr(foo, 'bar', MagicMock(return_value=some_known_object))
```

is way more readable than:

```
def foo():
   return some_known_object

monkeypach.setattr(foo, 'bar', foo)
```

Should you need more action when it comes to returned value (computing it on the fly), patching with custom function is absolutely acceptable.

2.4 How to: `gluetool` documentation

This text is a (hopefully complete) list of best practices, dos and don'ts and tips when it comes to writing documentation of `gluetool` APIs, options and other documents. When writing - or reviewing - `gluetool` docs, please adhere to these rules whenever possible.

> **Note:** These rules are not cast in stone - when we find out some are standing in our way to the most readable and usable documentation, let's just discuss the change and change what must be changed.

2.4.1 RST

`gluetool` uses *reStructuredText* for its docstrings and documentation. If you're not familiar with this markup language, please see following links to get some idea:

- RST primer
- Other helpful directives
- Referencing Python objects

Also inspecting sources - and the resulting documentation - is a good way to find out how to do something, e.g. how to use links to external documents.

2.4.2 How to generate HTML documentation locally

- Just run ansible playbook generate-docs.yml which can be found in the root directory of the project

```
/usr/bin/ansible-playbook generate-docs.yml
```

You documentation awaits you at `docs/build/html/index.html`.

2.4.3 Write multi-line docstrings

```
"""
Foo bar
"""
```

Most of the time, functions and classes take parameters, return values, etc. Unless there's a really good reason against that, e.g. in the case of very simple helpers, multi-line docstring should be the goal, allowing for detailed description of the documented API.

2.4.4 Every module must have a description

Short, one or two sentences describing the purpose of the module.

2.4.5 Every shared function must be documented

Shared functions are **the** API of `gluetool` modules. Their docstrings are used to generate HTML docs or command-line help, therefore it's crucial to document their usage.

2.4.6 Every module must be documented

Longer, detailed description of module's goal, provided services, required resources and possible pitfalls.

2.4.7 Check whether the documentation is up-to-date

Make sure the documentation describes the actual state of the affairs. E.g. developer could have changed semantics of a command-line option, or added another one that changed a behavior slightly, and forgot to update its help string.

> **Note:** Outdated documentation is probably even worse than no documentation at all. It leads reader to false assumptions which lead to anger. Anger leads to hate. Hate leads to suffering. When revieweing documentation, please take special care of making sure it's up-to-date.

2.4.8 Default values of parameters

If the parameter is a keyword parameter, having its default value right in function signature, Sphinx will use this information and add it to the output.

```
def foo(bar=None):
    """
    ...
    :param str bar: if set, it's printed to ``stdout``.
    """
```

If the default value only means *unspecified value* and function replaces it internally with the actual default value that cannot be declared in function signature (e.g. it's mutable object, or it's retreived from another API), then it should be noted in parameter description:

```
def foo(bar=None):
    """
    ...
    :param dict bar: if set, it's passed to Baz. Empty ``dict`` is used by␣
↪default.
    """

    bar = bar or {}
```

2.4.9 Reference what can be referenced

Hyperlinks are good. Hyperlinks are useful. Hyperlinks save lives. Sphinx makes it easy to reference Python stuff, you can find more information here.

It is not necessary to reference types of parameters when documented by `:param <type> name` directive - Sphinx will attempt to create correspondign link automagically.

2.4.10 Return values

Sphinx provides two directives for return value documentation:

- `:returns:` * describe the return value, you can include its type if it fits naturally into your text * if you include type, you must reference it manually, Sphinx won't do it
- `:rtype:` * type - and only a type - of the return value * creates a link to the type - it's not necessary to reference it with `:py:...`

If you can fit return value type into your description of the return value, then use `:returns:`. Most of the time you probably can, that makes `:rtype:` a bit redundant but sometimes it can be useful.

```
"""
...
:returns: :py:class:`gluetool.utils.ProcessOutput` instance whose attributes␣
↪contain data returned by the process.
"""
```

2.4.11 Code and data examples

If it'd be helpful, use an example, e.g. to show possible config file structure or to provide better idea about complex return type. For this, `.. code-block:: <language>` can be very useful:

This is what a config file may look like:

```
---
foo:
  - bar
  - baz
```

Note: Be careful of the alignment of text bellow the `code-block` directive - it starts at the same column as the `code-block` string, with one empty line separating them.

2.4.12 Style

- Use backquotes to mark literals
  - module names: `guest-setup`, `jenkins`, ...
  - commands: `jenkins-jobs`, `/bin/ls`, ...
  - when mentioning it, `gluetool` itself
  - basic Python types: `dict`, `list`, ...
  - command-line options: `--help`, `--pattern-map`, ...
- Sentences should start with capital letter and end with a full stop. This applies to parameter descriptions as well.
- Directives like `:param` can spread to multiple lines - in such case, indent the second and following lines by a single `<TAB>`.

2.5 Development

2.5.1 Environment

Before moving on to the actual setup, there are few important notes:

- **The only supported and (sort of tested) way of installation and using ‘‘gluetool‘‘ is a separate virtual environment!** It may be possible to install `gluetool` directly somewhere into your system but we don't recommend that, we don't use it that way, and we don't know what kind of hell you might run into. Please, stick with `virtualenv`.
- The tested distributions (as in “we're using these”) are either recent Fedora, RHEL or CentOS. You could try to install `gluetool` in a different environment - or even development trees of Fedora, for example - please, make notes about differences, and it'd be awesome if your first merge request could update this file :)

2.5.2 Requirements

To begin digging into `gluetool` sources, there are few requirements:

- `virtualenv` utility
- `ansible-playbook`

2.5.3 Installation

1. Create a virtual environment

```
virtualenv -p /usr/bin/python2.7 <virtualenv-dir>
. <virtualenv-dir>/bin/activate
```

2. Clone `gluetool` repository - your working copy

```
git clone github:<your username>/<your fork name>
cd gluetool
```

3. Install `gluetool`

```
python setup.py develop
```

4. (optional) Activate Bash completion

```
gluetool --module-path gluetool_modules/ bash-completion > gluetool-bash-completition
mv gluetool-bash-completition $VIRTUAL_ENV/bin/gluetool-bash-completition
echo "source $VIRTUAL_ENV/bin/gluetool-bash-completition" >> $VIRTUAL_ENV/bin/activate
```

To activate bash completion immediately, source the generated file. Otherwise, it'd start working next time you'd activate your virtualenv.

```
. ./gluetool-bash-completition
```

5. Add configuration

`gluetool` looks for its configuration in a local directory (among others), in `./.gluetool.d` to be specific. Add configuration for the modules according to your preference.

Now every time you activate your new virtualenv, you should be able to run `gluetool`:

```
gluetool -h
usage: gluetool [opts] module1 [opts] [args] module2 ...

optional arguments:
...
```

2.5.4 Test suites

The test suite is governed by `tox` and `py.test`. Before running the test suite, you have to install `tox`:

```
pip install tox
```

Tox can be easily executed by:

```
tox
```

Tox also accepts additional options which are then passed to `py.test`:

```
tox -- --cov=gluetool --cov-report=html:coverage-report
```

Tox creates (and caches) virtualenv for its test runs, and uses them for running the tests. It integrates multiple different types of test (you can see them by running `tox -l`).

2.5.5 Documentation

Auto-generated documentation is located in `docs/` directory. To update your local copy, run these commands:

```
ansible-playbook ./generate-docs.yml
```

Then you can read generated docs by opening `docs/build/html/index.html`.

CHAPTER 3

gluetool API

3.1 gluetool.glue module

class gluetool.glue.**ArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *version=None*, *parents=[]*, *formatter_class=<class 'argparse.HelpFormatter'>*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*)

Bases: argparse.ArgumentParser

Pretty much the argparse.ArgumentParser, it overrides just the argparse.ArgumentParser.error() method, to catch errors and to wrap them into nice and common *GlueError* instances.

The original prints (for us) useless message, including the program name, and raises SystemExit exception. Such action does not provide necessary information when encountered in Sentry, for example.

error(*message*)

Must not return - raising an exception is a good way to "not return".

Raises ***gluetool.glue.GlueError*** – When argument parser encounters an error.

class gluetool.glue.**CallbackModule**(*name*, *glue*, *callback*, **args*, ***kwargs*)

Bases: mock.MagicMock

Stand-in replacement for common **:py:'Module'** instances which does not represent any real module. We need it only to simplify code pipeline code - it can keep working with Module-like instances, since this class mocks each and every method, but calls given callback in its execute method.

Parameters

- **name** (*str*) – name of the pseudo-module.
- **glue** (Glue) – Glue instance governing the pipeline this module is part of.
- **callback** (*callable*) – called in the execute method. Its arguments will be glue, followed by remaining positional and keyword arguments (args and kwargs).

- **args** (*tuple*) – passed to `callback`.
- **kwargs** (*dict*) – passed to `callback`.

_get_child_mock(***kw*)
Create the child mocks for attributes and return value. By default child mocks will be the same type as the parent. Subclasses of Mock may want to override this to customize the way child mocks are made.

For non-callable mocks the callable variant will be used (rather than any custom subclass).

check_required_options()

destroy(*failure=None*)

execute()

sanity()

class gluetool.glue.**Configurable**(*logger*)
Bases: *gluetool.log.LoggerMixin*, `object`

Base class of two main `gluetool` classes - *gluetool.glue.Glue* and *gluetool.glue.Module*. Gives them the ability to use *options*, settable from configuration files and/or command-line arguments.

Variables **_config** (*dict*) – internal configuration store. Values of all options are stored here, regardless of them being set on command-line or by the configuration file.

classmethod _create_args_parser(***kwargs*)
Create an argument parser. Used by Sphinx to document "command-line" options of the module - which are, by the way, the module options as well.

Parameters **kwargs** (*dict*) – Additional arguments passed to `argparse.ArgumentParser`.

_dryrun_allows(*threshold*, *msg*)
Check whether current dry-run level allows an action. If the current dry-run level is equal of higher than `threshold`, then the action is not allowed.

E.g. when action's `threshold` is *DryRunLevels.ISOLATED*, and the current level is *DryRunLevels.DRY*, the action is allowed.

Parameters

- **threshold** (`DryRunLevels`) – Dry-run level the action is not allowed.
- **msg** (*str*) – Message logged (as a warning) when the action is deemed not allowed.

Returns `True` when action is allowed, `False` otherwise.

static _for_each_option(*callback*, *options*)
Given dictionary defining options, call a callback for each of them.

Parameters

- **options** (*dict*) – Dictionary of options, in a form `option-name: option-params`.
- **callback** (*callable*) – Must accept at least 3 parameters: option name (`str`), all option names (short and long ones) (`tuple(str, str)`), and option params (`dict`).

static _for_each_option_group(*callback*, *options*)
Given set of options, call a callback for each option group.

Parameters

- **options** – List of option groups, or a dict listing options directly.

- **callback** (*callable*) – Must accept at least 2 parameters: `options` (`dict`), listing options in the group, and keyword parameter `group_name` (`str`), which is set to group name when the `options` defines an option group.

_parse_args(*args*, ***kwargs*)
Parse command-line arguments. Uses `argparse` for the actual parsing. Updates module's configuration store with values returned by parser.

Parameters **args** (*list*) – arguments passed to this module. Similar to what `sys.argv` provides on program level.

_parse_config(*paths*)
Parse configuration files. Uses `ConfigParser` for the actual parsing. Updates module's configuration store with values found returned by the parser.

Parameters **paths** (*list*) – List of paths to possible configuration files.

check_dryrun()
Checks whether this object supports current dry-run level.

check_required_options()

dryrun_allows(*msg*)
Checks whether current dry-run level allows an action which is disallowed on *DryRunLevels.DRY* level.

See *Configurable._dryrun_allows()* for detailed description.

dryrun_enabled
`True` if dry-run level is enabled, on any level.

dryrun_level
Return current dry-run level. This must be implemented by class descendants because each one finds the necessary information in different places.

eval_context
Return "evaluation context" - a dictionary of variable names (usually in uppercase) and their values, which is supposed to be used in various "evaluate *this*" operations like rendering of templates.

To provide nice and readable documentation of variables, returned by a module's `eval_context` property, assign a dictionary, describing these variables, to a local variable named `__content__`:

```
...

@property
def eval_context(self):
    __content__ = {
        'FOO': 'This is an important variable, extracted from clouds.'
     }

     return {
         'FOO': 42
     }
```

`gluetool` core will extract this information and will use it to generate different help texts like your module's help or a list of all known context variables.

Return type dict

isolatedrun_allows(*msg*)
Checks whether current dry-run level allows an action which is disallowed on *DryRunLevels.ISOLATED* level.

name = None
Module name. Usually matches the name of the source file, no suffix.

option(**names*)
Return values of given options from module's configuration store.

Parameters **names** (*str*) – names of requested options.

Returns either a value or `None` if such option does not exist. When multiple options are requested, a tuple of their values is returned, for a single option its value is **not** wrapped by a tuple.

options = {}
The `options` variable defines options accepted by module, and their properties:

```
options = {
    <option name>: {
        <option properties>
    },
    ...
}
```

where

- `<option name>` is used to *name* the option in the parser, and two formats are accepted (don't add any leading dashes (`-` nor `--`):
  - `<long name>`
  - `tuple(<short name>, <long name #1>, <long name #2>, ...)`
- the first of long names (`long name #1`) is used to identify the option - other long names are understood by argument parser but their values are stored under `long name #1` option.
- dictionary `<option properties>` is passed to `argparse.ArgumentParser.add_argument()` as keyword arguments when the option is being added to the parser, therefore any arguments recognized by `argparse` can be used.

It is also possible to use groups:

```
options = [
    (<group name>,  <group options>),
    ...
]
```

where

- `<group name>` is the name of the group, e.g. `Debugging options`
- `<group options>` is the `dict` with all group options, as described above.

This way, you can split pile of options into conceptualy closer groups of options. A single `dict` you would have is split into multiple smaller dictionaries, and each one is coupled with the group name in a `tuple`.

options_note = None
If set, it will be printed after all options as a help's epilog.

parse_args(*args*)
Public entry point to argument parsing. Child classes must implement this method, e.g. by calling *gluetool.glue.Configurable._parse_args()* which makes use of additional `argparse.ArgumentParser` options.

parse_config()
Public entry point to configuration parsing. Child classes must implement this method, e.g. by calling *gluetool.glue.Configurable._parse_config()* which requires list of paths.

required_options = []
Iterable of names of required options.

supported_dryrun_level = 0
Highest supported level of dry-run.

unique_name = None
Unque name of this (module) instance.

Used by modules, has no meaning elsewhere, but since dry-run checks are done on this level, it must be declared here to make pylint happy :/

class gluetool.glue.**DiscoveredModule**(*klass*, *group*)
Bases: tuple

Describes one discovered gluetool module.

Variables

- ***klass*** (Module) – a module class.
- ***group*** (*str*) – group the module belongs to.

_asdict()
Return a new OrderedDict which maps field names to their values

_field_types = {'group': <type 'str'>, 'klass': typing.Type[gluetool.glue.Module]}

_fields = ('klass', 'group')

classmethod _make(*iterable*, *new=<built-in method __new__ of type object>*, *len=<built-in function len>*)
Make a new DiscoveredModule object from a sequence or iterable

_replace(***kwds*)
Return a new DiscoveredModule object replacing specified fields with new values

group
Alias for field number 1

klass
Alias for field number 0

class gluetool.glue.**DryRunLevels**
Bases: enum.IntEnum

Dry-run levels.

Variables

- ***DEFAULT*** (*int*) – Default level - everything is allowed.
- ***DRY*** (*int*) – Well-known "dry-run" - no changes to the outside world are allowed.
- ***ISOLATED*** (*int*) – No interaction with the outside world is allowed (networks connections, reading files, etc.)

DEFAULT = 0

DRY = 1

ISOLATED = 2

class gluetool.glue.**Failure**(*module*, *exc_info*)
Bases: object

Bundles exception related info. Used to inform modules in their `destroy()` phase that `gluetool` session was killed because of exception raised by one of modules.

Parameters

- **module** (gluetool.glue.Module) – module in which the error happened, or `None`.
- **exc_info** (*tuple*) – Exception information as returned by `sys.exc_info()`.

Variables

- **module** (gluetool.glue.Module) – module in which the error happened, or `None`.
- ***exception*** (*Exception*) – Shortcut to `exc_info[1]`, if available, or `None`.
- **exc_info** (*tuple*) – Exception information as returned by `sys.exc_info()`.
- **sentry_event_id** (*str*) – If set, the failure was reported to the Sentry under this ID.

class gluetool.glue.**Glue**(*tool=None*, *sentry=None*)
Bases: *gluetool.glue.Configurable*

Main workhorse of the `gluetool`. Manages modules, their instances and runs them as requested.

Parameters

- **tool** (*gluetool.tool.Tool*) – If set, it's an `gluetool`-like tool that created this instance. Some functionality may need it to gain access to bits like its command-name.
- **sentry** (*gluetool.sentry.Sentry*) – If set, it provides interface to Sentry.

_check_pm_file(*filepath*)
Make sure a file looks like a `gluetool` module:

- can be processed by Python parser,
- imports *gluetool.glue.Glue* and *gluetool.glue.Module*,
- contains child class of *gluetool.glue.Module*.

Parameters **filepath** (*str*) – path to a file.

Returns `True` if file contains `gluetool` module, `False` otherwise.

Raises ***gluetool.glue.GlueError*** – when it's not possible to finish the check.

_discover_gm_in_dir(*dirpath*, *registry*, *pm_prefix*)
Discover `gluetool` modules in a directory tree.

In essence, it scans directory and its subdirectories for files with `.py` suffix, and searches for classes derived from *gluetool.glue.Module* in these files.

Parameters

- **dirpath** (*str*) – path to a directory.
- **DiscoveredModule) registry** (*dict(str,*) – registry of modules to which new ones would be added.
- **pm_prefix** (*str*) – a string used to prefix all imported Python module names.

_discover_gm_in_entry_point(*entry_point*, *registry*)

_discover_gm_in_file(*registry*, *filepath*, *pm_name*, *group_name*)
Discover `gluetool` modules in a file.

Attempts to import the file as a Python module, and then checks its content and looks for `gluetool` modules.

Parameters

- **DiscoveredModule) registry** (*dict(str,)*) – registry of modules to which new ones would be added.
- **filepath** (*str*) – path to a file.
- **pm_name** (*str*) – a Python module name to use for the imported module.
- **group_name** (*str*) – a `gluetool` module group name assigned to `gluetool` modules discovered in the file.

_do_import_pm(*filepath*, *pm_name*)
Attempt to import a file as a Python module.

Parameters

- **filepath** (*str*) – a file to import.
- **pm_name** (*str*) – name assigned to the imported module.

Returns imported Python module.

Raises ***gluetool.glue.GlueError*** – when import failed.

_eval_context()
Gather contexts of all modules in a pipeline and merge them together.

Always returns a unique dictionary object, therefore it is safe for caller to update it. The return value is not cached in any way, therefore the change if its content won't affect future callers.

Provided as a shared function, registered by the Glue instance itself.

Return type dict

_eval_context_module_caller()
Infere module instance calling the eval context shared function.

Return type *gluetool.glue.Module*

_import_pm(*filepath*, *pm_name*)
If a file contains `gluetool` modules, import the file as a Python module. If the file does not look like it contains `gluetool` modules, or when it's not possible to import the Python module successfully, method simply warns user and ignores the file.

Parameters

- **filepath** (*str*) – file to load.
- **pm_name** (*str*) – Python module name for the loaded module.

Returns loaded Python module.

Raises ***gluetool.glue.GlueError*** – when import failed.

_register_module(*registry*, *group_name*, *klass*, *filepath*)
Register one discovered `gluetool` module.

Parameters

- **DiscoveredModule) registry** (*dict(str,*) – module registry to add module to.
- **group_name** (*str*) – group the module belongs to.
- **klass** (Module) – module class.
- **filepath** (*str*) – path to a file module comes from.

add_shared(*funcname*, *module*)
Add a shared function. Overwrites previously registered shared function of the same name.

Parameters

- **funcname** (*str*) – name of the shared function.
- **module** (Module) – module providing the shared function.

current_module

current_pipeline

discover_modules(*entry_points=None*, *paths=None*)
Discover and load all accessible modules.

Two sources are examined:

1. entry points, handled by setuptools, to which Python packages can attach `gluetool` modules they provide,
2. directory trees.

Parameters

- **entry_points** (*list(str)*) – list of entry point names to which `gluetool` modules are attached. If not set, entry points set byt the configuration (`--module-entry-point` option) are used.
- **paths** (*list(str)*) – list of directories to search for `gluetool` modules. If not set, paths set by the configuration (`--module-path` option) are used.

Return type dict(str, *DiscoveredModule*)

Returns mapping between module names and `DiscoveredModule` instances, describing each module.

dryrun_level

eval_context
Returns "global" evaluation context - some variables that are nice to have in all contexts.

get_shared(*funcname*)
Return a shared function.

Parameters **funcname** (*str*) – name of the shared function.

Returns a callable (shared function), or `None` if no such shared function exists.

has_shared(*funcname*)
Check whether a shared function of a given name exists.

Parameters **funcname** (*str*) – name of the shared function.

Return type bool

init_module(*module_name*, *actual_module_name=None*)
Given a name of the module, create its instance and give it a name.

Parameters

- **module_name** (*str*) – Name under which will be the module instance known.
- **actual_module_name** (*str*) – Name of the module to instantiate. It does not have to match `module_name` - `actual_module_name` refers to the list of known `gluetool` modules while `module_name` is basically an arbitrary name new instance calls itself. If it's not set, which is the most common situation, it defaults to `module_name`.

Returns A *Module* instance.

module_config_paths
List of paths in which module config files reside.

module_data_paths
List of paths in which module data files reside.

module_entry_points
List of setuptools entry points to which modules are attached.

module_paths
List of paths in which modules reside.

modules_as_groups(*modules=None*)
Gathers modules by their groups.

Return type dict(str, dict(str, *DiscoveredModule*))

Returns dictonary where keys represent module groups, and values are mappings between module names and the corresponding modules.

modules_descriptions(*modules=None*, *groups=None*)
Returns a string with modules and their descriptions.

Parameters

- **DiscoveredModule) modules** (*dict(str,*) – mapping with modules. If not set, current known modules are used.
- **groups** (*list(str)*) – if set, limit descriptions to modules belinging to the given groups.

Return type str

name = 'gluetool core'

options = [('Global options', {('V', 'version'): {'action': 'store_true', 'help': '

parse_args(*args*)
Public entry point to argument parsing. Child classes must implement this method, e.g. by calling *gluetool.glue.Configurable._parse_args()* which makes use of additional `argparse.ArgumentParser` options.

parse_config(*paths*)
Public entry point to configuration parsing. Child classes must implement this method, e.g. by calling *gluetool.glue.Configurable._parse_config()* which requires list of paths.

require_shared(**names*, ***kwargs*)
Make sure given shared functions exist.

Parameters

- **names** (*tuple(str)*) – iterable of shared function names.
- **warn_only** (*bool*) – if set, only warning is emitted. Otherwise, when any required shared function didn't exist, an exception is raised.

run_module(*module_name*, *module_argv=None*, *actual_module_name=None*)
Syntax sugar for *run_modules()*, in the case you want to run just a one-shot module.

Parameters

- **module_name** (*str*) – Name under which will be the module instance known.
- **module_argv** (*list(str)*) – Arguments of the module.
- **actual_module_name** (*str*) – Name of the module to instantiate. It does not have to match `module_name` - `actual_module_name` refers to the list of known `gluetool` modules while `module_name` is basically an arbitrary name new instance calls itself. If it's not set, which is the most common situation, it defaults to `module_name`.

run_modules(*steps*)

run_pipeline(*pipeline*)

sentry_submit_exception(**args*, ***kwargs*)
Submits exceptions to the Sentry server. Does nothing by default, unless this instance is initialized with a `gluetool.sentry.Sentry` instance which actually does the job.

See `gluetool.sentry.Sentry.submit_exception()`.

sentry_submit_message(**args*, ***kwargs*)
Submits message to the Sentry server. Does nothing by default, unless this instance is initialized with a `gluetool.sentry.Sentry` instance which actually does the job.

See `gluetool.sentry.Sentry.submit_warning()`.

shared(*funcname*, **args*, ***kwargs*)
Call a shared function, passing it all positional and keyword arguments.

exception gluetool.glue.**GlueCommandError**(*cmd*, *output*, ***kwargs*)
Bases: *gluetool.glue.GlueError*

Exception raised when external command failes.

Parameters

- **cmd** (*list*) – Command as passed to gluetool.utils.run_command helper.
- **output** (gluetool.utils.ProcessOutput) – Process output data.

Variables

- **cmd** (*list*) – Command as passed to gluetool.utils.run_command helper.
- **output** (gluetool.utils.ProcessOutput) – Process output data.

exception gluetool.glue.**GlueError**(*message*, *caused_by=None*, *sentry_fingerprint=None*, *sentry_tags=None*, ***kwargs*)
Bases: exceptions.Exception

Generic `gluetool` exception.

Parameters

- **message** (*str*) – Exception message, describing what happened.

- **caused_by** (*tuple*) – If set, contains tuple as returned by `sys.exc_info()`, describing the exception that caused this one to be born. If not set, constructor will try to auto-detect this information, and if there's no such information, instance property `caused_by` will be set to `None`.
- **sentry_fingerprint** (*list(str)*) – if set, it is used as a Sentry fingerprint of the exception. See *sentry_fingerprint()* for more details.
- **str) sentry_tags** (*dict(str,*) – if set, it is merged with other tags when submitting the exception. See *sentry_tags()* for more details.

Variables

- **message** (*str*) – Exception message, describing what happened.
- **caused_by** (*tuple*) – If set, contains tuple as returned by `sys.exc_info()`, describing the exception that caused this one to be born. `None` otherwise.

no_sentry_exceptions = []

sentry_fingerprint(*current*)

Default grouping of events into issues might be too general for some cases. This method gives users a chance to provide custom fingerprint Sentry could use to group events in a more suitable way.

E.g. user might be interested in some sort of connection issues but they would like to have them grouped not by a traceback (which is the default method) but per remote host IP. For that, the Sentry integration code will call `sentry_fingerprint` method of a raised exception, and the method should return new fingerprint, let's say `[<exception class name>, <remote IP>]`, and Sentry will group events using this fingerprint.

If the exception was raised with `sentry_fingerprint` parameter set, it is returned instead of `current`, after prefixing the list of tags with a name of the exception's class.

Parameters **current** (*list(str)*) – current fingerprint. Usually `['{{ default }}']` telling Sentry to use its default method, but it could already be more specific.

Return type list(str)

Returns new fingerprint, e.g. `['FailedToConnectToAPI', '10.20.30.40']`

sentry_tags(*current*)

Add, modify or remove tags attached to a Sentry event, reported when the exception was raised.

Most common usage would be an addition of tags, e.g. `remote-host` to allow search for events related to the same remote address.

If the exception was raised with `sentry_tags` parameter set, its value is injected to `current` before returning it.

Parameters **str) current** (*dict(str,*) – current set of tags and their values.

Return type dict(str, str)

Returns new set of tags. It is possible to add tags directly into `current` and then return it.

submit_to_sentry

Decide whether the exception should be submitted to Sentry or not. By default, all exceptions are submitted. Exception listed in *no_sentry_exceptions* are not submitted.

Return type bool

Returns `True` when the exception should be submitted to Sentry, `False` otherwise.

exception gluetool.glue.**GlueRetryError**(*message*, *caused_by=None*, *sentry_fingerprint=None*, *sentry_tags=None*, ***kwargs*)

Bases: *gluetool.glue.GlueError*

Retry gluetool exception

class gluetool.glue.**Module**(*glue*, *name*)

Bases: *gluetool.glue.Configurable*

Base class of all `gluetool` modules.

Parameters **glue** (*gluetool.glue.Glue*) – `Glue` instance owning the module.

Variables

- ***glue*** (*gluetool.glue.Glue*) – `Glue` instance owning the module.
- **_config** (*dict*) – internal configuration store. Values of all module options are stored here, regardless of them being set on command-line or in the configuration file.
- **_overloaded_shared_functions** (*dict*) – If a shared function added by this module overloades an older function of the same name, registered by a previous module, the overloaded one is added into this dictionary. The module can then call this saved function - using *overloaded_shared()* - to implement a "chain" of shared functions, when one calls another, implementing the same operation.

_generate_shared_functions_help()

Generate help for shared functions provided by the module.

Returns Formatted help, describing module's shared functions.

_paths_with_module(*roots*)

Return paths cretaed by joining roots with module's unique name.

Parameters **roots** (*list(str)*) – List of root directories.

add_shared()

Add all shared functions declared by the module.

description = None

Short module description, displayed in `gluetool`'s module listing.

destroy(*failure=None*)

Here should go any code that needs to be run on exit, like job cleanup etc.

Parameters **failure** (*gluetool.glue.Failure*) – if set, carries information about failure that made `gluetool` to destroy the whole session. Modules might want to take actions based on provided information, e.g. send different notifications.

dryrun_level

execute()

In this method, modules can perform any work they deemed necessary for completing their purpose. E.g. if the module promises to run some tests, this is the place where the code belongs to.

By default, this method does nothing. Reimplement as needed.

get_shared(*funcname*)

Return a shared function.

Parameters **funcname** (*str*) – name of the shared function.

Returns a callable (shared function), or `None` if no such shared function exists.

has_shared(*funcname*)
Check whether a shared function of a given name exists.

Parameters **funcname** (*str*) – name of the shared function.

Return type bool

overloaded_shared(*funcname*, **args*, ***kwargs*)
Call a shared function overloaded by the one provided by this module. This way, a module can give chance to other implementations of the same name, e.g. function named `publish_message`, working with message bus A, would call previously shared function holding of this name, registered by a module earlier in the pipeline, which works with message bus B.

parse_args(*args*)
Public entry point to argument parsing. Child classes must implement this method, e.g. by calling *gluetool.glue.Configurable._parse_args()* which makes use of additional `argparse.ArgumentParser` options.

parse_config()
Public entry point to configuration parsing. Child classes must implement this method, e.g. by calling *gluetool.glue.Configurable._parse_config()* which requires list of paths.

require_shared(**names*, ***kwargs*)
Make sure given shared functions exist.

Parameters

- **names** (*tuple(str)*) – iterable of shared function names.
- **warn_only** (*bool*) – if set, only warning is emitted. Otherwise, when any required shared function didn't exist, an exception is raised.

run_module(*module*, *args=None*)

sanity()
In this method, modules can define additional checks before execution.

Some examples:

- Advanced checks on passed options
- Check for additional requirements (tools, data, etc.)

By default, this method does nothing. Reimplement as needed.

shared(*funcname*, **args*, ***kwargs*)
Call a shared function, passing it all positional and keyword arguments.

shared_functions = []
Iterable of names of shared functions exported by the module.

gluetool.glue.**ModuleRegistryType**
alias of `typing.Dict`

class gluetool.glue.**NamedPipeline**(*glue*, *name*, *steps*)
Bases: *gluetool.glue.Pipeline*

Pipeline with a name. The name is recorded in log messages emitted by the pipeline itself.

Parameters

- **glue** (`Glue`) – *Glue* instance, taking care of this pipeline.
- **name** (*str*) – name of the pipeline.
- **steps** (*list(PipelineStep)*) – modules to run and their options.

Variables

- **modules** (*list (Module)*) – list of instantiated modules forming the pipeline.
- **current_module** (Module) – if set, it is the module which is currently being executed.

class gluetool.glue.**Pipeline**(*glue*, *steps*, *logger=None*)

Bases: *gluetool.log.LoggerMixin*, object

Pipeline of gluetool modules. Defined by its steps, takes care of registering their shared functions, running modules and destroying the pipeline.

To simplify the workflow, 2 primitives are defined:

- *_safe_call()* - calls a given callback, returns its return value. Any exception raised by the callback is wrapped by *Failure* instance and returned instead of what callback would return.
- *_for_each_module()* - loop over given list of modules, calling given callback for each of the modules. *_safe_call()* is used for the call, making sure we always have a return value and no exceptions.

Coupled with the following rules, things clear up a bit:

- Callbacks are allowed to return either None or Failure instance. * This rule matches _safe_call behavior. There's no need to worry about the return values, return value

  of a callback can be immediately passed through _safe_call.

  - There are no other possible return values - return value is **always** either *nothing* or *failure*.

- There are no "naked" exceptions, all are catched by _safe_call and converted to a common return value.
- Users of _safe_call and _for_each_module also return either None or Failure instance, therefore it is very easy to end method by a `return self._for_each_method(...)` call - return types of callback, _safe_call, _for_each_module and our method match, no need to translate them between these method.

Parameters

- **glue** (Glue) – *Glue* instance, taking care of this pipeline.
- **steps** (*list (PipelineStep)*) – modules to run and their options.

Variables

- **modules** (*list (Module)*) – list of instantiated modules forming the pipeline.
- **current_module** (Module) – if set, it is the module which is currently being executed.

_add_shared(*funcname*, *module*, *func*)

Add a shared function. Overwrites previously registered shared function of the same name.

Private part of API, for easier testing.

Parameters

- **funcname** (*str*) – name of the shared function.
- **module** (Module) – module providing the shared function.
- **func** (*callable*) – the shared function.

_destroy(*failure=None*)

"Destroy" the pipeline - call each module's destroy method, reversing the order of modules. If a destroy method raises an exception, loop ends and the failure is returned.

Parameters **failure** (Failure) – if set, it represents a failure that caused pipeline to stop, which was followed by a call to currently running _destroy. It is passed to modules' destroy methods.

Returns None if everything went well, or a *Failure* instance if any destroy method raised an exception.

_execute()

_for_each_module(*modules*, *callback*, **args*, ***kwargs*)

For each module in a list, call a given function with module as its first argument. If the call returns anything but None, the value is returned by this function as well, ending the loop.

Parameters

- **modules** (*list (Module)*) – list of modules to iterate over.
- **callback** (*callable*) – a callback, accepting at least one parameter, current module of the loop. Must return either None or *Failure* instance, although it can freely raise exceptions.

Returns value returned by callback, or None when loop finished.

_log_failure(*module*, *failure*, *label=None*)

Log a failure, and submit it to Sentry.

Parameters

- **module** (Module) – module to use for logging - apparently, the failure appeared when this module was running.
- **failure** (Failure) – failure to log.
- **label** (*str*) – label for logging purposes. If it's set and exception exists, exception message is appended. If it's not set and exception exists, exception message is used. If failure has no exception, a generic message is the final choice.

_safe_call(*callback*, **args*, ***kwargs*)

"Safe" call a function with given arguments, converting raised exceptions to *Failure* instance.

Parameters **callback** (*callable*) – callable to call. Must return either None or *Failure* instance, although it can freely raise exceptions.

Returns value returned by callback, or *Failure* instance wrapping exception raise by callback.

_sanity()

_setup()

add_shared(*funcname*, *module*)

Add a shared function. Overwrites previously registered shared function of the same name.

Parameters

- **funcname** (*str*) – name of the shared function.
- **module** (Module) – module providing the shared function.

get_shared(*funcname*)

Return a shared function.

Parameters **funcname** (*str*) – name of the shared function.

Returns a callable (shared function), or None if no such shared function exists.

has_shared(*funcname*)
Check whether a shared function of a given name exists.

Parameters **funcname** (*str*) – name of the shared function.

Return type bool

run()
Run a pipeline - instantiate modules, prepare and execute each of them. When done, destroy all modules.

Returns tuple of two items, each of them either `None` or a *Failure* instance. The first item represents the output of the pipeline, the second item represents the output of the destroy chain. If the item is `None`, the stage finished without any issues, if it's a `Failure` instance, then an exception was raised during the stage, and `Failure` wraps it.

shared_functions = None
Shared function registry. funcname: (module, fn)

class gluetool.glue.**PipelineAdapter**(*logger*, *pipeline_name*)
Bases: *gluetool.log.ContextAdapter*

Custom logger adapter, adding pipeline name as a context.

Parameters **logger** (*logging.Logger*) – parent logger this adapter modifies.

class gluetool.glue.**PipelineStep**
Bases: `object`

Step of `gluetool`'s pipeline - which is basically just a list of steps.

to_module(*glue*)

class gluetool.glue.**PipelineStepCallback**(*name*, *callback*, **args*, ***kwargs*)
Bases: *gluetool.glue.PipelineStep*

Step of `gluetool`'s pipeline backed by callable.

Parameters

- **name** (*str*) – name to give to the module instance. This name is used e.g. in logging or when searching for module's config file.
- **callback** (*callable*) – a callable to execute.

serialize_to_json()

to_module(*glue*)

classmethod unserialize_from_json(*serialized*)

class gluetool.glue.**PipelineStepModule**(*module*, *actual_module=None*, *argv=None*)
Bases: *gluetool.glue.PipelineStep*

Step of `gluetool`'s pipeline backed by a module.

Parameters

- **module** (*str*) – name to give to the module instance. This name is used e.g. in logging or when searching for module's config file.
- **actual_module** (*str*) – The actual module class the step uses. Usually it is same as `module` but may differ, `module` is then a mere "alias". `actual_module` is used to locate a module class, whose instance is then given name `module`.
- **argv** (*list(str)*) – list of options to be given to the module, in a form similar to `sys.argv`.

module_designation

serialize_to_json()

to_module(*glue*)

classmethod unserialize_from_json(*serialized*)

exception gluetool.glue.**SoftGlueError**(*message*, *caused_by=None*, *sentry_fingerprint=None*, *sentry_tags=None*, ***kwargs*)

Bases: *gluetool.glue.GlueError*

Soft errors are errors Glue Ops and/or developers shouldn't be bothered with, things that are up to the user to fix, e.g. empty set of tests. **Hard** errors are supposed to warn Ops/Devel teams about important infrastructure issues, code deficiencies, bugs and other issues that are fixable only by actions of Glue staff.

However, we still must provide notification to user(s), and since we expect them to fix the issues that led to raising the soft error, we must provide them with as much information as possible. Therefore modules dealing with notifications are expected to give these exceptions a chance to influence the outgoing messages, e.g. by letting them provide an e-mail body template.

gluetool.glue.**retry**(**args*)

Retry decorator This decorator catches given exceptions and returns libRetryError exception instead.

usage: @retry(exception1, exception2, ..)

3.2 gluetool.help module

Command-line `--help` helpers (muhaha!).

`gluetool` uses docstrings to generate help for command-line options, modules, shared functions and other stuff. To generate good looking and useful help texts a bit of work is required. Add the Sphinx which we use to generate nice documentation of `gluetool`'s API and structures, with its directives, and it's even more work to make readable output. Therefore these helpers, separated in their own file to keep things clean.

gluetool.help.**C_ARGNAME**(*text*)

gluetool.help.**C_FUNCNAME**(*text*)

gluetool.help.**C_LITERAL**(*text*)

class gluetool.help.**DummyTextBuilder**

Sphinx `TextWriter` (and other writers as well) requires an instance of `Builder` class that brings configuration into the rendering process. The original `TextBuilder` requires Sphinx *application* which brings a lot of other dependencies (e.g. source paths and similar stuff) which are impractical in our use case ("render short string to plain text"). Therefore this dummy class which just provides minimal configuration - `TextWriter` requires nothing else from `Builder` instance.

See `sphinx/writers/text.py` for the original implementation.

class DummyConfig

text_newlines = '\n'

text_sectionchars = '*=-~"+`'

config

alias of *DummyConfig*

translator_class = None

class gluetool.help.**LineWrapRawTextHelpFormatter**(**args*, ***kwargs*)
Bases: argparse.RawDescriptionHelpFormatter

_split_lines(*text*, *width*)

class gluetool.help.**TextTranslator**(*document*, *builder*)
Bases: sphinx.writers.text.TextTranslator

depart_field_name(*node*)

depart_literal(*node*)

visit_field_name(*node*)

visit_literal(*node*)

gluetool.help.**doc_role_handler**(*role*, *rawtext*, *text*, *lineno*, *inliner*, *options=None*, *context=None*)
Format `:doc:` roles, used to reference another bits of documentation.

gluetool.help.**docstring_to_help**(*docstring*, *width=None*, *line_prefix=' '*)
Given docstring, process and render it as a plain text. This conversion function is used to generate nice and readable help strings used when printing help on command line.

Parameters

- **docstring** (*str*) – raw docstring of Python object (function, method, module, etc.).
- **width** (*int*) – Maximal line width allowed.
- **line_prefix** (*str*) – prefix each line with this string (e.g. to indent it with few spaces or tabs).

Returns formatted docstring.

gluetool.help.**eval_context_help**(*source*)
Generate and format help for an evaluation context of a module. Looks for context content, and gives it a nice header, suitable for command-line help, applying formatting along the way.

Parameters **source** (gluetool.glue.Configurable) – object whose eval context help we should format.

Returns Formatted help.

gluetool.help.**extract_eval_context_info**(*source*, *logger=None*)
Extract information of evaluation context content from the `source` - a module or any other object with `eval_context` property. The information we're looking for is represented by an assignment to special variable, `__content__`, in the body of `eval_context` getter. `__content__` is expected to be assigned a dictionary, listing context variables (keys) and their descriptions (values).

If it's not possible to find such information, or an exception is raised, the function returns an empty dictionary.

Parameters **source** (gluetool.glue.Configurable) – object to extract information from.

Return type dict(str, str)

gluetool.help.**function_help**(*func*, *name=None*)
Uses function's signature and docstring to generate a plain text help describing the function.

Parameters

- **func** (*callable*) – Function to generate help for.
- **name** (*str*) – If not set, `func.__name__` is used by default.

Returns `(signature, body)` pair.

gluetool.help.**functions_help**(*functions*)
Generate help for a set of functions.

Parameters **callable) functions** (*list (str,)*) – Functions to generate help for, passed as name and the corresponding callable pairs.

Return type str

Returns Formatted help.

gluetool.help.**option_help**(*txt*)
Given option help text, format it to be more suitable for command-line help. Options can provide a single line of text, or mutiple lines (using triple quotes and docstring-like indentation).

Parameters **txt** (*str*) – Raw option help text.

Returns Formatted option help text.

gluetool.help.**py_default_role**(*role*, *rawtext*, *text*, *lineno*, *inliner*, *options=None*, *content=None*)
Default handler we use for `py:...` roles, translates text to literal node.

gluetool.help.**rst_to_text**(*text*)
Render given text, written with RST, as plain text.

Parameters **text** (*str*) – string to render.

Return type str

Returns plain text representation of `text`.

gluetool.help.**trim_docstring**(*docstring*)
Quoting *PEP 257 <https://www.python.org/dev/peps/pep-0257/#handling-docstring-indentation>*:

Docstring processing tools will strip a uniform amount of indentation from the second and further lines of the docstring, equal to the minimum indentation of all non-blank lines after the first line. Any indentation in the first line of the docstring (i.e., up to the first newline) is insignificant and removed. Relative indentation of later lines in the docstring is retained. Blank lines should be removed from the beginning and end of the docstring.

Code bellow follows the quote.

This method does exactly that, therefore we can keep properly aligned docstrings while still use them for reasonably formatted help texts.

Parameters **docstring** (*str*) – raw docstring.

Return type str

Returns docstring with lines stripped of leading whitespace.

3.3 gluetool.log module

Logging support.

Sets up logging environment for use by `gluetool` and modules. Based on standard library's `logging` module, augmented a bit to support features loke colorized messages and stackable context information.

Example usage:

```
# initialize logger as soon as possible
Logging.setup_logger()
logger = Logging.get_logger()
```

(continues on next page)

(continued from previous page)

```
# now it's possible to use it for logging:
logger.debug('foo!')

# find out what your logging should look like, e.g. by parsing command-line options
...

# tell logger about the final setup
logger = Logging.setup_logger(output_file='/tmp/foo.log', level=...)

# when you want to make logging methods easily accessible in your class, throw in
# LoggerMixin:
class Foo(LoggerMixin, ...):
  def __init__(self, some_logger):
      super(Foo, self).__init__(logger)

      self.debug('foo!')
```

class gluetool.log.**BlobLogger**(*intro*, *outro=None*, *on_finally=None*, *writer=None*)

Bases: object

Context manager to help with "real time" logging - some code may produce output continuously, e.g. when running a command and streaming its output to our stdout, and yet we still want to wrap it with boundaries and add a header.

This code:

```
with BlobLogger('ls of root', outro='end of ls'):
    subprocess.call(['ls', '/'])
```

will lead to the output similar to this:

```
[20:30:50] [+] ---v---v---v---v---v--- ls of root
bin  boot  data  dev ...
[20:30:50] [+] ---^---^---^---^---^--- end of ls
```

Note: When you already hold the data you wish to log, please use *gluetool.log.log_blob()* or *gluetool.log.log_dict()*. The example above could be rewritten using `log_blob` by using `subprocess.check_output()` and passing its return value to `log_blob`. `BlobLogger` is designed to wrap output whose creation caller don't want to (or cannot) control.

Parameters

- **intro** (*str*) – Label to show what is the meaning of the logged data.
- **outro** (*str*) – Label to show by the final boundary to mark the end of logging.
- **on_finally** (*callable*) – When set, it will be called in `__exit__` method. User of this context manager might need to flush used streams or close resources even in case the exception was raised while inside the context manager. `on_finally` is called with all arguments the `__exit__` was called, and its return value is returned by `__exit__` itself, therefore it can examine possible exceptions, and override them.
- **writer** (*callable*) – A function which is used to actually log the text. Usually a one of some logger methods.

class gluetool.log.**ContextAdapter**(*logger*, *extra=None*, *contexts=None*)

Bases: logging.LoggerAdapter

Generic logger adapter that collects "contexts", and prepends them to the message.

A "context" is a descriptive string, with a priority. Contexts are then sorted by their priorities before inserting them into the message (lower priority means context will be placed closer to the beggining of the line - highest priority comes last.

`ContextAdapter` is a corner stone of our logging infrastructure, **everything** is supposed to, one way or another, use this class (or one of its children) for logging. We should avoid using bare `Logger` instances because they lack context propagation, message routing to debug and verbose files, `verbose` method & `sentry` parameter.

Parent class, `logging.LoggerAdapter`, provides all common logging methods. We overload them with our own implementations because we want to 1) handle type annotations on our side, 2) let users submit messages to Sentry.

Parameters

- **logger** – parent logger this adapter modifies.
- **extras** (*dict*) – additional extra keys passed to the parent class. The dictionary is then used to update messages' `extra` key with the information about context. Keys starting with *ctx_* are removed from this dictionary, and become contexts.
- **tuple(int obj)) contexts** (*dict(str,)*) – mapping of context names and tuples of their priority and value.

addHandler(**args*, ***kwargs*)

debug(*msg*, *exc_info=None*, *extra=None*, *sentry=False*)

Delegate a debug call to the underlying logger, after adding contextual information from this adapter instance.

error(*msg*, *exc_info=None*, *extra=None*, *sentry=False*)

Delegate an error call to the underlying logger, after adding contextual information from this adapter instance.

exception(*msg*, *exc_info=None*, *extra=None*, *sentry=False*)

Delegate an exception call to the underlying logger, after adding contextual information from this adapter instance.

info(*msg*, *exc_info=None*, *extra=None*, *sentry=False*)

Delegate an info call to the underlying logger, after adding contextual information from this adapter instance.

isEnabledFor(*level*)

See if the underlying logger is enabled for the specified level.

log(*level*, *msg*, *exc_info=None*, *extra=None*, *sentry=False*)

Delegate a log call to the underlying logger, after adding contextual information from this adapter instance.

process(*msg*, *kwargs*)

Process the logging message and keyword arguments passed in to a logging call to insert contextual information. You can either manipulate the message itself, the keyword args or both. Return the message and kwargs modified (or not) to suit your needs.

Normally, you'll only need to override this one method in a LoggerAdapter subclass for your specific needs.

removeHandler(**args*, ***kwargs*)

verbose(*msg*, *exc_info=None*, *extra=None*, *sentry=False*)

warn(*msg*, *exc_info=None*, *extra=None*, *sentry=False*)
Delegate a warning call to the underlying logger, after adding contextual information from this adapter instance.

warning(*msg*, *exc_info=None*, *extra=None*, *sentry=False*)
Delegate a warning call to the underlying logger, after adding contextual information from this adapter instance.

class gluetool.log.**JSONLoggingFormatter**(*colors=False*, *log_tracebacks=False*, *prettify=False*)
Bases: logging.Formatter

Custom logging formatter producing a JSON dictionary describing the log record.

static _format_exception_chain(*serialized*, *exc_info*)
"Format" exception chain - transform it into a bunch of JSON structures describing exceptions, stack frames, local variables and so on.

Serves the same purpose as LoggingFormatter._format_exception_chain but that one produces a string, textual representation suitable for printing. This method produces JSON structures, suitable for, hm, JSON log.

format(*record*)
Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

class gluetool.log.**LoggerMixin**(*logger*, **args*, ***kwargs*)
Bases: object

Use as a parent class (or one of them) when you want to "attach" methods of a given logger to class' instances.

Parameters **logger** (ContextAdapter) – logger to propagate.

attach_logger(*logger*)
Initialize this object's logging methods to those provided by a given logger.

class gluetool.log.**Logging**
Bases: object

Container wrapping configuration and access to logging infrastructure gluetool uses for logging.

OUR_LOGGERS = (<logging.Logger object>, <logging.Logger object>, <logging.Logger objec

static _setup_log_file(*filepath*, *level*, *limit_level=False*, *formatter_class=<class 'gluetool.log.LoggingFormatter'>*, *prettify=False*)

adapted_logger = None
Logger singleton - if anyone asks for a logger, they will get this one. Needs to be properly initialized by calling *setup_logger()*.

static configure_logger(*logger*)
Configure given logger to conform with Gluetool's idea of logging. The logger is set to VERBOSE level, shared stderr handler is added, and Sentry integration status is propagated as well.

After this method, the logger will behave like Gluetool's main logger.

debug_file_handler = None

static enable_debug_file(*logger*)

static enable_json_file(*logger*)

static enable_logger_sentry(*logger*)

static enable_verbose_file(*logger*)

static get_logger()
Returns a logger-like object suitable for logging stuff.

Return type *ContextAdapter*

Returns an instance of ContextAdapter wrapping root logger, or `None` when there's no logger yet.

json_file_handler = None

logger = None
Bare root logger we're hiding inside `adapted_logger`

sentry = None

static setup_logger(*level=20*, *debug_file=None*, *verbose_file=None*, *json_file=None*, *json_file_pretty=False*, *json_output=False*, *json_output_pretty=False*, *sentry=None*, *show_traceback=False*)
Create and setup logger.

This method is called at least twice:

- when *gluetool.glue.Glue* is instantiated: only a `stderr` handler is set up, with loglevel being `INFO`;
- when all arguments and options are processed, and Glue instance can determine desired log level, whether it's expected to stream debugging messages into a file, etc. This time, method only modifies propagates necessary updates to already existing logger.

Parameters

- **debug_file** (*str*) – if set, new handler will be attached to the logger, streaming messages of at least `DEBUG` level into this this file.
- **verbose_file** (*str*) – if set, new handler will be attached to the logger, streaming messages of `VERBOSE` log levels into this this file.
- **json_file** (*str*) – if set, all logging messages are sent to this file in a form of JSON structures.
- **json_output** (*bool*) – if set, all logging messages sent to the terminal are emitted as JSON structures.
- **level** (*int*) – desired log level. One of constants defined in `logging` module, e.g. `logging.DEBUG` or `logging.ERROR`.
- **sentry** (*bool*) – if set, logger will be augmented to send every log message to the Sentry server.
- **show_traceback** (*bool*) – if set, exception tracebacks would be sent to `stderr` handler as well as to the debug file.

Return type *ContextAdapter*

Returns a *ContextAdapter* instance, set up for logging.

stderr_handler = None
Stream handler printing out to stderr.

verbose_file_handler = None

class gluetool.log.**LoggingFormatter**(*colors=True*, *log_tracebacks=False*, *prettify=False*)
Bases: logging.Formatter

Custom log record formatter. Produces output in form of:

```
[stamp] [level] [ctx1] [ctx2] ... message
```

Parameters

- **colors** (*bool*) – if set, colorize output. Enabled by default but when used with file-backed destinations, colors are disabled by logging subsystem.
- **log_tracebacks** (*bool*) – if set, add tracebacks to the message. By default, we don't need tracebacks on the terminal, unless its loglevel is verbose enough, but we want them in the debugging file.

static _format_exception_chain(*exc_info*)
Format exception chain. Start with the one we're given, and follow its *caused_by* property until we ran out of exceptions to format.

_level_color = {20: <function <lambda>>, 30: <function <lambda>>, 40: <function <la

_level_tags = {5: 'V', 10: 'D', 20: '+', 30: 'W', 40: 'E', 50: 'C'}
Tags used to express loglevel.

format(*record*)
Format a logging record. It puts together pieces like time stamp, log level, possibly also different contexts if there are any stored in the record, and finally applies colors if asked to do so.

Parameters **record** (*logging.LogRecord*) – record describing the event.

Return type str

Returns string representation of the event record.

class gluetool.log.**ModuleAdapter**(*logger*, *module*)
Bases: *gluetool.log.ContextAdapter*

Custom logger adapter, adding module name as a context.

Parameters

- **logger** – parent logger this adapter modifies.
- **module** (gluetool.glue.Module) – module whose name is added as a context.

class gluetool.log.**PackageAdapter**(*logger*, *name*)
Bases: *gluetool.log.ContextAdapter*

Custom logger dapter, adding a package name as a context. Intended to taint log records produced by a 3rd party packages.

Parameters

- **logger** – parent logger this adapter modifies.
- **name** (*str*) – name of the library.

class gluetool.log.**SingleLogLevelFileHandler**(*level*, **args*, ***kwargs*)
Bases: logging.FileHandler

emit(*record*)
Emit a record.

If the stream was not opened because 'delay' was specified in the constructor, open it before calling the superclass's emit.

class gluetool.log.**StreamToLogger**(*log_fn*)
Bases: object

Fake file-like stream object that redirects writes to a given logging method.

write(*buf*)

gluetool.log.**_add_thread_context**(*contexts*, *record*)

gluetool.log.**_extract_stack**(*tb*)
Construct a "stack" by merging two sources of data:

1. what's provided by traceback.extract_tb(), i.e. (filename, lineno, fnname, text) tuple for each frame;
2. stack frame objects, hidden inside traceback object and available via following links from one frame to another.

Return type list(list(str, int, str, str, frame))

gluetool.log.**_json_dump**(*struct*, ***kwargs*)
Dump given data structure as a JSON string. Additional arguments are passed to json.dumps.

gluetool.log.**_move_contexts**(*src*, *dst*)

gluetool.log.**format_blob**(*blob*)
Format a blob of text for printing. Wraps the text with boundaries to mark its borders.

gluetool.log.**format_dict**(*dictionary*)
Format a Python data structure for printing. Uses json.dumps() formatting capabilities to present readable representation of a given structure.

gluetool.log.**format_table**(*table*, ***kwargs*)
Format a table, represented by an iterable of rows, represented by iterables.

Internally, tabulate is used to do the formatting. All keyword arguments are passed to tabulate call.

Parameters **table** (*list(list())*) – table to format.

Returns formatted table.

gluetool.log.**format_xml**(*element*)
Format an XML element, e.g. Beaker job description, for printing.

Parameters **element** – XML element to format.

gluetool.log.**log_blob**(*writer*, *intro*, *blob*)
Log "blob" of characters of unknown structure, e.g. output of a command or response of a HTTP request. The blob is preceded by a header and followed by a footer to mark exactly the blob boundaries.

Note: For logging structured data, e.g. JSON or Python structures, use *gluetool.log.log_dict()*. It will make structure of the data more visible, resulting in better readability of the log.

Parameters

- **writer** (*callable*) – A function which is used to actually log the text. Usually a one of some logger methods.
- **intro** (*str*) – Label to show what is the meaning of the logged blob.
- **blob** (*str*) – The actual blob of text.

gluetool.log.**log_dict**(*writer*, *intro*, *data*)

Log structured data, e.g. JSON responses or a Python `list`.

Note: For logging unstructured "blobs" of text, use *gluetool.log.log_blob()*. It does not attempt to format the output, and wraps it by header and footer to mark its boundaries.

Note: Using *gluetool.log.format_dict()* directly might be shorter, depending on your your code. For example, this code:

```
self.debug('Some data:\n{}'.format(format_dict(data)))
```

is equivalent to:

```
log_dict(self.debug, 'Some data', data)
```

If you need more formatting, or you wish to fit more information into a single message, using logger methods with `format_dict` is a way to go, while for logging a single structure `log_dict` is more suitable.

Parameters

- **writer** (*callable*) – A function which is used to actually log the text. Usually a one of some logger methods.
- **intro** (*str*) – Label to show what is the meaning of the logged structure.
- **blob** (*str*) – The actual data to log.

gluetool.log.**log_table**(*writer*, *intro*, *table*, ***kwargs*)

Log a formatted table.

All keyword arguments are passed to *format_table()* call which does the actual formatting.

Parameters

- **writer** (*callable*) – A function which is used to actually log the text. Usually a one of some logger methods.
- **intro** (*str*) – Label to show what is the meaning of the logged table.
- **table** (*list(list())*) – table to format.

gluetool.log.**log_xml**(*writer*, *intro*, *element*)

Log an XML element, e.g. Beaker job description.

Parameters

- **writer** (*callable*) – A function which is used to actually log the text. Usually a one of some logger methods.
- **intro** (*str*) – Label to show what is the meaning of the logged blob.
- **element** – XML element to log.

gluetool.log.**print_wrapper**(**args*, ***kwds*)
While active, replaces sys.stdout and sys.stderr streams with fake file-like streams which send all outgoing data to a given logging method.

Parameters

- **log_fn** (*call*) – A callback that is called for every line, produced by print. If not set, a info method of gluetool main logger is used.
- **label** (*text*) – short description, presented in the intro and outro headers, wrapping captured output.

3.4 gluetool.proxy module

Proxying object wrapper.

class gluetool.proxy.**Proxy**(*obj*)
Bases: future.types.newobject.newobject

Taking advantage of duck typing - wrap instance of class Foo with a Proxy, and the result will behave pretty much like Foo instance. Inherit from Proxy to create custom wrappers, extended with your own methods.

classmethod _create_class_proxy(*theclass*)
creates a proxy for the given class

_obj

_special_names = ['__abs__', '__add__', '__and__', '__call__', '__cmp__', '__coerce__'

3.5 gluetool.tool module

Heart of the "gluetool" script. Referred to by setuptools' entry point.

class gluetool.tool.**Gluetool**
Bases: object

_deduce_pipeline_desc(*argv*, *modules*)
Split command-line arguments, left by gluetool, into a pipeline description, splitting them by modules and their options.

Parameters

- **argv** (*list*) – Remainder of sys.argv after removing gluetool's own options.
- **modules** (*list (text)*) – List of known module names.

Returns Pipeline description in a form of a list of *gluetool.glue.PipelineStepModule* instances.

_exit_logger
Return logger for use when finishing the gluetool pipeline.

_handle_failure(*failure*, *do_quit=True*)

_handle_failure_core(*failure*, *do_quit=True*)

_quit(*exit_status*)
Log exit status and quit.

_version

check_options(**args*, ***kwargs*)

log_cmdline(*argv*, *pipeline_desc*)

main()

run_pipeline(**args*, ***kwargs*)

setup(**args*, ***kwargs*)

gluetool.tool.**handle_exc**(*func*)

gluetool.tool.**main**()

3.6 gluetool.utils module

Various helpers.

class gluetool.utils.**Bunch**(***kwargs*)
Bases: object

class gluetool.utils.**Command**(*executable*, *options=None*, *logger=None*)
Bases: *gluetool.log.LoggerMixin*, object

Wrap an external command, its options and other information, necessary for running the command.

The main purpose is to gather all relevant pieces into a single space, call subprocess.Popen, and log everything.

By default, both standard output and error output are of the process are captured and returned back to caller. Under some conditions, caller might want to see the output in "real-time". For that purpose, they can pass callable via inspect_callback parameter - such callable will be called for every received bit of input on both standard and error outputs. E.g.

```
def foo(stream, s, flush=False):
    if s is not None and 'a' in s:
        print s

Command(['/bin/foo']).run(inspect=foo)
```

This example will print all substrings containing letter *a*. Strings passed to foo may be of arbitrary lengths, and may change between subsequent use of Command class.

Parameters

- **executable** (*list*) – Executable to run. Feel free to use the whole command, including its options, if you have no intention to modify them before running the command.
- **options** (*list*) – If set, it's a list of options to pass to the executable. Options are specified in a separate list to allow modifications of executable and options before actually running the command.
- **logger** (gluetool.log.ContextAdapter) – Parent logger whose methods will be used for logging.

New in version 1.1.

_apply_quotes()
Return options to pass to Popen. Applies quotes as necessary.

_communicate_batch()

_communicate_inspect(*inspect_callback*)

_construct_output()

run(*inspect=False*, *inspect_callback=None*, ***kwargs*)
Run the command, wait for it to finish and return the output.

Parameters

- **inspect** (*bool*) – If set, `inspect_callback` will receive the output of command in "real-time".
- **inspect_callback** (*callable*) – callable that will receive command output. If not set, default "write to `sys.stdout`" is used.

Return type gluetool.utils.ProcessOutput instance

Returns *gluetool.utils.ProcessOutput* instance whose attributes contain data returned by the child process.

Raises

- ***gluetool.glue.GlueError*** – When somethign went wrong.
- ***gluetool.glue.GlueCommandError*** – When command exited with non-zero exit code.

exception gluetool.utils.**IncompatibleOptionsError**(*message*, *caused_by=None*, *sentry_fingerprint=None*, *sentry_tags=None*, ***kwargs*)
Bases: *gluetool.glue.SoftGlueError*

class gluetool.utils.**PatternMap**(*filepath*, *spices=None*, *logger=None*, *allow_variables=False*)
Bases: *gluetool.log.LoggerMixin*, `object`

Pattern map is a list of `<pattern>`: `<converter>` pairs. `Pattern` is a regular expression used to match a string, `converter` is a function that transforms a string into another one, accepting the pattern and the string as arguments.

It is defined in a YAML file:

```
---
- 'foo-(\d+)': 'bar-\1'
- 'baz-(\d+)': 'baz, find_the_most_recent, append_dot'
- 'bar-(\d+)':
  - 'bar, find_the_most_recent, append_dot'
  - 'bar, find_the_oldest, append_dot'
```

Patterns are the keys in each pair, while `converter` is a string (or list of strings), consisting of multiple items, separated by comma. The first item is **always** a string, let's call it `R`. `R`, given input string `S1` and the pattern, is used to transform `S1` to a new string, `S2`, by calling `pattern.sub(R, S1)`. `R` can make use of anything `re.sub()` supports, including capturing groups.

If there are other items in the `converter` string, they are names of *spices*, additional functions that will be called with `pattern` and the output of the previous spicing function, starting with `S2` in the case of the first *spice*.

To allow spicing, user of `PatternMap` class must provide *spice makers* - mapping between *spice* names and functions that generate spicing functions. E.g.:

```
def create_spice_append_dot(previous_spice):
    def _spice(pattern, s):
```

(continues on next page)

(continued from previous page)

```
        s = previous_spice(pattern, s)
        return s + '.'
    return _spice
```

`create_spice_append_dot` is a *spice maker*, used during creation of a pattern map after its definition is read, `_spice` is the actual spicing function used during the transformation process.

There can be multiple converters for a single pattern, resulting in multiple values returned when the input string matches the corresponding pattern.

Parameters

- **filepath** (*text*) – Path to a YAML file with map definition.
- **spices** (*dict*) – apping between *spices* and their *makers*.
- **logger** (*gluetool.log.ContextLogger*) – Logger used for logging.
- **allow_variables** (*bool*) – if set, both patterns and converters are first treated as templates, and as such are rendered before doing anything else. Map may contain special comments, `# !import-variables <path>`, where path refers to a YAML file providing the necessary variables.

match(*s*, *multiple=False*)

Try to match `s` by the map. If the match is found - the first one wins - then its conversions are applied to the `s`.

There can be multiple conversions for a pattern, by default only the product of the first one is returned. If `multiple` is set to `True`, list of all products is returned instead.

Return type text

Returns if matched, output of the corresponding transformation.

class gluetool.utils.**ProcessOutput**(*cmd*, *exit_code*, *stdout*, *stderr*, *kwargs*)

Bases: object

Result of external process.

log(*logger*)

log_stream(*stream*, *logger*)

class gluetool.utils.**SimplePatternMap**(*filepath*, *logger=None*, *allow_variables=False*)

Bases: *gluetool.log.LoggerMixin*, object

Pattern map is a list of `<pattern>: <result>` pairs. `Pattern` is a regular expression used to match a string, `result` is what the matching string maps to.

Basically an ordered dictionary with regexp matching of keys, backed by an YAML file.

Parameters

- **filepath** (*str*) – Path to a YAML file with map definition.
- **logger** (*gluetool.log.ContextLogger*) – Logger used for logging.
- **allow_variables** (*bool*) – if set, both patterns and converters are first treated as templates, and as such are rendered before doing anything else. Map may contain special comments, `# !import-variables <path>`, where path refers to a YAML file providing the necessary variables.

match(*s*)
Try to match `s` by the map. If the match is found - the first one wins - then its transformation is applied to the `s`.

Return type text

Returns if matched, output of the corresponding transformation.

class gluetool.utils.**StreamReader**(*stream*, *name=None*, *block=16*)
Bases: `object`

content

name

read()

wait()

class gluetool.utils.**ThreadAdapter**(*logger*, *thread*)
Bases: *gluetool.log.ContextAdapter*

Custom logger adapter, adding thread name as a context.

Parameters

- **logger** (gluetool.log.ContextAdapter) – parent logger whose methods will be used for logging.
- **thread** (*threading.Thread*) – thread whose name will be added.

gluetool.utils.**WaitCheckType**
alias of `typing.Callable`

class gluetool.utils.**WorkerThread**(*logger*, *fn*, *fn_args=None*, *fn_kwargs=None*, ***kwargs*)
Bases: *gluetool.log.LoggerMixin*, `threading.Thread`

Worker threads gets a job to do, and returns a result. It gets a callable, `fn`, which will be called in thread's `run()` method, and thread's `result` property will be the result - value returned by `fn`, or exception raised during the runtime of `fn`.

Parameters

- **logger** (gluetool.log.ContextAdapter) – logger to use for logging.
- **fn** – thread will start *fn* to do the job.
- **fn_args** – arguments for *fn*
- **fn_kwargs** – keyword arguments for *fn*

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

gluetool.utils.**YAML**(*loader_type=None*)
Provides YAML read/write interface with common settings.

Parameters **loader_type** (*str*) – type of YAML parser and loader. `None` or `rt` for round-trip (default), `safe`, `unsafe` or `base`.

Return type ruamel.yaml.YAML

gluetool.utils.**_json_byteify**(*data*, *ignore_dicts=False*)

gluetool.utils.**_load_yaml_variables**(*data*, *enabled=True*, *logger=None*)
Load all variables from files referenced by a YAML, and return function to render a string as a template using these variables. The files containing variables are mentioned in comments, in a form `# ! import-variables <filepath>` form.

Parameters

- **data** – data loaded from a YAML file.
- **enabled** (*bool*) – when set to `False`, variables are not loaded and a simple no-op function is returned.
- **logger** (*gluetool.log.ContextLogger*) – Logger used for logging.

Returns Function accepting a string and returning a rendered template.

class gluetool.utils.**cached_property**(*method*)
Bases: `object`

`property`-like decorator - at first access, it calls decorated method to acquire the real value, and then replaces itself with this value, making it effectively "cached". Useful for properties whose value does not change over time, and where getting the real value could penalize execution with unnecessary (network, memory) overhead.

Delete attribute to clear the cached value - on next access, decorated method will be called again, to acquire the real value.

Of possible options, only read-only instance attribute access is supported so far.

gluetool.utils.**check_for_commands**(*cmds*)
Checks if all commands in list cmds are valid

gluetool.utils.**deprecated**(*func*)
This is a decorator which can be used to mark functions as deprecated. It will result in a warning being emitted when the function is used.

gluetool.utils.**dict_update**(*dst*, **args*)
Python's `dict.update` does not return the dictionary just updated but a `None`. This function is a helper that does updates the dictionary *and* returns it. So, instead of:

```
d.update(other)
return d
```

you can use:

```
return dict_update(d, other)
```

Parameters

- **dst** (*dict*) – dictionary to be updated.
- **args** – dictionaries to update `dst` with.

gluetool.utils.**dump_yaml**(*data*, *filepath*, *logger=None*)
Save data stored in variable to YAML file.

Parameters

- **data** (*object*) – Data to store in YAML file
- **filepath** (*text*) – Path to an output file.

Raises ***gluetool.glue.GlueError*** – if it was not possible to successfully save data to file.

gluetool.utils.**fetch_url**(**args*, ***kwargs*)
"Get me content of this URL" helper.

Very thin wrapper around urllib. Added value is logging, and converting possible errors to *gluetool.glue.GlueError* exception.

Parameters

- **url** – URL to get.
- **logger** (*gluetool.log.ContextLogger*) – Logger used for logging.
- **success_codes** (*tuple*) – tuple of HTTP response codes representing successfull request.

Returns tuple `(response, content)` where `response` is what `requests.get()` returns, and `content` is the payload of the response.

gluetool.utils.**format_command_line**(*cmdline*)
Return formatted command-line.

All but the first line are indented by 4 spaces.

Parameters **cmdline** (*list*) – list of iterables, representing command-line split to multiple lines.

gluetool.utils.**from_json**(*json_string*)
Convert JSON in a string into Python data structures.

Similar to `json.loads()` but uses special object hook to avoid unicode strings in the output..

gluetool.utils.**from_yaml**(*yaml_string*, *loader_type=None*)
Convert YAML in a string into Python data structures.

Uses internal YAML parser to produce result. Paired with *load_yaml()* and their JSON siblings to provide unified access to JSON and YAML.

Parameters **loader_type** (*str*) – type of YAML parser and loader. `None` or `rt` for round-trip (default), `safe`, `unsafe` or `base`.

gluetool.utils.**load_json**(*filepath*, *logger=None*)
Load data stored in JSON file, and return their Python representation.

Parameters

- **filepath** (*text*) – Path to a file. `~` or `~<username>` are expanded before using.
- **logger** (*gluetool.log.ContextLogger*) – Logger used for logging.

Return type object

Returns structures representing data in the file.

Raises ***gluetool.glue.GlueError*** – if it was not possible to successfully load content of the file.

gluetool.utils.**load_yaml**(*filepath*, *loader_type=None*, *logger=None*)
Load data stored in YAML file, and return their Python representation.

Parameters

- **filepath** (*text*) – Path to a file. `~` or `~<username>` are expanded before using.
- **loader_type** (*str*) – type of YAML parser and loader. `None` or `rt` for round-trip (default), `safe`, `unsafe` or `base`.
- **logger** (*gluetool.log.ContextLogger*) – Logger used for logging.

Return type object

Returns structures representing data in the file.

Raises ***gluetool.glue.GlueError*** – if it was not possible to successfully load content of the file.

gluetool.utils.**new_xml_element**(*tag_name*, *_parent=None*, ***attrs*)

Create new XML element.

Parameters

- **tag_name** (*text*) – Name of the element.
- **_parent** (*element*) – If set, the newly created element will be appended to this element.
- **attrs** (*dict*) – Attributes to set on the newly created element.

Returns Newly created XML element.

gluetool.utils.**normalize_bool_option**(*option_value*)

Convert option value to Python's boolean.

`option_value` is what all those internal option processing return, which may be a default value set for an option, or what user passed in.

As switches, options with values can be used:

```
--foo=yes|no
--foo=true|false
--foo=1|0
--foo=Y|N
--foo=on|off
```

With combination of `store_true`/`store_false` and a default value module developer sets for the option, simple form without value is evaluated as easily. With `store_true` and `False` default, following option turn the feature *foo* on:

```
--enable-foo
```

With `store_false` and `True` default, following simple option turn the feature *foo* off:

```
--disable-foo
```

gluetool.utils.**normalize_multistring_option**(*option_value*, *separator=', '*)

Reduce string, representing comma-separated list of items, or possibly a list of such strings, to a simple list of items. Strips away the whitespace wrapping such items.

```
foo --option value1 --option value2, value3
foo --option value1,value2,value3
```

Or, when option is set by a config file:

```
option = value1
option = value1, value2,                    value3
```

After processing, different variants can be found when `option('option')` is called, `['value1', 'value2,value3']`, `['value1,value2,value3']`, `'value1'` and `value1, value2, value3`.

To reduce the necessary work, use this helper function to treat such option's value, and get simple `['value1', 'value2', 'value3']` structure.

gluetool.utils.**normalize_path**(*path*)
Apply common treatments on a given path:

- replace home directory reference (~ and similar), and
- convert `path` to a normalized absolutized version of the pathname.

gluetool.utils.**normalize_path_option**(*option_value*, *separator=', '*)
Reduce many ways how list of paths is specified by user, to a simple list of paths. See *normalize_multistring_option()* for more details.

gluetool.utils.**normalize_shell_option**(*option_value*)
Reduce string, using a shell-like syntax, or possibly a list of such strings, to a simple list of items. Strips away the whitespace wrapping such items.

```
foo --option value1 --option value2\ value3 --option "value4 value5"
```

Or, when option is set by a config file:

```
option = value1 value2\ value3 "value4 value5"
```

After processing, different variants can be found when `option('option')` is called, `['value1', 'value2,value3']`, `['value1,value2,value3']`, `'value1'` and `value1, value2, value3`.

To reduce the necessary work, use this helper function to treat such option's value, and get simple `['value1', 'value2 value3', 'value4 value5']` structure.

gluetool.utils.**render_template**(*template*, *logger=None*, ***kwargs*)
Render Jinja2 template. Logs errors, and raises an exception when it's not possible to correctly render the template.

Parameters

- **template** – Template to render. It can be either `jinja2.environment.Template` instance, or a string.
- **kwargs** (*dict*) – Keyword arguments passed to render process.

Returns Rendered template.

Raises *gluetool.glue.GlueError* – when the rednering failed.

gluetool.utils.**requests**(**args*, ***kwds*)
Wrap `requests` with few layers providing us with the logging and better insight into what has been happening when `requests` did their job.

Used as a context manager, yields a patched `requests` module. As long as inside the context, detailed information about HTTP traffic are logged via given logger.

Note: The original `requests` library is returned, with slight modifications for better integration with `gluetool` logging facilities. Each and every `requests` API feature is available and , hopefully, enhancements applied by this wrapper wouldn't interact with `requests` functionality.

```
with gluetool.utils.requests() as R:
    R.get(...).json()
    ...

    r = R.post(...)
```

(continues on next page)

(continued from previous page)

```
    assert r.code == 404
    ...
```

Parameters **logger** – used for logging.

Returns `requests` module.

`gluetool.utils.`**`run_command`**(**args*, ***kwargs*)

Wrapper for ‘‘Command(...).run().

Provided for backward compatibility.

Deprecated since version 1.1: Use *`gluetool.utils.Command`* instead.

`gluetool.utils.`**`treat_url`**(*url*, *logger=None*)

Remove “weird” artifacts from the given URL. Collapse adjacent ‘.’s, apply ‘..’, etc.

Parameters

- **url** (*text*) – URL to clear.
- **logger** (`gluetool.log.ContextAdapter`) – logger to use for logging.

Return type text

Raises gluetool.glue.GlueError: if URL is invalid

Returns Treated URL.

`gluetool.utils.`**`wait`**(*label*, *check*, *timeout=None*, *tick=30*, *logger=None*)

Wait for a condition to be true.

Parameters

- **label** (*text*) – printable label used for logging.
- **check** (*callable*) – called to test the condition. It must be of type *`WaitCheckType`*: takes no arguments, must return instance of `gluetool.Result`. If the result is valid, the condition is assumed to pass the check and waiting ends.
- **timeout** (*int*) – fail after this many seconds. `None` means test forever.
- **tick** (*int*) – test condition every `tick` seconds.
- **logger** (`gluetool.log.ContextAdapter`) – parent logger whose methods will be used for logging.

Raises ***`gluetool.glue.GlueError`*** – when `timeout` elapses while condition did not pass the check.

Returns if the condition became true, the value returned by the `check` function is returned. It is unpacked from the `Result` returned by `check`.

3.7 gluetool.version module

3.8 gluetool.tests package

3.8.1 Submodules

3.8.2 Module contents

class gluetool.tests.**Bunch**(**kwargs*)

Bases: object

Object-like access to a dictionary - useful for many mock objects.

class gluetool.tests.**NonLoadingGlue**(*tool=None*, *sentry=None*)
Bases: *gluetool.glue.Glue*

Current Glue implementation loads modules and configs when instantiated, which makes it *really* hard to make assumptions of the state of its internals - they will always be spoiled by other modules, other external resources the tests cannot control. So, to overcome this I use this custom Glue class that disables loading of modules and configs on its instantiation.

_load_modules()

parse_args(**args*, ***kwargs*)
Public entry point to argument parsing. Child classes must implement this method, e.g. by calling *gluetool.glue.Configurable._parse_args()* which makes use of additional argparse.ArgumentParser options.

parse_config(**args*, ***kwargs*)
Public entry point to configuration parsing. Child classes must implement this method, e.g. by calling *gluetool.glue.Configurable._parse_config()* which requires list of paths.

gluetool.tests.**create_module**(*module_class*, *glue=None*, *glue_class=<class 'gluetool.tests.NonLoadingGlue'>*, *name='dummy-module'*, *add_shared=True*)

gluetool.tests.**create_yaml**(*tmpdir*, *name*, *data*)

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y