gluetool Documentation

Release 1.1.dev18+gf01b014

mvadkerti@redhat.com

Mar 29, 2018

Contents

Gluetool 1.1 (1.1.dev18+gf01b014)

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

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.

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.

`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*.

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):
...
```

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*>.

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).

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'
```

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 = 'example-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
```

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')
    ....
```

gluetool features

A comprehensive list of gluetool features, available helpers, tricks and tips. All the ways gluetool have to help module developers.

Core

Module and gluetool configuration

Configuration of gluetool and every module is gathered from different sources of different priorities, and merged into a single store, accessible by *option()* method. Configuration from later sources replaces values set by earlier sources, with lower priority. That way it is possible to combine multiple configuration files for a module, e.g. a generic site-wide configuration, with user-specific configuration overriding the global settings. Options specified on a command-line have the highest priority, overriding all configuration files.

Consider following example module - it has just a single option, whom, whose value is logged in a form of greeting. The option has a default value, unknown being:

```
from gluetool import Module

class M(Module):
    name = 'dummy-module'

    options = {
        'whom': {
           'default': 'unknown being'
        }
    }

    def execute(self):
        self.info('Hi, {}!'.format(self.option('whom')))
```

With a configuration file, `~/.gluetool.d/config/dummy-module`, you can change the value of `whom`:

```
[default]
whom = happz
```

As you can see, configuration file for `dummy-module` is loaded and `option()` method returns the correct value, `happz`.

Options specified on a command-line are merged into the store transparently, without any additional action necessary:

Todo

- re-record video because of `name` => `whom`
- `seealso`:
  - options definitions

See also:

core-config-files to see what configuration files are examined.

Configuration files

For every module - including `gluetool` itself as well - `gluetool` checks several possible sources of configuration, merging all information found into a single configuration store, which can be queried during runtime using *`option()`* method.

Configuration files follow simple INI format, with a single section called `[default]`, containing all options:

```
[default]
option-foo = value bar
```

Warning: Options can have short and long names (e.g. `-v` vs. `--verbose`). Configuration files are using **only** the long option names to propagate their values to `gluetool`. If you use a short name (e.g. `v = yes`), such setting won't affect `gluetool` behavior!

These files are checked for `gluetool` configuration:

- `/etc/gluetool.d/gluetool`
- `~/.gluetool.d/gluetool`
- `./.gluetool.d/gluetool`
- options specified on a command-line

These files are checked for module configuration:

- `/etc/gluetool.d/config/<module name>`
- `~/.gluetool.d/config/<module name>`
- `./.gluetool.d/config/<module name>`
- options specified on a command-line

If you're using a tool derived from `gluetool`, it may add its own set of directories, e.g. using its name insead of `gluetool`, but lists mentioned above should be honored by such tool anyway, to stay compatible with the base `gluetool`.

It is possible to change the list of directories, using `--module-config-path` option, the default list mentioned above is then replaced by directories provided by this option.

Todo

- `seealso`:
  - option definitions

See also:

core-config-store for more information on configuration handling.

core-module-aliases for more information on module names and how to rename them

Module aliases

Each module has a name, as set by its `name` class attribute, but sometimes it might be good to use the module under another name. Remember, the module configuration is loaded from files named just like the module, and if there's a way to "rename" module when used in different pipelines, user might use different configuration files for the same module.

Consider following example module - it has just a single option, `whom`, whose value is logged in a form of greeting:

```
from gluetool import Module

class M(Module):
    name = 'dummy-module'

     options = {
         'whom': {}
     }

     def execute(self):
         self.info('Hi, {}!'.format(self.option('whom')))
```

With the following configuration, `~/.gluetool.d/config/dummy-module`, it will greet your users in a more friendly fashion:

```
[default]
whom = handsome gluetool user
```

For some reason, you might wish to use the module in another pipeline, sharing the configuration between both pipelines, but you want to change the greeted entity. One option is to use a command-line option, which overrides configuration files but that would make one of your pipelines a bit exceptional, having some extra command-line stuff. Other way is to tell `gluetool` to use the module but give it a different name. Add the extra configuration file for your "renamed" module, `~/.gluetool.d/config/customized-dummy-module`:

```
[default]
whom = beautiful
```

Module named `customized-dummy-module:dummy-module` does not exist but this form tells `gluetool` it should create an instance of `dummy-module` module, and name it `customized-dummy-module`. This is **the** name used to find and load module's configuration.

You may combine aliases and original modules as much as you wish - `gluetool` will keep track of names and the actual modules, and it will load the correct configuration:

Todo

- re-record video because of `name` => `whom`

Evaluation context

`gluetool` and its modules rely heavily on separating code from configuration, offloading things to easily editable files instead of hard-coding them into module sources. Values in configuration files can often be seen as templates, which need a bit of "polishing" to fill in missing bits that depend on the actual state of a pipeline and resources it operates on. To let modules easily participate and use information encapsulated in other modules in the pipeline, `gluetool` uses concept called *evaluation context* - a module can provide a set of variables it thinks might be interesting to other modules. These variables are collected over all modules in the pipeline, and made available as a "context", mapping of variable names and their values, which is a form generaly understood by pretty much any functionality that evaluates things, like templating engines.

To provide evaluation context, module has to define a property named *`eval_context`*. This property should return a mapping of variable names and their values.

For example:

```
from gluetool import Module
from gluetool.utils import render_template

class M(Module):
    name = 'dummy-module'

    @property
    def eval_context(self):
        return {
            'FOO': id(self)
        }

    def execute(self):
        context = self.shared('eval_context')
        self.info('Known variables: {}'.format(', '.join(context.keys())))

        message = render_template('Absolutely useless ID of this module is {{ FOO }}',
↪ **context)

        self.info(message)
```

It provides an interesting information to other modules - named `FOO` - for use in templates and other forms of runtime evaluation. To get access to the global context, collected from all modules, shared function `eval_context` is called.

Expected output:

```
[12:48:41] [+] [dummy-module] Known variables: FOO, ENV
[12:48:41] [+] [dummy-module] Absolutely useless ID of this module is 139695598692432
```

Note: Modules are asked to provide their context in the same order they are listed in the pipeline, and their contexts are merged, after each query, into a single mapping. It is therefore easy to overwrite variables provided by modules that were queried earlier by simply providing the same variable with a different value.

Note: It is a good practice to prefix names of provided variables, to make them module specific and avoid confusion when it comes to names that might be considered too generic. E.g. variable `ID` is probably way too universal - is it a user ID, or a task ID? Instead, `USER_ID` or `ARTIFACT_OWNER_ID` is much better.

Todo

- `seealso`:
  - rendering templates

Long and short option names

When specifying options on a command-line, each option can be set using its name: `--foo` for option named `foo`. Historicaly, it is also common to use "short" variants of option names, using just a single character. For example, `--help` and `-h` control the same thing. By default, each option defined by a module is a "long" one, suitable for use in a `--foo` form. If developer wishes to enable short form as well, he can simply express this wish by using both variants when defining the option, grouping them in a tuple.

Consider following example module - it has just a single option, `whom`, whose value is logged in a form of greeting. It is possible to use `--whom` or `-w` to control the value.

```
from gluetool import Module

class M(Module):
    name = 'dummy-module'

    options = {
        ('w', 'whom'): {}
    }

    def execute(self):
        self.info('Hi, {}!'.format(self.option('whom')))
```

Note: Configuration files deal with "long" option names only. I.e. `whom = handsome` will be correctly propagated into module's configuration store while `w = handsome` won't.

Todo

Features yet to describe:

- system-level, user-level and local dir configs
- configurable list of module paths (with default based on sys.prefix)
- dry-run support

- controled by core
- module can check what level is set, and take corresponding action. core takes care of logging
- exception hierarchy
- hard vs soft errors
- chaining supported
- custom sentry fingerprint and tags
- Failure class to pass by internally
- processes config file, command line options
- argparser to configure option
- option groups
- required options
- note to print as a part of help
- shared functions
- overloaded shared
- require_shared
- module logging helpers
- sanity => execute => destroy - pipeline flow
- failure access
- module discovery mechanism

Help

`gluetool` tries hard to simplify writing of consistent and useful help for modules, their shared functions, options and, of course, a source code. Its markup syntax of choice is reStructured (reST), which is being used in all docstrings. Sphinx is then used to generate documentation from documents and source code.

Module help

Every module supports a command-line option `-h` or `--help` that prints information on module's usage on terminal. To provide as much information on module's "public API", several sources are taken into account when generating the overall help for the module. Use of reST syntaxt is supported by each of them, that should allow authors to highligh important bits or syntax.

module's docstring Developer should describe module's purpose, use cases, configuration files and their syntax. Bear in mind that this is the text an end user would read to find out how to use the module, how to configure it and what they should expect from it. Feel free to use reST to include code blocks, emphasize importat bits and so on.

module's options Every option module has should have its own help set, using `help` key. These texts are gathered.

module's shared functions If the module provides shared functions, their signatures and help texts are gathered.

module's evaluation context If the module provides an evaluation context, description for each of its variables is extracted.

All parts are put together, formatted properly, and printed out to terminal in response to `--help` option.

Example:

```
from gluetool import Module

class M(Module):
    """
    This module greets its user.

    See ``--whom`` option.
    """

    name = 'dummy-module'

    options = {
        'whom': {
            'help': 'Greet our caller, whose NAME we are told by this option.',
            'default': 'unknown being',
            'metavar': 'NAME'
        }
    }

    shared_functions = ('hello',)

    def hello(self, name):
        """
        Say "Hi!" to someone.

        :param str name: Name of entity we're supposed to greet.
        """

        self.info('Hi, {}!'.format(name))

    @property
    def eval_context(self):
        __content__ = {
            'NAME': 'Name of entity this module should greet.'
        }

        return {
            'NAME': self.option('whom')
        }

    def execute(self):
        self.hello(self.option('whom'))
```

Todo

- run example with a `gluetool` supporting eval context help
- `seealso`:
  - module options
  - shared functions
  - shared functions help
  - eval context

- help colors

Todo

Features yet to describe:

- modules, shared functions, etc. help strings generated from their docstrings
- options help from their definitions in self.options
- RST formatting supported and evaluated before printing
- colorized to highlight RST
- keeps track of terminal width, tries to fit in

Logging

Early debug messages

Default logging level is set to `INFO`. While debugging actions happening early in pipeline workflow, like module discovery and loading, it may be useful to enable more verbose logging. Unfortunatelly, this feature is controlled by a `debug` option, and this option will be taken into account too late to shed light on your problem. For that case, it is possible to tell `gluetool` to enable debug logging right from its beginning, by setting an environment variable `GLUETOOL_DEBUG` to any value:

```
export GLUETOOL_DEBUG=does-not-matter
gluetool -l
```

As you can see, `gluetool` dumps much more verbose logging messages - about processing of options, config files and other stuff - on terminal with the variable set.

Note: You can set the variable in any way supported by your shell, session or environment in general. The only important thing is that such variable must exist when `gluetool` starts.

Logging of structured data

To format structured data, like lists, tuples and dictionaries, for output, use *gluetool.log.format_dict()*

Example:

```
import gluetool

print gluetool.log.format_dict([1, 2, (3, 4)])
```

Output:

```
[
    1,
    2,
    [
        3,
```

```
        4,
    ]
]
```

To actually log structured data, the *gluetool.log.log_dict()* helper is a nice shortcut.

Example:

```
import gluetool

logger = gluetool.log.Logging.create_logger()

gluetool.log.log_dict(logger.info, 'logging structured data', [1, 2, (3, 4)])
```

Output:

```
[14:43:03] [+] logging structured data:
[
    1,
    2,
    [
        3,
        4
    ]
]
```

The first parameter of `log_dict` expects a callback which is given the formatted data to actually log them. It is therefore easy to use `log_dict` on every level of your code, e.g. in methods of your module, just give it proper callback, like `self.info`.

Todo

- `seealso`:
  - logging helpers
  - connecting loggers

See also:

log-blob to find out how to log text blobs.

gluetool.log.format_dict()*, *gluetool.log.log_dict() for developer documentation.

Logging of unstructured blobs of text

To format a "blob" of text, without any apparent structure other than new-lines and similar markings, use `gluetool.log.format_blob()`:

It will preserve text formatting over multiple lines, and it will add borders to allow easy separation of the blob from neighbouring text.

To actually log a blob of text, *gluetool.log.log_blob()* is a shortcut:

The first parameter of `log_blob` expects a callback which is given the formatted data to actually log them. It is therefore easy to use `log_blob` on every level of your code, e.g. in methods of your module, just give it proper callback, like `self.info`.

Todo

- seealso:
  - logging helpers
  - connecting loggers

See also:

log-dict to find out how to log structured data.

gluetool.log.format_blob(), ***gluetool.log.log_blob()*** for developer documentation.

Logging of XML elements

To format an XML element, use `gluetool.log.format_xml()`:

It will indent nested elements, presenting the tree in a more readable form.

To actually log an XML element, *`gluetool.log.log_xml()`* is a shortcut:

The first parameter of `log_xml` expects a callback which is given the formatted data to actually log them. It is therefore easy to use `log_xml` on every level of your code, e.g. in methods of your module, just give it proper callback, like `self.info`.

Todo

- seealso:
  - logging helpers
  - connecting loggers

See also:

log-dict to find out how to log structured data.

gluetool.log.format_blob(), ***gluetool.log.log_blob()*** for developer documentation.

Object logging helpers

Note: When we talk about logger, we mean it as a description - an object that has logging methods we can use. It's not necessarilly the instance of `logging.Logger` - in fact, given how logging part of `gluetool` works, it is most likely it's an instance of `gluetool.logging.ContextAdapter`. But that is not important, the API - logging methods like `info` or `error` are available in such "logger" object, no matter what its class is.

Python's logging system provides a log function for each major log level, usually named by its corresponding level in lowercase, e.g. `debug` or `info`. These are reachable as methods of a logger (or logging context adapter) instance. If you have a class which is given a logger, to ease access to these methods, it is possible to "connect" the logger and your class, making logger's `debug` & co. direct members of your objects, allowing you to call `self.debug`, for example.

Example:

```
from gluetool.log import Logging, ContextAdapter

logger = ContextAdapter(Logging.create_logger())

class Foo(object):
    def __init__(self, logger):
        logger.connect(self)

Foo(logger).info('a message')
```

Output:

```
[10:01:15] [+] a message
```

All standard logging method `debug`, `info`, `warn`, `error` and `exception` are made available after connecting a logger.

Todo

- `seealso`:
  - context adapter

See also:

`logging.Logger.debug()` for logging methods.

Todo

Features yet to describe:

- clear separation of logging records, making it visible where each of them starts and what is a log message and what a logged blob of command output
- default log level controlled by env var
- warn(sentry=True)
- verbose, readable, formatted traceback logging
- using context adapters to add "structure" to loged messages
- colorized messages based on their level
- optional "log everything" dump in a file
- correct and readable logging of exception chains

Colorized output

`gluetool` uses awesome colorama library to enhance many of its outputs with colors. This is done in a transparent way, when developer does not need to think about it, and user can control this feature with a single option.

Control

Color support is disabled by default, and can be turned on using `--color` option:

If `colorama` package is **not** installed, color support cannot be turned on. If user tries to do that, `gluetool` will emit a warning:

Note: As of now, `colorama` is `gluetool`'s hard requirement, therefore it should not be possible - at least out of the box - to run `gluetool` wihout having `colorama` installed. However, this may change in the future, leaving this support up to user decision.

To control this feature programatically, see `gluetool.color.switch()`.

Todo

- `seealso`:
  - how to specify options

Colorized logs

Messages, logged on the terminal, are colorized based on their level:

`DEBUG` log level inherits default text color of your terminal, while, for example, `ERROR` is highlighted by being red, and `INFO` level is printed with nice, comforting green.

Todo

- `seealso`:
  - logging

Colorized help

`gluetool` uses reStructuredText (reST) to document modules, shared functions, opitons and other things, and to make the help texts even more readable, formatting, provided by reST, is enhanced with colors, to help users orient and focus on important information.

Todo

- `seealso`:
  - generic help
  - module help
  - option help

Colors in templates

Color support is available for templates as well, via `style` filter.

Example:

```
import gluetool

gluetool.log.Logging.create_logger()
gluetool.color.switch(True)

print gluetool.utils.render_template('{{ "foo" | style(fg="red", bg="green") }}')
```

See also:

rendering-templates for more information about rendering templates with `gluetool`.

Sentry integration

`gluetool` integrates easily with Sentry platform, simplifying the collection of trouble issues, code crashes, warnings and other important events your deployed code produces. This integration is optional - it must be explicitly enabled - and transparent - it is not necessary to report common events, like exceptions.

When enabled, every unhandled exception is automatically reported to Sentry. Helpers for explicit reporting of handled exceptions and warnings are available, as well as the bare method for reporting arbitrary events.

Control

Sentry integration is controlled by environmental variables. It must be possible to configure itilable even before `gluetool` has a chance to process given options. To enable Sentry integration, one has to set at least `SENTRY_DSN` variable:

```
export SENTRY_DSN="https://<key>:<secret>@sentry.io/<project>"
```

This variable tells Sentry-related code where it should report the events. Without this variable set, Sentry integration is disabled. All relevant functions still can be called but do not report any events to Sentry, since they don't know where to send their reports.

See also:

About the DSN for detaield information on Sentry DSN and their use.

`gluetool.sentry` module for developer documentation.

Sentry tags & environment variables

Sentry allows attaching "tags" to reported events. To use environment variables as such tags, set `SENTRY_TAG_MAP` variable. It lists comma-separated pairs of names, tag and its source variable.

```
export SENTRY_TAG_MAP="username=USER,hostname=HOSTNAME"
```

Should there be an event to report, integration code will attach 2 labels to it, `username` and `hostname`, using environmen variables `USER` and `HOSTNAME` respectively as source of values.

See also:

Tagging Events for detailed information on event tags.

Logging of submitted events

Integration code is able to log every reported event. To enable this feature, simply set `SENTRY_BASE_URL` environment variable to URL of the project `gluetool` is reporting events to. While `SENTRY_DSN` controls the whole integration and has its meaning within Sentry server your `gluetool` runs report to, `SENTRY_BASE_URL` is used only in a cosmetic way and `gluetool` code adds ID of reported event to it. The resulting URL, if followed, should lead to your project and the relevant event.

As you can see, the exception, raised by `gluetool` when there were no command-line options telling it what to do, has been submitted to Sentry, and immediately logged, with `ERROR` loglevel.

See also:

Sentry - Control for more information about Sentry integration.

Warnings

By default, only unhandled exceptions are submitted to Sentry. it is however possible, among others, to submit warnings, e.g. in case when such warning is good to capture yet it is not necessary to raise an exception and kill the whole `gluetool` pipeline. For that case, `warn` logging method accepts `sentry` keword parameter, which, when set to `True`, uses Sentry-related code to submit given message to the configured Sentry instance. It is also always logged like any other warning.

Example:

```
from gluetool.log import Logging

logger = Logging.create_logger()

logger.warn('foo', sentry=True)
```

Output:

```
[17:16:50] [W] foo
```

Todo

- video

See also:

Object logging helpers for more information on logging methods.

gluetool.log.warn_sentry() for developer documentation.

Todo

Features yet to describe:

- all env variables are attached to events (breadcrumbs)
- logging records are attached to events (breadcrumbs)
- URL of every reported event available for examination by code
- soft-error tag for failure.soft errors
- raised exceptions can provide custom fingerprints and tags

- submit_exception and submit_warning for explicit submissions
- logger.warn(sentry=True)

Utils

Rendering templates

`gluetool` and its modules make heavy use of Jinja2 templates. To help with their processing, it provides *`gluetool.utils.render_template()`* helper which accepts both raw string templates and instances of `jinja2.Template`, and renders it with given context variables. Added value is uniform logging of template and used variables.

Example:

```
import gluetool

gluetool.log.Logging.create_logger()

print gluetool.utils.render_template('Say hi to {{ user }}', user='happz')
```

Output:

```
Say hi to happz
```

See also:

Jinja2 templates for information about this fast & modern templating engine.

gluetool.utils.render_template() for developer documentation.

colors-in-templates for using colors in templates

Normalize URL

URLs, comming from different systems, or created by joining their parts, might contain redundant bits, duplicities, multiple `..` entries, mix of uppercase and lowercase characters and similar stuff. Such URLs are not verry pretty. To "prettify" your URLs, use *`gluetool.utils.treat_url()`*:

For example:

```
from gluetool.log import Logging
from gluetool.utils import treat_url

print treat_url('HTTP://FoO.bAr.coM././foo/././../foo/index.html')
```

Output:

```
http://foo.bar.com/foo/index.html
```

Todo

Features yet to describe:

- *dict_update*

- converting various command-line options to unified output
- boolean switches via normalize_bool_option
- multiple string values (–foo A,B –foo C => [A,B,C])
- path - expanduser & abspath applied
- multiple paths - like multiple string values, but normalized like above
- “worker thread” class - give it a callable, it will return its return value, taking care of catching exceptions
- running external apps via run_command
- Bunch object for grouping arbitrary data into a single object, or warping dictionary as an object (d[key] => d.key)
- cached_property decorator
- formatted logging of arbitrary command-line - if you have a command-line to format, we have a function for that
- fetch data from a given URL
- load data from YAML or JSON file or string
- write data structures as a YAML of JSON
- pattern maps
- waiting for things to finish
- creating XML elements
- checking whether external apps are available and runnable

Tool

Todo

Features yet to describe:

- reusable heart of gluetool
- config file on system level, user level or in a local dir

Todo

Features yet to describe:

- custom pylint checkers
- option names
- shared function definitions

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.

---

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.

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
```

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 :)

Module tests should be in the same file

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

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.

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.

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`.

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`.

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'])
```

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.

`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.

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.

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.

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`.

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.

Every module must have a description

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

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.

Every module must be documented

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

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.

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 {}
```

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.

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.
"""
```

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.

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>`.

Development

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 :)

Requirements

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

- `virtualenv` utility
- `ansible-playbook`

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:
...
```

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`).

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

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.**Configurable**

Bases: 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.

Return type dict

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

option(*name*)
Return a value of given option from module's configuration store.

Parameters **name** (*str*) – name of requested option.

Returns option value or `None` when no such option exists.

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>)`

- 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 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.**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

_member_map_ = OrderedDict([('DEFAULT', <DryRunLevels.DEFAULT: 0>), ('DRY', <DryRunLevels.DRY: 1>), ('ISC

_member_names_ = ['DEFAULT', 'DRY', 'ISOLATED']

_member_type_
alias of int

_value2member_map_ = {0: <DryRunLevels.DEFAULT: 0>, 1: <DryRunLevels.DRY: 1>, 2: <DryRunLevels.ISOLATE

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.

_add_shared(*funcname*, *module*, *func*)
Register a shared function. Overwrite previously registered function with the same name, if there was any such.

This is a helper method for easier testability. It is not a part of public API of this class.

Parameters

- **funcname** (*str*) – Name of the shared function.
- **module** (gluetool.glue.Module) – Module instance providing the shared function.
- **func** (*callable*) – Shared function.

_check_module_file(*mfile*)
Make sure the 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 **mfile** (*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.

_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

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

_import_module(*import_name*, *filename*)
Attempt to import a Python module from a file.

Parameters

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

Returns imported Python module.

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

_load_gluetool_modules(*group*, *module_name*, *filepath*)
Load `gluetool` modules from a file. Method attempts to import the file as a Python module, and then checks its content and adds all *gluetool* modules to internal module registry.

Parameters

- **group** (*str*) – module group.
- **module_name** (*str*) – name assigned to the imported Python module.
- **filepath** (*str*) – path to a file.

Return type [(module_group, module_class), ..]

Returns list of loaded `gluetool` modules

_load_module_path(*ppath*)
Search and load `gluetool` modules from a directory.

In essence, it scans every file with `.py` suffix, and searches for classes derived from *gluetool.glue.Module*.

Parameters **ppath** (*str*) – directory to search for *gluetool* modules.

_load_python_module(*group*, *module_name*, *filepath*)
Load Python module from a file, if it contains `gluetool` modules. 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

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

Returns loaded Python module.

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

add_shared(*funcname*, *module*)
Register a shared function. Overwrite previously registered function with the same name, if there was any such.

Parameters

- **funcname** (*str*) – Name of the shared function.
- **module** (`gluetool.glue.Module`) – Module instance providing the shared function.

del_shared(*funcname*)

destroy_modules(*failure=None*)

dryrun_level

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

Variables **ENV** (*dict*) – `os.environ`.

get_shared(*funcname*)

has_shared(*funcname*)

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.

load_modules()
Load all available *gluetool* modules.

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_group_list()
Returns a dictionary of groups of modules with description

module_list()

module_list_usage(*groups*)
Returns a string with modules description

module_paths
List of paths in which modules reside.

options = [('Global options', {('L', 'list-shared'): {'action': 'store_true', 'default': False, 'help': 'List all available share

parse_args(*args*)

parse_config(*paths*)

require_shared(**names*, ***kwargs*)

run_module(*module_name*, *module_argv=None*, *actual_module_name=None*, *register=False*)
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`.
- **register** (*bool*) – If `True`, module instance is added to a list of modules in this `Glue` instance, and it will be collected when *destroy_modules()* gets called.

run_modules(*pipeline_desc*, *register=False*)
Run a pipeline, consisting of multiple modules.

Parameters

- **pipeline_desc** (*list (PipelineStep)*) – List of pipeline steps.
- **register** (*bool*) – If `True`, module instance is added to a list of modules in this `Glue` instance, and it will be collected when *destroy_modules()* gets called.

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_warning (**args*, ***kwargs*)
Submits warnings 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*)

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

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*, ***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`.

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.

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.

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.

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.

exception gluetool.glue.**GlueRetryError**(*message*, *caused_by=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()

Register module's shared functions with Glue, to allow other modules to use them.

del_shared(*funcname*)

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*)

has_shared(*funcname*)

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

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 its action, e.g. to publish messages on a different message bus.

parse_args(*args*)

parse_config()

require_shared(**names*, ***kwargs*)

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

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

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

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

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

exception gluetool.glue.**SoftGlueError**(*message*, *caused_by=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, ..)

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 = '*=-~"+'

DummyTextBuilder.**config**
alias of *DummyConfig*

DummyTextBuilder.**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.**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.**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.

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
logger = Logging.create_logger()

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

# or connect it with current instance (if you're doing all this
# inside some class' constructor):
logger.connect(self)

# now you can access logger's methods directly:
self.debug('foo once again!')

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

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

# and, finally, create a root context logger - when we create another loggers during
# the code flow, this context logger will be in the root of this tree of loggers.
logger = ContextAdapter(logger)

# don't forget to re-connect with the context logger if you connected your instance
# with previous logger, to make sure helpers are set correctly
logger.connect(self)
```

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*)

Bases: `logging.LoggerAdapter`

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

"context" is any key in `extra` dictionary starting with `ctx_`, whose value is expected to be tuple of `(priority, value)`. 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.

Parameters

- **logger** (*logging.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.

connect(*parent*)

Create helper methods in `parent`, by assigning adapter's methods to its attributes. One can then call `parent.debug` and so on, instead of less readable `parent.logger.debug`.

Simply instantiate adapter and call its `connect` with an object as a `parent` argument, and the object will be enhanced with all these logging helpers.

Parameters **parent** – object to enhance with logging helpers.

process(*msg*, *kwargs*)

Original `process` overwrites `kwargs['extra']` which doesn't work for us - we want to chain adapters, getting more and more contexts on the way. Therefore `update` instead of assignment.

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

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

static _close_output_file()
If opened, close output file used for logging.

This method is registered with atexit.

static create_logger(*output_file=None*, *level=20*, *sentry=None*, *sentry_submit_warning=None*)
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

- **output_file** (*str*) – if set, new handler will be attached to the logger, streaming messages of **all** log levels into this this file.
- **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.
- **sentry_submit_warning** (*callable*) – if set, it is used by warning methods of derived loggers to submit warning to the Sentry server, if asked by a caller to do so.

Return type logging.Logger

Returns a logging.Logger instance, set up for logging.

static get_logger()
Returns a logger instance.

Expects there was a call to *create_logger()* method before calling this method that would actually create and set up the logger.

Return type logging.Logger

Returns a logging.Logger instance, set up for logging, or None when there's no logger yet.

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

output_file = None
If enabled, handles output to catch-everything file.

output_file_handler = None

stderr_handler = None
Stream handler printing out to stderr.

class gluetool.log.**LoggingFormatter**(*colors=True*, *log_tracebacks=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*)

_level_color = {40: <function <lambda>>, 50: <function <lambda>>, 20: <function <lambda>>, 30: <function <lamb
Colorizers assigned to loglevels

_level_tags = {5: 'V', 40: 'E', 10: 'D', 50: 'C', 20: '+', 30: 'W'}
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** (*logging.Logger*) – parent logger this adapter modifies.
- **module** (gluetool.glue.Module) – module whose name is added as a context.

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.**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_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.**verbose_adapter**(*self*, *message*, **args*, ***kwargs*)

gluetool.log.**verbose_logger**(*self*, *message*, **args*, ***kwargs*)

gluetool.log.**warn_sentry**(*self*, *message*, **args*, ***kwargs*)
Beside calling the original the `warning` method (stored as `self.orig_warning`), this one also submits warning to the Sentry server when asked to do so by a keyword argument `sentry` set to `True`.

gluetool.proxy module

Proxying object wrapper.

class gluetool.proxy.**Proxy**(*obj*)
Bases: object

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__', '__contains__', '__delitem__

gluetool.tool module

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

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

_cleanup(*failure=None*)
Clear Glue pipeline by calling modules' destroy methods.

_command_name

_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(str)*) – List of known module names.

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

_exit_logger
Return logger for use when finishing the gluetool pipeline.

_handle_failure(*failure*)

_handle_failure_core(*failure*)

_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**()

gluetool.utils module

Various helpers.

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

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

class gluetool.utils.**PatternMap**(*filepath*, *spices=None*, *logger=None*)
Bases: 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):
        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** (*str*) – Path to a YAML file with map definition.
- **spices** (*dict*) – apping between *spices* and their *makers*.
- **logger** (*gluetool.log.ContextLogger*) – Logger used for logging.

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 str

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*)
Bases: 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.

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 str

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.

class gluetool.utils.**WorkerThread**(*logger*, *fn*, *fn_args=None*, *fn_kwargs=None*, ***kwargs*)
Bases: 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()

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

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.**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** (*str*) – Path to an output file.

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

gluetool.utils.**fetch_url**(*url*, *logger=None*, *success_codes=(200,)*)
"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** (*str*) – 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 `urllib2.urlopen()` 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*)
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.

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

Parameters

- **filepath** (*str*) – 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*, *logger=None*)
Load data stored in YAML file, and return their Python representation.

Parameters

- **filepath** (*str*) – 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.`**new_xml_element**(*tag_name*, *_parent=None*, ***attrs*)
Create new XML element.

Parameters

- **tag_name** (*str*) – 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.**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.

Return type str

Returns Rendered template.

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

`gluetool.utils.`**run_command**(*cmd*, *logger=None*, *inspect=False*, *inspect_callback=None*, ***kwargs*)

Run external command, and return it's exit code and output.

This is a very thin and simple wrapper above `subprocess.Popen`, and its main purpose is to log everything that happens before and after execution. All additional arguments are passed directly to *Popen* constructor.

If `stdout` or `stderr` keyword arguments are not specified, function will set them to `subprocess.PIPE`, to capture both output streams in separate strings.

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

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

run_command(['/bin/foo'], 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 calls of `run_command`.

Parameters

- **cmd** (*list*) – command to execute.
- **logger** (`gluetool.log.ContextAdapter`) – parent logger whose methods will be used for logging.
- **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 process.

Raises

- ***gluetool.glue.GlueError*** – when command was not found.
- ***gluetool.glue.GlueCommandError*** – when command exited with non-zero exit code.
- **Exception** – when anything else breaks.

gluetool.utils.**treat_url**(*url*, *logger=None*)
Remove "weird" artifacts from the given URL. Collapse adjacent '.'s, apply '..', etc.

Parameters

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

Return type str

Returns Treated URL.

gluetool.utils.**wait**(*label*, *check*, *timeout=None*, *tick=30*, *logger=None*)
Wait for a condition to be true.

Parameters

- **label** (*str*) – printable label used for logging.
- **check** (*callable*) – called to test the condition. If its return value evaluates as `True`, the condition is assumed to pass the test 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.

gluetool.version module

gluetool.tests package

Submodules

gluetool.tests.conftest module

gluetool.tests.test_core module

gluetool.tests.test_error module

gluetool.tests.test_json module

gluetool.tests.test_load_yaml module

gluetool.tests.test_new_xml_element module

gluetool.tests.test_normalize_option module

gluetool.tests.test_pipeline_step module

gluetool.tests.test_render_template module

gluetool.tests.test_run_command module

gluetool.tests.test_shared module

gluetool.tests.test_treat_url module

gluetool.tests.test_utils module

gluetool.tests.test_wait module

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*)

parse_config(**args*, ***kwargs*)

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

L

M

N

O

P

R

S

T

U

V

W