Zambeze

Release 0.1

Oak Ridge National Laboratory

Mar 06, 2023

API REFERENCE

The source code for the Zambeze's Python package is available on GitHub. Our preferred channel to report a bug or request a feature is via Zambeze's Github Issues Track.

You can also reach the Zambeze team via our support email: support@zambeze.org.

CHAPTER

ONE

GETTING STARTED

In order to set up and start using Zambeze, you will need the following dependencies:

- A running rabbitmq server, see rabbitmq documentation. RabbitMQ does not need admin privileges to install and run on Linux systems.
- Python >= 3.9. We suggest using Anaconda.
- (Optional) ImageMagick to try a first mock example use-case. It is available here.

1.1 Installing and Testing Zambeze

1. Checkout or Download the latest release from the Github.
2. From the terminal in the zambeze directory run the following:

   pip install -e .
3. Ensure that you have a RabbitMQ server is running and you are able to reach it. You may verify this by telnet'ing the rabbitmq server like so:

   telnet <rabbitmq.server> 5672
4. Navigate to the examples directory and run the python program *imagemagick_files.py* like so:

   python3 imagemagick_files.py

   If all goes well, the command will finish with no errors and you will see a new file called *a.gif* in your home directory.

CHAPTER

TWO

USER API REFERENCE

The user API reference targets users who want to use Zambeze's Python package for running their .

class zambeze.campaign.activities.shell.**ShellActivity**(*name: str*, *files: list[str]*, *command: str*, *arguments: list[str]*, *logger: Logger*, *campaign_id: Optional[str] = None*, *agent_id: Optional[str] = None*, *message_id: Optional[str] = None*, ***kwargs*)

Bases: `Activity`

A Unix Shell script/command activity.

Parameters

- **name** (*str*) – Campaign activity name.
- **files** (*Optional[list[str]]*) – List of file URIs.
- **command** (*Optional[str]*) – Action's command.
- **arguments** (*Optional[list[str]]*) – List of arguments.
- **logger** (*Optional[logging.Logger]*) – The logger where to log information/warning or errors.

activity_id: Optional[str]

agent_id: Optional[str]

arguments: list[str]

campaign_id: Optional[str]

command: Optional[str]

files: list[str]

generate_message() → AbstractMessage

logger: Optional[Logger]

message_id: Optional[str]

class zambeze.campaign.campaign.**Campaign**(*name: str*, *activities: list[zambeze.campaign.activities.abstract_activity.Activity] = []*, *logger: Optional[Logger] = None*)

Bases: `object`

A Scientific Campaign.

Parameters

- **name** (*str*) – The campaign name.
- **activities** (*Optional[list[Activity]]*) – List of activities.
- **logger** (*Optional[logging.Logger]*) – The logger where to log information/warning or errors.

add_activity(*activity: Activity*) → None

Add an activity to the campaign.

Parameters

activity (*Activity*) – An activity object.

dispatch() → None

Dispatch the set of current activities in the campaign.

CHAPTER

THREE

DEVELOPER API

class zambeze.orchestration.plugins.**PluginChecks**(*val: dict*)

error_detected() → bool

Detects if an error was found in the results of the plugin checks

class zambeze.orchestration.plugins.**Plugins**(*logger: Optional[Logger] = None*)

Plugins class takes care of managing all plugins.

Plugins can be added as plugins by creating packages in the plugin_modules

Parameters

logger (*Optional[logging.Logger]*) – The logger where to log information/warning or errors.

check(*msg: AbstractMessage*, *arguments: Optional[dict] = None*) → *PluginChecks*

check(*msg: str*, *arguments: dict = {}*) → *PluginChecks*

Check that the arguments passed to the plugin "plugin_name" are valid

Parameters

- **plugin_name** (*str*) – the name of the plugin to validate against
- **arguments** (*dict*) – the arguments to be validated for plugin "plugin_name"

Returns

What is returned are a list of the plugins and their actions along with an indication on whether there was a problem with them

Example

Using rsync

For the rsync plugin to be useful, both the local and remote host ssh keys must have been configured. By default the rsync plugin will look for the private key located at ~/.ssh/id_rsa. If the private key is different then it must be specified with the "private_ssh_key" key value pair.

plugins = Plugins() config = {

"rsync": {
"private_ssh_key": "path to private ssh key" }

}

plugins.configure(config) arguments = {

"transfer": {

"source": {
"ip": local_ip, user": current_user, "path": current_valid_path, },

“destination”: {
“ip”: “172.22.1.69”, “user”: “cades”, “path”: “/home/cades/josh-testing”, },

“arguments”: [“-a”], }

} checked_args = plugins.check(“rsync”, arguments) print(checked_args) >> { >> “rsync”: { “transfer”: (True, “”) } >> }

`configure`(*config: dict*)

Configuration options for each plugin

This method is responsible for initializing all the plugins that are supported in the plugin_modules folder. It should be called before the plugins can be run, all plugins should be configured before they can be run.

Parameters
`config` (*dict*) – This contains relevant configuration information for each plugin, if provided will only configure the plugins listed

Example
Arguments

The configuration options for each plugin will appear under their name in the configuration parameter.

I.e. for plugins ‘globus’ and ‘shell’

config = {

‘globus’: {

‘authentication flow’: {
‘type’: ‘credential flow’, ‘secret’: “blahblah” },

‘shell’: {
‘arguments’ : [‘’] } }

plugins = Plugins() plugins.configure(config, [‘shell’])

This will just configure the “shell” plugin

`property configured: list[str]`

Will return a list of all the plugins that have been configured.

Returns
list of all plugins that are ready to be run

Return type
list[str]

Example
If nothing has been configured

plugins = Plugins() assert len(plugins.configured) == 0

Example
If globus is configured

config = {

“globus”: {
“client id”: “…” }

}

plugins.configure(config) assert len(plugins.configured) == 1 assert “globus” in plugins.configured

property info: dict

Will return the current state of the registered plugins

Parameters
plugins (*list[str]*) – the plugins to provide information about defaults to information about all plugins

Returns
The actual information of each plugin that was specified

Return type
dict

Example

these_plugins = [“globus”, “shell”] plugins.configure(configuration_options) information = plugins.info(these_plugins) print(information)

{
“globus”: {...} “shell”: {...}

}

property registered: list[str]

List all plugins that have been registered.

This method can be called at any time and is meant to simply display which packages are supported and present in the plugin_modules folder. It does not mean that these plugins have been configured. All plugins must be configured before they can be run.

Returns
Returns the names of all the plugins that have been registered

Return type
list[str]

run(*msg: AbstractMessage*, *arguments: Optional[dict] = None*) → None

run(*msg: str*, *arguments: dict = {}*) → None

Run a specific plugins.

Parameters

- **plugin_name** (*str*) – Plugin name
- **arguments** (*dict*) – Plugin arguments

Example

plugins = Plugins() config = {

“rsync”: {
“ssh_key”: “path to private ssh key” }

}

plugins.configure(config) arguments = {

“transfer”: {

“source”: {
“ip”: “hostname”: “path”: },

“destination”: {
“ip”: “hostname”: “path”: }

}

}

Should return True for each action that was found to be # correctly validated checks = plugins.check('rsync', arguments) print(checks) # Should print { "rsync": { "transfer": True } } plugins.run('rsync', arguments)

3.1 Abstract Plugin API

class zambeze.orchestration.plugin_modules.abstract_plugin.**Plugin**(*name: str*, *logger: Optional[Logger] = None*)

Abstract base class for ensuring that all registered plugins have the same interface

Parameters

logger (*Optional[logging.Logger]*) – The logger where to log information/warning or errors.

abstract check(*arguments: list[dict]*) → list[dict]

Determine if the proposed arguments can be executed by this instance.

Parameters

arguments – The arguments are checked to ensure their types and

formats are valid :type arguments: list[dict] :return: Returns the list of actions that are vaid :rtype: list[dict] with the actions valid actions listed with bool set to True and invalid ones False, along with a message.

Example

```
>>> arguments =
>>> [
>>>     { "action1": { "dothis": ...} },
>>>     { "action2": { "dothat": ...} },
>>> ]
>>> checked_actions = plugin.check(arguments)
>>> for action in checked_actions:
>>>     print(f"{action}: {checked_actions[action]}")
>>> # Should print
>>> # action1 True, ""
>>> # action2 False, "This was the problem"
```

abstract configure(*config: dict*) → None

Configure this set up the plugin.

abstract property info: dict

This method is to be used after configuration step and will return information about the plugin such as configuration settings and defaults.

property name: str

Returns the name of the plugin.

The name of the plugin, should be lower case

Returns

Name of the plugin

Return type
string

abstract process(*arguments: list[dict]*) → dict
Will run the plugin with the provided arguments

3.2 GitHub Plugin API

3.3 Globus Plugin API

3.4 Rsync Plugin API

PYTHON MODULE INDEX

Z

INDEX

R

S

Z