Zambeze

Release 0.2

Oak Ridge National Laboratory

Apr 25, 2024

GETTING STARTED

Zambeze is a distributed task orchestration system to orchestrate science application tasks across the large-scale and edge computing systems.

The source code for 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

CONTRIBUTING

Formatting and linting tools for zambeze development are discussed below.

2.1 Linting

Here.

2.2 Formatting

Python source code is formatted using Black. A GitHub action has been enabled that will automatically check if your code is in compliance with Black. An action is also enabled so that you can format the code automatically by commenting in an open pr with the text: `@par-hermes format`.

CHAPTER

THREE

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 | None = None*, *campaign_id: str | None = None*, *origin_agent_id: str | None = None*, *message_id: str | None = 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.

generate_message() → AbstractMessage

class zambeze.campaign.campaign.**Campaign**(*name: str*, *activities: list[Activity] | None = None*, *logger: Logger | None = None*, *force_login: bool = False*)

Bases: `object`

A Scientific Campaign class to manage and dispatch campaign activities.

name

The name of the campaign.

Type
str

campaign_id

A unique identifier for the campaign.

Type
str

needs_globus_login

A flag indicating if Globus login is needed.

Type
bool

`activities`

A list of activities associated with the campaign.

Type
list[Activity]

`_logger`

Logger for logging information, warnings, and errors.

Type
logging.Logger

`add_activity`(*activity: Activity*) → None

Adds an activity to the campaign.

Parameters
activity (*Activity*) – The activity to add to the campaign.

`dispatch`() → None

Dispatches the set of current activities in the campaign.

`result`()

`status`()

CHAPTER

FOUR

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: Logger | None = None*)

Plugins class takes care of managing all plugins.

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

Parameters

logger – The logger where to log information/warning or errors.

check(*msg: Activity*, *arguments: list = []*) → *PluginChecks*

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

Parameters

- **msg** (*Activity*) – 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.

Return type

PluginChecks

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

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 of str

Examples

If nothing has been configured

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

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 of 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
The names of all the plugins that have been registered.

Return type
list of str

run(*msg: AbstractMessage*, *arguments: dict | None = None*) → None

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

Run a specific plugins.

Parameters

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

4.1 Abstract Plugin API

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

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

Parameters

- **name** – The name.
- **logger** – The logger where to log information/warning or errors.

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

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

Parameters

arguments – Arguments are checked to ensure their types and formats are valid.

Returns

- *List of actions that are valid. Valid actions are True, invalid ones*
- *are False, along with a message.*

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

Name of the plugin, should be lower case.

Return type

Name of the plugin.

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

Will run the plugin with the provided arguments.

4.2 Rsync Plugin API

PYTHON MODULE INDEX

Z

INDEX

Symbols

A

C

D

E

G

I

M

N

P

R

S

Z