Python Jenkins Documentation

Release 1.8.0

Ken Conley, James Page, Tully Foote, Matthew Gertner

Mar 24, 2023

Contents

CHAPTER 1

README

Python Jenkins is a python wrapper for the Jenkins REST API which aims to provide a more conventionally pythonic way of controlling a Jenkins server. It provides a higher-level API containing a number of convenience functions.

We like to use python-jenkins to automate our Jenkins servers. Here are some of the things you can use it for:

- Create new jobs
- Copy existing jobs
- Delete jobs
- Update jobs
- Get a job's build information
- Get Jenkins master version information
- Get Jenkins plugin information
- Start a build on a job
- Create nodes
- Enable/Disable nodes
- Get information on nodes
- Create/delete/reconfig views
- Put server in shutdown mode (quiet down)
- List running builds
- Delete builds
- Wipeout job workspace
- Create/delete/update folders[1]

[1] The free Cloudbees Folders Plugin provides support for a subset of the full folders functionality. For the complete capabilities you will need the paid for version of the plugin.

- Set the next build number[2]
- Install plugins
- and many more..

To install:

```
$ sudo python setup.py install
```

Online documentation:

- http://python-jenkins.readthedocs.org/en/latest/

1.1 Developers

Bug report:

- https://bugs.launchpad.net/python-jenkins

Repository:

- https://opendev.org/jjb/python-jenkins

Cloning:

- git clone https://opendev.org/jjb/python-jenkins

Patches are submitted via Gerrit at:

- https://review.opendev.org/#/q/project:jjb/python-jenkins

Please do not submit GitHub pull requests, they will be automatically closed.

The python-jenkins developers communicate in the `#openstack-jjb` channel on Freenode's IRC network.

More details on how you can contribute is available on our wiki at:

- https://docs.opendev.org/opendev/infra-manual/latest/developers.html

1.2 Writing a patch

Be sure that you lint code before created an code review. The easiest way to do this is to install git pre-commit hooks.

1.3 Installing without setup.py

Then install the required python packages using pip:

```
$ sudo pip install python-jenkins
```

[2] The Next Build Number Plugin provides support for setting the next build number.

CHAPTER 2

Contents

2.1 API reference

See examples at *Using Python-Jenkins*

exception jenkins.**JenkinsException**
General exception type for jenkins-API-related failures.

exception jenkins.**NotFoundException**
A special exception to call out the case of receiving a 404.

exception jenkins.**EmptyResponseException**
A special exception to call out the case receiving an empty response.

exception jenkins.**BadHTTPException**
A special exception to call out the case of a broken HTTP response.

exception jenkins.**TimeoutException**
A special exception to call out in the case of a socket timeout.

class jenkins.**WrappedSession**
A wrapper for requests.Session to override 'verify' property, ignoring REQUESTS_CA_BUNDLE environment variable.

This is a workaround for https://github.com/kennethreitz/requests/issues/3829 (will be fixed in requests 3.0.0)

merge_environment_settings(*url*, *proxies*, *stream*, *verify*, **args*, ***kwargs*)
Check the environment and merge it with some settings.

Return type dict

class jenkins.**Jenkins**(*url*, *username=None*, *password=None*, *timeout=<object object>*)
Create handle to Jenkins instance.

All methods will raise *JenkinsException* on failure.

Parameters

- **url** – URL of Jenkins server, str

- **username** – Server username, str
- **password** – Server password, str
- **timeout** – Server connection timeout in secs (default: not set), int

maybe_add_crumb(*req*)

get_job_info(*name*, *depth=0*, *fetch_all_builds=False*)
Get job information dictionary.

Parameters

- **name** – Job name, str
- **depth** – JSON depth, int
- **fetch_all_builds** – If true, all builds will be retrieved from Jenkins. Otherwise, Jenkins will only return the most recent 100 builds. This comes at the expense of an additional API call which may return significant amounts of data. bool

Returns dictionary of job information

get_job_info_regex(*pattern*, *depth=0*, *folder_depth=0*, *folder_depth_per_request=10*)

Get a list of jobs information that contain names which match the regex pattern.

Parameters

- **pattern** – regex pattern, str
- **depth** – JSON depth, int
- **folder_depth** – folder level depth to search int
- **folder_depth_per_request** – Number of levels to fetch at once, int. See *get_all_jobs()*.

Returns List of jobs info, list

get_job_name(*name*)
Return the name of a job using the API.

That is roughly an identity method which can be used to quickly verify a job exists or is accessible without causing too much stress on the server side.

Parameters **name** – Job name, str

Returns Name of job or None

debug_job_info(*job_name*)
Print out job info in more readable format.

jenkins_open(*req*, *add_crumb=True*, *resolve_auth=True*)
Return the HTTP response body from a requests.Request.

Returns str

jenkins_request(*req*, *add_crumb=True*, *resolve_auth=True*)
Utility routine for opening an HTTP request to a Jenkins server.

Parameters

- **req** – A requests.Request to submit.
- **add_crumb** – If True, try to add a crumb header to this req before submitting. Defaults to True.

- **resolve_auth** – If True, maybe add authentication. Defaults to `True`.

Returns A `requests.Response` object.

get_queue_item(*number*, *depth=0*)
Get information about a queued item (to-be-created job).

The returned dict will have a "why" key if the queued item is still waiting for an executor.

The returned dict will have an "executable" key if the queued item is running on an executor, or has completed running. Use this to determine the job number / URL.

Parameters **name** – queue number, `int`

Returns dictionary of queued information, `dict`

get_build_info(*name*, *number*, *depth=0*)
Get build information dictionary.

Parameters

- **name** – Job name, `str`
- **number** – Build number, `str` (also accepts `int`)
- **depth** – JSON depth, `int`

Returns dictionary of build information, `dict`

Example:

```
>>> next_build_number = server.get_job_info('build_name')['nextBuildNumber']
>>> output = server.build_job('build_name')
>>> from time import sleep; sleep(10)
>>> build_info = server.get_build_info('build_name', next_build_number)
>>> print(build_info)
{u'building': False, u'changeSet': {u'items': [{u'date': u'2011-12-
↪19T18:01:52.540557Z', u'msg': u'test', u'revision': 66, u'user': u'unknown',
↪ u'paths': [{u'editType': u'edit', u'file': u'/branches/demo/index.html'}]}
↪], u'kind': u'svn', u'revisions': [{u'module': u'http://eaas-svn01.i3.
↪level3.com/eaas', u'revision': 66}]}, u'builtOn': u'', u'description': None,
↪ u'artifacts': [{u'relativePath': u'dist/eaas-87-2011-12-19_18-01-57.war', u
↪'displayPath': u'eaas-87-2011-12-19_18-01-57.war', u'fileName': u'eaas-87-
↪2011-12-19_18-01-57.war'}, {u'relativePath': u'dist/eaas-87-2011-12-19_18-
↪01-57.war.zip', u'displayPath': u'eaas-87-2011-12-19_18-01-57.war.zip', u
↪'fileName': u'eaas-87-2011-12-19_18-01-57.war.zip'}], u'timestamp':␣
↪1324317717000, u'number': 87, u'actions': [{u'parameters': [{u'name': u
↪'SERVICE_NAME', u'value': u'eaas'}, {u'name': u'PROJECT_NAME', u'value': u
↪'demo'}]}, {u'causes': [{u'userName': u'anonymous', u'shortDescription': u
↪'Started by user anonymous'}]}, {}, {}, {}], u'id': u'2011-12-19_18-01-57',␣
↪u'keepLog': False, u'url': u'http://eaas-jenkins01.i3.level3.com:9080/job/
↪build_war/87/', u'culprits': [{u'absoluteUrl': u'http://eaas-jenkins01.i3.
↪level3.com:9080/user/unknown', u'fullName': u'unknown'}], u'result': u
↪'SUCCESS', u'duration': 8826, u'fullDisplayName': u'build_war #87'}
```

get_build_env_vars(*name*, *number*, *depth=0*)
Get build environment variables.

Parameters

- **name** – Job name, `str`
- **number** – Build number, `str` (also accepts `int`)
- **depth** – JSON depth, `int`

Returns dictionary of build env vars, `dict` or None for workflow jobs, or if InjectEnvVars plugin not installed

get_build_test_report(*name*, *number*, *depth=0*)
Get test results report.

Parameters

- **name** – Job name, `str`
- **number** – Build number, `str` (also accepts `int`)

Returns dictionary of test report results, `dict` or None if there is no Test Report

get_build_artifact(*name*, *number*, *artifact*)
Get artifacts from job

Parameters

- **name** – Job name, `str`
- **number** – Build number, `str` (also accepts `int`)
- **artifact** – Artifact relative path, `str`

Returns artifact to download, `dict`

get_build_stages(*name*, *number*)
Get stages info from job

Parameters

- **name** – Job name, `str`
- **number** – Build number, `str` (also accepts `int`)

Returns dictionary of stages in the job, `dict`

get_queue_info()

Returns list of job dictionaries, `[dict]`

Example::

```
>>> queue_info = server.get_queue_info()
>>> print(queue_info[0])
{u'task': {u'url': u'http://your_url/job/my_job/', u'color': u'aborted_
↪anime', u'name': u'my_job'}, u'stuck': False, u'actions': [{u'causes': [
↪{u'shortDescription': u'Started by timer'}]}], u'buildable': False, u
↪'params': u'', u'buildableStartMilliseconds': 1315087293316, u'why': u
↪'Build #2,532 is already in progress (ETA:10 min)', u'blocked': True}
```

cancel_queue(*id*)
Cancel a queued build.

Parameters **id** – Jenkins job id number for the build, `int`

get_info(*item=''*, *query=None*)
Get information on this Master or item on Master.

This information includes job list and view information and can be used to retreive information on items such as job folders.

Parameters

- **item** – item to get information about on this Master

- **query** – xpath to extract information about on this Master

Returns dictionary of information about Master or item, `dict`

Example:

```
>>> info = server.get_info()
>>> jobs = info['jobs']
>>> print(jobs[0])
{u'url': u'http://your_url_here/job/my_job/', u'color': u'blue',
u'name': u'my_job'}
```

get_whoami(*depth=0*)

Get information about the user account that authenticated to Jenkins. This is a simple way to verify that your credentials are correct.

Returns Information about the current user `dict`

Example:

```
>>> me = server.get_whoami()
>>> print me['fullName']
>>> 'John'
```

get_version()

Get the version of this Master.

Returns This master's version number `str`

Example:

```
>>> info = server.get_version()
>>> print info
>>> 1.541
```

get_plugins_info(*depth=2*)

Get all installed plugins information on this Master.

This method retrieves information about each plugin that is installed on master returning the raw plugin data in a JSON format.

Deprecated since version 0.4.9: Use *get_plugins()* instead.

Parameters **depth** – JSON depth, `int`

Returns info on all plugins `[dict]`

Example:

```
>>> info = server.get_plugins_info()
>>> print(info)
[{u'backupVersion': None, u'version': u'0.0.4', u'deleted': False,
u'supportsDynamicLoad': u'MAYBE', u'hasUpdate': True,
u'enabled': True, u'pinned': False, u'downgradable': False,
u'dependencies': [], u'url':
u'http://wiki.jenkins-ci.org/display/JENKINS/Gearman+Plugin',
u'longName': u'Gearman Plugin', u'active': True, u'shortName':
u'gearman-plugin', u'bundled': False}, ..]
```

get_plugin_info(*name*, *depth=2*)

Get an installed plugin information on this Master.

This method retrieves information about a specific plugin and returns the raw plugin data in a JSON format. The passed in plugin name (short or long) must be an exact match.

Note: Calling this method will query Jenkins fresh for the information for all plugins on each call. If you need to retrieve information for multiple plugins it's recommended to use *get_plugins()* instead, which will return a multi key dictionary that can be accessed via either the short or long name of the plugin.

Parameters

- **name** – Name (short or long) of plugin, `str`
- **depth** – JSON depth, `int`

Returns a specific plugin `dict`

Example:

```
>>> info = server.get_plugin_info("Gearman Plugin")
>>> print(info)
{u'backupVersion': None, u'version': u'0.0.4', u'deleted': False,
u'supportsDynamicLoad': u'MAYBE', u'hasUpdate': True,
u'enabled': True, u'pinned': False, u'downgradable': False,
u'dependencies': [], u'url':
u'http://wiki.jenkins-ci.org/display/JENKINS/Gearman+Plugin',
u'longName': u'Gearman Plugin', u'active': True, u'shortName':
u'gearman-plugin', u'bundled': False}
```

get_plugins(*depth=2*)

Return plugins info using helper class for version comparison

This method retrieves information about all the installed plugins and uses a Plugin helper class to simplify version comparison. Also uses a multi key dict to allow retrieval via either short or long names.

When printing/dumping the data, the version will transparently return a unicode string, which is exactly what was previously returned by the API.

Parameters **depth** – JSON depth, `int`

Returns info on all plugins `[dict]`

Example:

```
>>> j = Jenkins()
>>> info = j.get_plugins()
>>> print(info)
{('gearman-plugin', 'Gearman Plugin'):
  {u'backupVersion': None, u'version': u'0.0.4',
   u'deleted': False, u'supportsDynamicLoad': u'MAYBE',
   u'hasUpdate': True, u'enabled': True, u'pinned': False,
   u'downgradable': False, u'dependencies': [], u'url':
   u'http://wiki.jenkins-ci.org/display/JENKINS/Gearman+Plugin',
   u'longName': u'Gearman Plugin', u'active': True, u'shortName':
   u'gearman-plugin', u'bundled': False}, ...}
```

get_jobs(*folder_depth=0*, *folder_depth_per_request=10*, *view_name=None*)

Get list of jobs.

Each job is a dictionary with 'name', 'url', 'color' and 'fullname' keys.

If the `view_name` parameter is present, the list of jobs will be limited to only those configured in the specified view. In this case, the job dictionary 'fullname' key would be equal to the job name.

Parameters

- **folder_depth** – Number of levels to search, `int`. By default 0, which will limit search to toplevel. None disables the limit.
- **folder_depth_per_request** – Number of levels to fetch at once, `int`. See *get_all_jobs()*.
- **view_name** – Name of a Jenkins view for which to retrieve jobs, `str`. By default, the job list is not limited to a specific view.

Returns list of jobs, `[{str: str, str: str, str: str, str: str}]`

Example:

```
>>> jobs = server.get_jobs()
>>> print(jobs)
[{
    u'name': u'all_tests',
    u'url': u'http://your_url.here/job/all_tests/',
    u'color': u'blue',
    u'fullname': u'all_tests'
}]
```

get_all_jobs(*folder_depth=None*, *folder_depth_per_request=10*)

Get list of all jobs recursively to the given folder depth.

Each job is a dictionary with 'name', 'url', 'color' and 'fullname' keys.

Parameters

- **folder_depth** – Number of levels to search, `int`. By default None, which will search all levels. 0 limits to toplevel.
- **folder_depth_per_request** – Number of levels to fetch at once, `int`. By default 10, which is usually enough to fetch all jobs using a single request and still easily fits into an HTTP request.

Returns list of jobs, `[ { str: str} ]`

Note: On instances with many folders it would not be efficient to fetch each folder separately, hence *folder_depth_per_request* levels are fetched at once using the `tree` query parameter:

```
?tree=jobs[url,color,name,jobs[...,jobs[...,jobs[...,jobs]]]]
```

If there are more folder levels than the query asks for, Jenkins returns empty[1] objects at the deepest level:

```
{"name": "folder", "url": "...", "jobs": [{}, {}, ...]}
```

This makes it possible to detect when additional requests are needed.

---

copy_job(*from_name*, *to_name*)

Copy a Jenkins job.

Will raise an exception whenever the source and destination folder for this jobs won't be the same.

[1] Actually recent Jenkins includes a `_class` field everywhere, but it's missing the requested fields.

Parameters

- **from_name** – Name of Jenkins job to copy from, str
- **to_name** – Name of Jenkins job to copy to, str

Throws *JenkinsException* whenever the source and destination folder are not the same

rename_job(*from_name*, *to_name*)
Rename an existing Jenkins job

Will raise an exception whenever the source and destination folder for this jobs won't be the same.

Parameters

- **from_name** – Name of Jenkins job to rename, str
- **to_name** – New Jenkins job name, str

Throws *JenkinsException* whenever the source and destination folder are not the same

delete_job(*name*)
Delete Jenkins job permanently.

Parameters **name** – Name of Jenkins job, str

enable_job(*name*)
Enable Jenkins job.

Parameters **name** – Name of Jenkins job, str

disable_job(*name*)
Disable Jenkins job.

To re-enable, call *Jenkins.enable_job()*.

Parameters **name** – Name of Jenkins job, str

set_next_build_number(*name*, *number*)
Set a job's next build number.

The current next build number is contained within the job information retrieved using *Jenkins.get_job_info()*. If the specified next build number is less than the last build number, Jenkins will ignore the request.

Note that the Next Build Number Plugin must be installed to enable this functionality.

Parameters

- **name** – Name of Jenkins job, str
- **number** – Next build number to set, int

Example:

```
>>> next_bn = server.get_job_info('job_name')['nextBuildNumber']
>>> server.set_next_build_number('job_name', next_bn + 50)
```

job_exists(*name*)
Check whether a job exists

Parameters **name** – Name of Jenkins job, str

Returns True if Jenkins job exists

jobs_count()
Get the number of jobs on the Jenkins server

Returns Total number of jobs, `int`

assert_job_exists(*name*, *exception_message='job[%s] does not exist'*)
Raise an exception if a job does not exist

Parameters

- **name** – Name of Jenkins job, `str`
- **exception_message** – Message to use for the exception. Formatted with `name`

Throws *JenkinsException* whenever the job does not exist

create_folder(*folder_name*, *ignore_failures=False*)
Create a new Jenkins folder

Parameters

- **folder_name** – Name of Jenkins Folder, `str`
- **ignore_failures** – if True, don't raise if it was not possible to create the folder, `bool`

upsert_job(*name*, *config_xml*)
Create a new Jenkins job or reconfigures it if it exists

Parameters

- **name** – Name of Jenkins job, `str`
- **config_xml** – config file text, `str`

check_jenkinsfile_syntax(*jenkinsfile*)
Checks if a Pipeline Jenkinsfile has a valid syntax

Parameters **jenkinsfile** – Jenkinsfile text, `str`

Returns List of errors in the Jenkinsfile. Empty list if no errors.

create_job(*name*, *config_xml*)
Create a new Jenkins job

Parameters

- **name** – Name of Jenkins job, `str`
- **config_xml** – config file text, `str`

get_job_config(*name*)
Get configuration of existing Jenkins job.

Parameters **name** – Name of Jenkins job, `str`

Returns job configuration (XML format)

reconfig_job(*name*, *config_xml*)
Change configuration of existing Jenkins job.

To create a new job, see *Jenkins.create_job()*.

Parameters

- **name** – Name of Jenkins job, `str`
- **config_xml** – New XML configuration, `str`

build_job_url(*name*, *parameters=None*, *token=None*)
Get URL to trigger build job.

Authenticated setups may require configuring a token on the server side.

Use `list of two membered tuples` to supply parameters with multi select options.

Parameters

- **name** – Name of Jenkins job, `str`
- **parameters** – parameters for job, or None., `dict` or `list of two membered tuples`
- **token** – (optional) token for building job, `str`

Returns URL for building job

build_job(*name*, *parameters=None*, *token=None*)

Trigger build job.

This method returns a queue item number that you can pass to *Jenkins.get_queue_item()*. Note that this queue number is only valid for about five minutes after the job completes, so you should get/poll the queue information as soon as possible to determine the job's URL.

Parameters

- **name** – name of job
- **parameters** – parameters for job, or `None`, `dict`
- **token** – Jenkins API token

Returns `int` queue item

run_script(*script*, *node=None*)

Execute a groovy script on the jenkins master or on a node if specified..

Parameters

- **script** – The groovy script, `string`
- **node** – Node to run the script on, defaults to None (master).

Returns The result of the script run.

Example::

```
>>> info = server.run_script("println(Jenkins.instance.pluginManager.
↪plugins)")
>>> print(info)
u'[Plugin:windows-slaves, Plugin:ssh-slaves, Plugin:translation,
Plugin:cvs, Plugin:nodelabelparameter, Plugin:external-monitor-job,
Plugin:mailer, Plugin:jquery, Plugin:antisamy-markup-formatter,
Plugin:maven-plugin, Plugin:pam-auth]'
```

install_plugin(*name*, *include_dependencies=True*)

Install a plugin and its dependencies from the Jenkins public repository at http://repo.jenkins-ci.org/repo/org/jenkins-ci/plugins

Parameters

- **name** – The plugin short name, `string`
- **include_dependencies** – Install the plugin's dependencies, `bool`

Returns Whether a Jenkins restart is required, `bool`

Example::

```
>>> info = server.install_plugin("jabber")
>>> print(info)
True
```

stop_build(*name*, *number*)
Stop a running Jenkins build.

Parameters

- **name** – Name of Jenkins job, str
- **number** – Jenkins build number for the job, int

delete_build(*name*, *number*)
Delete a Jenkins build.

Parameters

- **name** – Name of Jenkins job, str
- **number** – Jenkins build number for the job, int

wipeout_job_workspace(*name*)
Wipe out workspace for given Jenkins job.

Parameters **name** – Name of Jenkins job, str

get_running_builds()
Return list of running builds.

Each build is a dict with keys 'name', 'number', 'url', 'node', and 'executor'.

Returns List of builds, [{ str: str, str: int, str:str, str: str, str: int}]

Example::

```
>>> builds = server.get_running_builds()
>>> print(builds)
[{'node': 'foo-slave', 'url': 'https://localhost/job/test/15/',
  'executor': 0, 'name': 'test', 'number': 15}]
```

get_nodes(*depth=0*)
Get a list of nodes connected to the Master

Each node is a dict with keys 'name' and 'offline'

Returns List of nodes, [{ str: str, str: bool}]

get_node_info(*name*, *depth=0*)
Get node information dictionary

Parameters

- **name** – Node name, str
- **depth** – JSON depth, int

Returns Dictionary of node info, dict

node_exists(*name*)
Check whether a node exists

Parameters **name** – Name of Jenkins node, str

Returns `True` if Jenkins node exists

assert_node_exists(*name*, *exception_message='node[%s] does not exist'*)
Raise an exception if a node does not exist

Parameters

- **name** – Name of Jenkins node, `str`
- **exception_message** – Message to use for the exception. Formatted with `name`

Throws *JenkinsException* whenever the node does not exist

delete_node(*name*)
Delete Jenkins node permanently.

Parameters **name** – Name of Jenkins node, `str`

disable_node(*name*, *msg=''*)
Disable a node

Parameters

- **name** – Jenkins node name, `str`
- **msg** – Offline message, `str`

enable_node(*name*)
Enable a node

Parameters **name** – Jenkins node name, `str`

create_node(*name*, *numExecutors=2*, *nodeDescription=None*, *remoteFS='/var/lib/jenkins'*, *labels=None*, *exclusive=False*, *launcher='hudson.slaves.CommandLauncher'*, *launcher_params={}*)
Create a node

Parameters

- **name** – name of node to create, `str`
- **numExecutors** – number of executors for node, `int`
- **nodeDescription** – Description of node, `str`
- **remoteFS** – Remote filesystem location to use, `str`
- **labels** – Labels to associate with node, `str`
- **exclusive** – Use this node for tied jobs only, `bool`
- **launcher** – The launch method for the slave, `jenkins.LAUNCHER_COMMAND`, `jenkins.LAUNCHER_SSH`, `jenkins.LAUNCHER_JNLP`, `jenkins.LAUNCHER_WINDOWS_SERVICE`
- **launcher_params** – Additional parameters for the launcher, `dict`

get_node_config(*name*)
Get the configuration for a node.

Parameters **name** – Jenkins node name, `str`

reconfig_node(*name*, *config_xml*)
Change the configuration for an existing node.

Parameters

- **name** – Jenkins node name, `str`

- **config_xml** – New XML configuration, str

get_build_console_output(*name*, *number*)
Get build console text.

Parameters

- **name** – Job name, str
- **number** – Build number, str (also accepts int)

Returns Build console output, str

get_view_name(*name*)
Return the name of a view using the API.

That is roughly an identity method which can be used to quickly verify a view exists or is accessible without causing too much stress on the server side.

Parameters **name** – View name, str

Returns Name of view or None

assert_view_exists(*name*, *exception_message='view[%s] does not exist'*)
Raise an exception if a view does not exist

Parameters

- **name** – Name of Jenkins view, str
- **exception_message** – Message to use for the exception. Formatted with name

Throws *JenkinsException* whenever the view does not exist

view_exists(*name*)
Check whether a view exists

Parameters **name** – Name of Jenkins view, str

Returns True if Jenkins view exists

get_views()
Get list of views running.

Each view is a dictionary with 'name' and 'url' keys.

Returns list of views, [{ str: str}]

delete_view(*name*)
Delete Jenkins view permanently.

Parameters **name** – Name of Jenkins view, str

create_view(*name*, *config_xml*)
Create a new Jenkins view

Parameters

- **name** – Name of Jenkins view, str
- **config_xml** – config file text, str

reconfig_view(*name*, *config_xml*)
Change configuration of existing Jenkins view.

To create a new view, see *Jenkins.create_view()*.

Parameters

- **name** – Name of Jenkins view, str
- **config_xml** – New XML configuration, str

get_view_config(*name*)
Get configuration of existing Jenkins view.

Parameters **name** – Name of Jenkins view, str

Returns view configuration (XML format)

get_promotion_name(*name*, *job_name*)
Return the name of a promotion using the API.

That is roughly an identity method which can be used to quickly verify a promotion exists for a job or is accessible without causing too much stress on the server side.

Parameters

- **name** – Promotion name, str
- **job_name** – Job name, str

Returns Name of promotion or None

assert_promotion_exists(*name*, *job_name*, *exception_message='promotion[%s] does not exist for job[%s]'*)
Raise an exception if a job lacks a promotion

Parameters

- **name** – Name of Jenkins promotion, str
- **job_name** – Job name, str
- **exception_message** – Message to use for the exception. Formatted with name and job_name

Throws *JenkinsException* whenever the promotion does not exist on a job

promotion_exists(*name*, *job_name*)
Check whether a job has a certain promotion

Parameters

- **name** – Name of Jenkins promotion, str
- **job_name** – Job name, str

Returns True if Jenkins promotion exists

get_promotions_info(*job_name*, *depth=0*)
Get promotion information dictionary of a job

Parameters

- **job_name** – job_name, str
- **depth** – JSON depth, int

Returns Dictionary of promotion info, dict

get_promotions(*job_name*)
Get list of promotions running.

Each promotion is a dictionary with 'name' and 'url' keys.

Parameters **job_name** – Job name, str

Returns list of promotions, `[ { str: str} ]`

delete_promotion(*name*, *job_name*)
Delete Jenkins promotion permanently.

Parameters

- **name** – Name of Jenkins promotion, `str`
- **job_name** – Job name, `str`

create_promotion(*name*, *job_name*, *config_xml*)
Create a new Jenkins promotion

Parameters

- **name** – Name of Jenkins promotion, `str`
- **job_name** – Job name, `str`
- **config_xml** – config file text, `str`

reconfig_promotion(*name*, *job_name*, *config_xml*)
Change configuration of existing Jenkins promotion.

To create a new promotion, see *Jenkins.create_promotion()*.

Parameters

- **name** – Name of Jenkins promotion, `str`
- **job_name** – Job name, `str`
- **config_xml** – New XML configuration, `str`

get_promotion_config(*name*, *job_name*)
Get configuration of existing Jenkins promotion.

Parameters

- **name** – Name of Jenkins promotion, `str`
- **job_name** – Job name, `str`

Returns promotion configuration (XML format)

assert_folder(*name*, *exception_message='job[%s] is not a folder'*)
Raise an exception if job is not Cloudbees Folder

Parameters

- **name** – Name of job, `str`
- **exception_message** – Message to use for the exception.

Throws *JenkinsException* whenever the job is not Cloudbees Folder

is_folder(*name*)
Check whether a job is Cloudbees Folder

Parameters **name** – Job name, `str`

Returns `True` if job is folder, `False` otherwise

assert_credential_exists(*name*, *folder_name*, *domain_name='_'*, *exception_message='credential[%s] does not exist in the domain[%s] of [%s]'*)
Raise an exception if credential does not exist in domain of folder

Parameters

- **name** – Name of credential, `str`
- **folder_name** – Folder name, `str`
- **domain_name** – Domain name, default is '_', `str`
- **exception_message** – Message to use for the exception. Formatted with `name`, `domain_name`, and `folder_name`

Throws *JenkinsException* whenever the credentail does not exist in domain of folder

credential_exists(*name*, *folder_name*, *domain_name='_'*)
Check whether a credentail exists in domain of folder

Parameters

- **name** – Name of credentail, `str`
- **folder_name** – Folder name, `str`
- **domain_name** – Domain name, default is '_', `str`

Returns `True` if credentail exists, `False` otherwise

get_credential_info(*name*, *folder_name*, *domain_name='_'*)
Get credential information dictionary in domain of folder

Parameters

- **name** – Name of credentail, `str`
- **folder_name** – folder_name, `str`
- **domain_name** – Domain name, default is '_', `str`

Returns Dictionary of credential info, `dict`

get_credential_config(*name*, *folder_name*, *domain_name='_'*)
Get configuration of credential in domain of folder.

Parameters

- **name** – Name of credentail, `str`
- **folder_name** – Folder name, `str`
- **domain_name** – Domain name, default is '_', `str`

Returns Credential configuration (XML format)

create_credential(*folder_name*, *config_xml*, *domain_name='_'*)
Create credentail in domain of folder

Parameters

- **folder_name** – Folder name, `str`
- **config_xml** – New XML configuration, `str`
- **domain_name** – Domain name, default is '_', `str`

delete_credential(*name*, *folder_name*, *domain_name='_'*)
Delete credential from domain of folder

Parameters

- **name** – Name of credentail, `str`

- **folder_name** – Folder name, str
- **domain_name** – Domain name, default is '_', str

reconfig_credential(*folder_name*, *config_xml*, *domain_name='_'*)
Reconfig credential with new config in domain of folder

Parameters

- **folder_name** – Folder name, str
- **config_xml** – New XML configuration, str
- **domain_name** – Domain name, default is '_', str

list_credentials(*folder_name*, *domain_name='_'*)
List credentials in domain of folder

Parameters

- **folder_name** – Folder name, str
- **domain_name** – Domain name, default is '_', str

Returns Credentials list, list

quiet_down()
Prepare Jenkins for shutdown.

No new builds will be started allowing running builds to complete prior to shutdown of the server.

wait_for_normal_op(*timeout*)
Wait for jenkins to enter normal operation mode.

Parameters **timeout** – number of seconds to wait, int Note this is not the same as the connection timeout set via __init__ as that controls the socket timeout. Instead this is how long to wait until the status returned.

Returns True if Jenkins became ready in time, False otherwise.

Setting timeout to be less than the configured connection timeout may result in this waiting for at least the connection timeout length of time before returning. It is recommended that the timeout here should be at least as long as any set connection timeout.

class jenkins.plugins.**Plugin**(**args*, ***kwargs*)
Dictionary object containing plugin metadata.

Populates dictionary using json object input.

accepts same arguments as python *dict* class.

class jenkins.plugins.**PluginVersion**(*version*)
Class providing comparison capabilities for plugin versions.

Parse plugin version and store it for comparison.

2.2 Using Python-Jenkins

The python-jenkins library allows management of a Jenkins server through the Jenkins REST endpoints. Below are examples to get you started using the library. If you need further help take a look at the *API reference* docs for more details.

2.2.1 Example 1: Get version of Jenkins

This is an example showing how to connect to a Jenkins instance and retrieve the Jenkins server version.

```
import jenkins

server = jenkins.Jenkins('http://localhost:8080', username='myuser', password=
↪'mypassword')
user = server.get_whoami()
version = server.get_version()
print('Hello %s from Jenkins %s' % (user['fullName'], version))
```

The above code prints the fullName attribute of the user and the version of the Jenkins master running on 'localhost:8080'. For example, it may print "Hello John from Jenkins 2.0".

From Jenkins version 1.426 onward you can specify an API token instead of your real password while authenticating the user against the Jenkins instance. Refer to the Jenkins Authentication wiki for details about how you can generate an API token. Once you have an API token you can pass the API token instead of a real password while creating a Jenkins instance.

2.2.2 Example 2: Logging into Jenkins using kerberos

Kerberos support is only enabled if you have "kerberos" python package installed. You can install the "kerberos" package from PyPI using the obvious pip command.

```
pip install kerberos
```

Note: This might require python header files as well as kerberos header files.

If you have "kerberos" python package installed, python-jenkins tries to authenticate using kerberos automatically when the Jenkins server replies "401 Unauthorized" and indicates it supports kerberos. That is, kerberos authentication should work automagically. For a quick test, just try the following.

```
import jenkins

server = jenkins.Jenkins('http://localhost:8080')
print server.jobs_count()
```

Note: Jenkins as such does not support kerberos, it needs to be supported by the Servlet container or a reverse proxy sitting in front of Jenkins.

2.2.3 Example 3: Working with Jenkins Jobs

This is an example showing how to create, configure and delete Jenkins jobs.

```
server.create_job('empty', jenkins.EMPTY_CONFIG_XML)
jobs = server.get_jobs()
print jobs
my_job = server.get_job_config('cool-job')
print(my_job) # prints XML configuration
server.build_job('empty')
```

(continues on next page)

(continued from previous page)

```
server.disable_job('empty')
server.copy_job('empty', 'empty_copy')
server.enable_job('empty_copy')
server.reconfig_job('empty_copy', jenkins.RECONFIG_XML)

server.delete_job('empty')
server.delete_job('empty_copy')

# build a parameterized job
# requires creating and configuring the api-test job to accept 'param1' & 'param2'
server.build_job('api-test', {'param1': 'test value 1', 'param2': 'test value 2'})
last_build_number = server.get_job_info('api-test')['lastCompletedBuild']['number']
build_info = server.get_build_info('api-test', last_build_number)
print build_info

# get all jobs from the specific view
jobs = server.get_jobs(view_name='View Name')
print jobs
```

2.2.4 Example 4: Working with Jenkins Views

This is an example showing how to create, configure and delete Jenkins views.

```
server.create_view('EMPTY', jenkins.EMPTY_VIEW_CONFIG_XML)
view_config = server.get_view_config('EMPTY')
views = server.get_views()
server.delete_view('EMPTY')
print views
```

2.2.5 Example 5: Working with Jenkins Plugins

This is an example showing how to retrieve Jenkins plugins information.

```
plugins = server.get_plugins_info()
print plugins
```

The above example will print a dictionary containing all the plugins that are installed on the Jenkins server. An example of what you can expect from the `get_plugins_info()` method is documented in the *API reference* doc.

2.2.6 Example 6: Working with Jenkins Nodes

This is an example showing how to add, configure, enable and delete Jenkins nodes.

```
server.create_node('slave1')
nodes = get_nodes()
print nodes
node_config = server.get_node_info('slave1')
print node_config
server.disable_node('slave1')
server.enable_node('slave1')

# create node with parameters
```

(continues on next page)

(continued from previous page)

```
params = {
    'port': '22',
    'username': 'juser',
    'credentialsId': '10f3a3c8-be35-327e-b60b-a3e5edb0e45f',
    'host': 'my.jenkins.slave1'
}
server.create_node(
    'slave1',
    nodeDescription='my test slave',
    remoteFS='/home/juser',
    labels='precise',
    exclusive=True,
    launcher=jenkins.LAUNCHER_SSH,
    launcher_params=params)
```

2.2.7 Example 7: Working with Jenkins Build Queue

This is an example showing how to retrieve information on the Jenkins queue.

```
server.build_job('foo')
queue_info = server.get_queue_info()
id = queue_info[0].get('id')
server.cancel_queue(id)
```

2.2.8 Example 8: Working with Jenkins Cloudbees Folders

Requires the Cloudbees Folders Plugin for Jenkins.

This is an example showing how to create, configure and delete Jenkins folders.

```
server.create_job('folder', jenkins.EMPTY_FOLDER_XML)
server.create_job('folder/empty', jenkins.EMPTY_FOLDER_XML)
server.copy_job('folder/empty', 'folder/empty_copy')
server.delete_job('folder/empty_copy')
server.delete_job('folder')
```

2.2.9 Example 9: Updating Next Build Number

Requires the Next Build Number Plugin for Jenkins.

This is an example showing how to update the next build number for a Jenkins job.

```
next_bn = server.get_job_info('job_name')['nextBuildNumber']
server.set_next_build_number('job_name', next_bn + 50)
```

2.2.10 Example 9: Working with Build Promotions

Requires the Promoted Builds Plugin for Jenkins.

This is an example showing how to create, configure and delete a promotion process for an existing job.

The job in this example is named *prom_job* and it needs to have this config xml snippet before creating the promotion:

```
<properties>
   <hudson.plugins.promoted_builds.JobPropertyImpl>
   <activeProcessNames>
      <string>prom_name</string>
    </activeProcessNames>
  </hudson.plugins.promoted_builds.JobPropertyImpl>
</properties>
```

where *prom_name* is the name of the promotion that will get added to the job.

```
server.create_promotion('prom_name', 'prom_job', jenkins.EMPTY_PROMO_CONFIG_XML)
server.promotion_exists('prom_name', 'prom_job')
print server.get_promotions('prom_job')

server.reconfig_promotion('prom_name', 'prom_job', jenkins.PROMO_RECONFIG_XML)
print server.get_promotion_config('prom_name', 'prom_job')

server.delete_promotion('prom_name', 'prom_job')
```

2.2.11 Example 10: Waiting for Jenkins to be ready

It is possible to ask the API to wait for Jenkins to be ready with a given timeout. This can be used to aid launching of Jenkins and then waiting for the REST API to be responsive before continuing with subsequent configuration.

```
# timeout here is the socket connection timeout, for each connection
# attempt it will wait at most 5 seconds before assuming there is
# nothing listening. Useful where firewalls may black hole connections.
server = jenkins.Jenkins('http://localhost:8080', timeout=5)

# wait for at least 30 seconds for Jenkins to be ready
if server.wait_for_normal_op(30):
    # actions once running
    ...
else:
    print("Jenkins failed to be ready in sufficient time")
    exit 2
```

Note that the timeout arg to *jenkins.Jenkins()* is the socket connection timeout. If you set this to be more than the timeout value passed to *wait_for_normal_op()*, then in cases where the underlying connection is not rejected (firewall black-hole, or slow connection) then *wait_for_normal_op()* may wait at least the connection timeout, or a multiple of it where multiple connection attempts are made. A connection timeout of 5 seconds and a wait timeout of 8 will result in potentially waiting 10 seconds if both connections attempts do not get responses.

2.3 Installing

The module is known to pip and Debian-based distributions as `python-jenkins`.

`pip`:

```
pip install python-jenkins
```

`easy_install`:

```
easy_install python-jenkins
```

The module has been packaged since Ubuntu Oneiric (11.10):

```
apt-get install python-jenkins
```

And on Fedora 19 and later:

```
yum install python-jenkins
```

For development:

```
python setup.py develop
```

2.3.1 Documentation

Documentation is included in the `doc` folder. To generate docs locally execute the command:

```
tox -e docs
```

The generated documentation is then available under `doc/build/html/index.html`.

2.3.2 Unit Tests

Unit tests have been included and are in the `tests` folder. We recently started including unit tests as examples in our documentation so to keep the examples up to date it is very important that we include unit tests for every module. To run the unit tests, execute the command:

```
tox -e py27
```

- Note: View `tox.ini` to run tests on other versions of Python.

Due to how the tests are split up into a dedicated class per API method, it is possible to execute tests against a single API at a time. To execute the tests for the *`Jenkins.get_version()`* API execute the command:

```
tox -e py27 -- tests.test_version.JenkinsVersionTest
```

For further details on how to list tests available and different ways to execute them, see https://wiki.openstack.org/wiki/Testr.

2.3.3 Test Coverage

To measure test coverage, execute the command:

```
tox -e cover
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

j

Index