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>*, *resolve=True*)

Create handle to Jenkins instance.

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`
- **resolve** – Attempts to resolve and auto-correct API redirection. default: True `bool`

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*)

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`

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, `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, `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, `int`

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

`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*, *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.
- **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*)

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.

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

Note

On instances with many folders it may be more efficient to use the run_script method to retrieve all jobs instead.

Example:

```
server.run_script("""
    import groovy.json.JsonBuilder;

    // get all projects excluding matrix configuration
    // as they are simply part of a matrix project.
    // there may be better ways to get just jobs
    items = Jenkins.instance.getAllItems(AbstractProject);
    items.removeAll {
      it instanceof hudson.matrix.MatrixConfiguration
    };

    def json = new JsonBuilder()
    def root = json {
      jobs items.collect {
        [
          name: it.name,
          url: Jenkins.instance.getRootUrl() + it.getUrl(),
          color: it.getIconColor().toString(),
          fullname: it.getFullName()
        ]
      }
    }

    // use json.toPrettyString() if viewing
    println json.toString()
    """)
```

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.

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`

Note

On instances with many folders it may be more efficient to use the run_script method to retrieve the total number of jobs instead.

Example:

```
# get all projects excluding matrix configuration
# as they are simply part of a matrix project.
server.run_script(
    "print(Hudson.instance.getAllItems("
    "    hudson.model.AbstractProject).count{"
    "        !(it instanceof hudson.matrix.MatrixConfiguration)"
    "    })")
```

`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_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, `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)

`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.