azkaban Documentation

Release 0.9.7

Author

Apr 04, 2017

Contents

A lightweight Azkaban client providing:

- A command line interface to run jobs, upload projects, and more.

```
$ azkaban --project=my_project upload archive.zip
Project my_project successfully uploaded (id: 1, size: 205kB, version: 1).
Details at https://azkaban.server.url/manager?project=my_project

$ azkaban --project=my_project run my_flow
Flow my_flow successfully submitted (execution id: 48).
Details at https://azkaban.server.url/executor?execid=48
```

- A convenient and extensible way to build project configuration files.

```
from azkaban import PigJob, Project
from getpass import getuser

PROJECT = Project('sample', root=__file__)

# default options for all jobs
DEFAULTS = {
  'user.to.proxy': getuser(),
  'param': {
    'input_root': 'sample_dir/',
    'n_reducers': 20,
  },
  'jvm.args.mapred': {
    'max.split.size': 2684354560,
    'min.split.size': 2684354560,
  },
}

# list of pig job options
OPTIONS = [
  {'pig.script': 'first.pig'},
  {'pig.script': 'second.pig', 'dependencies': 'first.pig'},
  {'pig.script': 'third.pig', 'param': {'foo': 48}},
  {'pig.script': 'fourth.pig', 'dependencies': 'second.pig,third.pig'},
]

for option in OPTIONS:
  PROJECT.add_job(option['pig.script'], PigJob(DEFAULTS, option))
```

CHAPTER 1

Table of contents

Quickstart

Command line interface

Overview

Once installed, the `azkaban` executable provides several useful commands. These are divided into two kinds. The first will work out of the box with any existing Azkaban project:

- `azkaban run [options] WORKFLOW [JOB ...]`

  Launch a workflow (asynchronously). By default the entire workflow will be run, but you can specify specific jobs to only run those. This command will print the corresponding execution's URL to standard out.

- `azkaban upload [options] ZIP`

  Upload an existing project zip archive to the Azkaban server.

- `azkaban schedule [options] (-d DATE) (-t TIME) [-s SPAN]`

  Schedule a workflow to be run on a particular day and time. An optional span argument can also be specified to enable recurring runs.

- `azkaban log [options] EXECUTION [JOB]`

  View execution logs for a workflow or single job. If the execution is still running, the command will return on completion.

The second require a project configuration file (cf. *building projects*):

- `azkaban build [options]`

  Generate a project's job files and package them in a zip file along with any other project dependencies (e.g. jars, pig scripts). This archive can either be saved to disk or directly uploaded to Azkaban.

- `azkaban info [options]`

  View information about all the jobs inside a project, its static dependencies, or a specific job's options. In the former case, each job will be prefixed by `W` if it has no children (i.e. it "commands" a workflow), or `J` otherwise (regular job).

Running `azkaban --help` will show the full list of commands and options available for each.

URLs and aliases

The previous commands all take a `--url`, option used to specify where to find the Azkaban server (and which user to connect as).

```
$ azkaban build -u http://url.to.foo.server:port
```

In order to avoid having to input the entire URL every time, it is possible to defines aliases in `~/.azkabanrc`:

```
[azkaban]
default.alias = foo
[alias.foo]
url = http://url.to.foo.server:port
[alias.bar]
url = http://baruser@url.to.bar.server
# Optional keys (see corresponding `Session` argument for details):
verify = false
attempts = 5
```

We can now interact directly with each of these URLs using the `--alias` option followed by their corresponding alias. In particular, note that since we also specified a default alias, it is also possible to omit the option altogether. As a result, the commands below are now all equivalent:

```
$ azkaban build -u http://url.to.foo.server:port
$ azkaban build -a foo
$ azkaban build
```

Session IDs are conveniently cached after each successful login, so that we don't have to authenticate every time.

Building projects

We provide here a framework to define projects, jobs, and workflows from a single python file.

Motivation

For medium to large sized projects, it quickly becomes tricky to manage the multitude of files required for each workflow. `.properties` files are helpful but still do not provide the flexibility to generate jobs programmatically (i.e. using `for` loops, etc.). This approach also requires us to manually bundle and upload our project to the gateway every time.

Additionally, this will enable the `build` and `info` commands.

Quickstart

We start by creating a file. Let's call it `jobs.py` (the default file name the command line tool will look for), although any name would work. Below is a simple example of how we could define a project with a single job and static file:

```
from azkaban import Job, Project

project = Project('foo')
project.add_file('/path/to/bar.txt', 'bar.txt')
project.add_job('bar', Job({'type': 'command', 'command': 'cat bar.txt'}))
```

The *Project* class corresponds transparently to a project on the Azkaban server. The *add_file()* method then adds a file to the project archive (the second optional argument specifies the destination path inside the zip file). Similarly, the *add_job()* method will trigger the creation of a `.job` file. The first argument will be the file's name, the second is a *Job* instance (cf. *Job options*).

Once we've saved our jobs file, running the `azkaban` executable in the same directory will pick it up automatically and activate all commands. Note that we could also specify a custom configuration file location with the `-p --project` option (e.g. if the jobs file was in a different location).

Job options

The *Job* class is a light wrapper which allows the creation of `.job` files using python dictionaries.

It also provides a convenient way to handle options shared across multiple jobs: the constructor can take in multiple options dictionaries and the last definition of an option (i.e. later in the arguments) will take precedence over earlier ones.

We can use this to efficiently share default options among jobs, for example:

```
defaults = {'user.to.proxy': 'foo', 'retries': 0}

jobs = [
  Job({'type': 'noop'}),
  Job(defaults, {'type': 'noop'}),
  Job(defaults, {'type': 'command', 'command': 'ls'}),
  Job(defaults, {'type': 'command', 'command': 'ls -l', 'retries': 1}),
]
```

All jobs except the first one will have their `user.to.proxy` property set. Note also that the last job overrides the `retries` property.

Alternatively, if we really don't want to pass the defaults dictionary around, we can create a new `Job` subclass to do it for us:

```
class FooJob(Job):

  def __init__(self, *options):
    super(FooJob, self).__init__(defaults, *options)
```

Finally, since many Azkaban options are space/comma-separated strings (e.g. dependencies), the *Job* class provides two helpers to better handle their configuration: *join_option()* and *join_prefix()*.

More

Project properties

Any options added to a *Project*'s `properties` attribute will be available to all jobs inside of the project (under the hood, these get written to a global `.properties` file):

```
project.properties = {
  'user.to.proxy': 'foo',
  'my.custom.key': 'bar',
}
```

Note that this is particularly useful when combined with the *merge_into()* method to avoid job duplication when running projects with the same jobs but different options (e.g. a test and a production project).

Nested options

Nested dictionaries can be used to group options concisely:

```
# e.g. this job
Job({
  'proxy.user': 'boo',
  'proxy.keytab.location': '/path',
  'param.input': 'foo',
  'param.output': 'bar',
})
# is equivalent to this one
Job({
  'proxy': {'user': 'boo', 'keytab.location': '/path'},
  'param': {'input': 'foo', 'output': 'bar'},
})
```

Merging projects

If you have multiple projects, you can merge them together to create a single project. The merge is done in place on the project the method is called on. The first project will retain its original name.

```
from azkaban import Job, Project

project1 = Project('foo')
project1.add_file('/path/to/bar.txt', 'bar.txt')
project1.add_job('bar', Job({'type': 'command', 'command': 'cat bar.txt'}))

project2 = Project('qux')
project2.add_file('/path/to/baz.txt', 'baz.txt')
project2.add_job('baz', Job({'type': 'command', 'command': 'cat baz.txt'}))

# project1 will now contain baz.txt and the baz job from project2
project2.merge_into(project1)
```

Next steps

Any valid python code can go inside a jobs configuration file. This includes using loops to add jobs, subclassing the base `Job` class to better suit a project's needs (e.g. by implementing the `on_add` handler), etc.

API

azkaban.project

Project definition module.

class azkaban.project.**Project**(*name*, *root=None*, *register=True*, *version=None*)
Bases: object

Azkaban project.

Parameters

- **name** – Name of the project.
- **register** – Add project to registry. Setting this to False will make it invisible to the CLI.
- **root** – Path to a root file or directory used to enable adding files using relative paths (typically used with root=__file__).
- **version** – Project version, currently only used for setting the name of the archive uploaded to Azkaban.

The properties attribute of a project is a dictionary which can be used to pass Azkaban options which will then be available to all jobs in the project. This can be used for example to set project wide defaults.

To avoid undefined behavior, both the name and root attributes should not be altered after instantiation.

add_file(*path*, *archive_path=None*, *overwrite=False*)
Include a file in the project archive.

Parameters

- **path** – Path to file. If no project root exists, only absolute paths are allowed. Otherwise, this path can also be relative to said root.
- **archive_path** – Path to file in archive (defaults to same as path).
- **overwrite** – Allow overwriting any previously existing file in this archive path.

If the current project has its root parameter specified, this method will allow relative paths (and join those with the project's root), otherwise it will throw an error. Furthermore, when a project root exists, adding files above it without specifying an archive_path will raise an error. This is done to avoid having files in the archive with lower level destinations than the base root directory.

add_job(*name*, *job*, ***kwargs*)
Include a job in the project.

Parameters

- **name** – Name assigned to job (must be unique).
- **job** – *Job* instance.
- **kwargs** – Keyword arguments that will be forwarded to the *on_add()* handler.

This method triggers the *on_add()* method on the added job (passing the project and name as arguments, along with any kwargs). The handler will be called right after the job is added.

build(*path*, *overwrite=False*)
Create the project archive.

Parameters

- **path** – Destination path.
- **overwrite** – Don't throw an error if a file already exists at `path`.

files
Returns a list of tuples of files included in the project archive.

The first element of each tuple is the absolute local path to the file, the second the path of the file in the archive.

Note: This property should not be used to add files. Use *add_file()* instead.

---

jobs
Returns a dictionary of all jobs in the project, keyed by name.

Note: This property should not be used to add jobs. Use *add_job()* instead.

---

classmethod load(*path*, *new=False*)
Load Azkaban projects from script.

Parameters

- **path** – Path to python module.
- **new** – If set to `True`, only projects loaded as a consequence of calling this method will be returned.
- **propagate** – Propagate any exception raised while importing the module at `path`.

Returns a dictionary of *Project*'s keyed by project name. Only registered projects (i.e. instantiated with `register=True`) can be discovered via this method.

merge_into(*project*, *overwrite=False*, *unregister=False*)
Merge one project with another.

Parameters

- **project** – Target *Project* to merge into.
- **overwrite** – Overwrite any existing files.
- **unregister** – Unregister project after merging it.

The current project remains unchanged while the target project gains all the current project's jobs and files. Note that project properties are not carried over.

versioned_name
Project name, including version if present.

azkaban.job

Job definition module.

class azkaban.job.**Job**(**options*)
Bases: `object`

Base Azkaban job.

Parameters **options** – tuple of dictionaries. The final job options are built from this tuple by keeping the latest definition of each option. Furthermore, by default, any nested dictionary will be flattened (combining keys with `'.'`). Both these features can be changed by simply overriding the job constructor.

To enable more functionality, subclass and override the *on_add()* and *build()* methods. The *join_option()* and *join_prefix()* methods are also provided as helpers to write custom jobs.

build(*path=None*, *header=None*)
Write job file.

Parameters

- **path** – Path where job file will be created. Any existing file will be overwritten. Writes to stdout if no path is specified.
- **header** – Optional comment to be included at the top of the job file.

join_option(*option*, *sep*, *formatter='%s'*)
Helper method to join iterable options into a string.

Parameters

- **key** – Option key. If the option doesn't exist, this method does nothing.
- **sep** – Separator used to concatenate the string.
- **formatter** – Pattern used to format the option values.

Example usage:

```
class MyJob(Job):

  def __init__(self, *options):
    super(MyJob, self).__init__(*options)
    self.join_option('dependencies', ',')

# we can now use lists to define job dependencies
job = MyJob({'type': 'noop', 'dependencies': ['bar', 'foo']})
```

join_prefix(*prefix*, *sep*, *formatter*)
Helper method to join options starting with a prefix into a string.

Parameters

- **prefix** – Option prefix.
- **sep** – Separator used to concatenate the string.
- **formatter** – String formatter. It is formatted using the tuple `(suffix, value)` where `suffix` is the part of `key` after `prefix`.

Example usage:

```
class MyJob(Job):

  def __init__(self, *options):
    super(MyJob, self).__init__(*options)
    self.join_prefix('jvm.args', ' ', '-D%s=%s')

# we can now define JVM args using nested dictionaries
job = MyJob({'type': 'java', 'jvm.args': {'foo': 48, 'bar': 23}})
```

on_add(*project*, *name*, ***kwargs*)
Handler called when the job is added to a project.

Parameters

- **project** – *Project* instance
- **name** – name corresponding to this job in the project.
- **kwargs** – Keyword arguments. If this method is triggered by *add_job()*, the latter's keyword arguments will simply be forwarded. Else if this method is triggered by a merge, kwargs will be a dictionary with single key `'merging'` and value the merged project.

The default implementation does nothing.

azkaban.remote

Azkaban remote interaction module.

This contains the `Session` class which will be used for all interactions with a remote Azkaban server.

class `azkaban.remote.`**Execution**(*session*, *exec_id*)
Bases: `object`

Remote workflow execution.

Parameters

- **session** – *Session* instance.
- **exec_id** – Execution ID.

cancel()
Cancel execution.

job_logs(*job*, *delay=5*)
Job log generator.

Parameters

- **job** – job name
- **delay** – time in seconds between each server poll

Yields line by line.

logs(*delay=5*)
Execution log generator.

Parameters **delay** – time in seconds between each server poll

Yields line by line.

classmethod start(*session*, *args*, ***kwargs*)
Convenience method to start a new execution.

Parameters

- **session** – *Session* instance.
- **args** – Cf. *Session.run_workflow()*.
- **kwargs** – Cf. *Session.run_workflow()*.

status
Execution status.

url
Execution URL.

class azkaban.remote.**Session**(*url=None*, *alias=None*, *config=None*, *attempts=3*, *verify=True*)
Bases: object

Azkaban session.

Parameters

- **url** – HTTP endpoint (including protocol, port and optional user).
- **alias** – Alias name.
- **config** – Configuration object used to store session IDs.
- **attempts** – Maximum number of attempts to refresh session.
- **verify** – Whether or not to verify HTTPS requests.

This class contains mostly low-level methods that translate directly into Azkaban API calls. The *Execution* class should be preferred for interacting with workflow executions.

Note that each session's ID is lazily updated. In particular, instantiating the *Session* doesn't guarantee that its current ID (e.g. loaded from the configuration file) is valid.

cancel_execution(*exec_id*)
Cancel workflow execution.

Parameters **exec_id** – Execution ID.

create_project(*name*, *description*)
Create project.

Parameters

- **name** – Project name.
- **description** – Project description.

delete_project(*name*)
Delete a project on Azkaban.

Parameters **name** – Project name.

classmethod from_alias(*alias*, *config=None*)
Create configured session from an alias.

Parameters

- **alias** – Alias name.
- **config** – Azkaban configuration object.

get_execution_logs(*exec_id*, *offset=0*, *limit=50000*)
Get execution logs.

Parameters

- **exec_id** – Execution ID.
- **offset** – Log offset.
- **limit** – Size of log to download.

get_execution_status(*exec_id*)
Get status of an execution.

Parameters **exec_id** – Execution ID.

get_job_logs(*exec_id*, *job*, *offset=0*, *limit=50000*)
Get logs from a job execution.

Parameters

- **exec_id** – Execution ID.
- **job** – Job name.
- **offset** – Log offset.
- **limit** – Size of log to download.

get_projects()
Get a list of all projects.

get_running_workflows(*project*, *flow*)
Get running executions of a flow.

Parameters

- **project** – Project name.
- **flow** – Flow name.

Note that if the project doesn't exist, the Azkaban server will return a somewhat cryptic error `Project 'null' not found.`, even though the name of the project isn't `null`.

get_schedule(*name*, *flow*)
Get schedule information.

Parameters

- **name** – Project name.
- **flow** – Name of flow in project.

get_sla(*schedule_id*)
Get SLA information.

Parameters **schedule_id** – Schedule Id - obtainable from get_schedule

get_workflow_executions(*project*, *flow*, *start=0*, *length=10*)
Fetch executions of a flow.

Parameters

- **project** – Project name.
- **flow** – Flow name.
- **start** – Start index (inclusive) of the returned list.
- **length** – Max length of the returned list.

get_workflow_info(*name*, *flow*)
Get list of jobs corresponding to a workflow.

Parameters

- **name** – Project name.
- **flow** – Name of flow in project.

get_workflows(*name*)
Get list of workflows corresponding to a project

Parameters **name** – Project name

is_valid(*response=None*)
Check if the current session ID is valid.

Parameters **response** – If passed, this reponse will be used to determine the validity of the session. Otherwise a simple test request will be emitted.

run_workflow(*name*, *flow*, *jobs=None*, *disabled_jobs=None*, *concurrent=True*, *properties=None*, *on_failure='finish'*, *notify_early=False*, *emails=None*)
Launch a workflow.

Parameters

- **name** – Name of the project.
- **flow** – Name of the workflow.
- **jobs** – List of names of jobs to run (run entire workflow by default). Mutually exclusive with `disabled_jobs` parameter.
- **disabled_jobs** – List of names of jobs not to run. Mutually exclusive with `jobs` parameter.
- **concurrent** – Run workflow concurrently with any previous executions. Can either be a boolean or a valid concurrency option string. Available string options: `'skip'` (do not run flow if it is already running), `'concurrent'` (run the flow in parallel with any current execution), `'pipeline:1'` (pipeline the flow such that the current execution will not be overrun: block job A until the previous flow job A has completed), `'pipeline:2'` (pipeline the flow such that the current execution will not be overrun: block job A until the previous flow job A's _children_ have completed).
- **properties** – Dictionary that will override global properties in this execution of the workflow. This dictionary will be flattened similarly to how *Job* options are handled.
- **on_failure** – Set the execution behavior on job failure. Available options: `'finish'` (finish currently running jobs, but do not start any others), `'continue'` (continue executing jobs as long as dependencies are met),‘’cancel’‘ (cancel all jobs immediately).
- **notify_early** – Send any notification emails when the first job fails rather than when the entire workflow finishes.
- **emails** – List of emails or pair of list of emails to be notified when the flow fails. Note that this will override any properties set in the worfklow. If a single list is passed, the emails will be used for both success and failure events. If a pair of lists is passed, the first will receive failure emails, the second success emails.

Note that in order to run a workflow on Azkaban, it must already have been uploaded and the corresponding user must have permissions to run it.

schedule_cron_workflow(*name*, *flow*, *cron_expression*, ***kwargs*)
Schedule a cron workflow.

Parameters

- **name** – Project name.
- **flow** – Name of flow in project.
- **cron_expression** – A CRON expression comprising 6 or 7 fields separated by white space that represents a set of times in Quartz Cron Format.
- ****kwargs** – See *run_workflow()* for documentation.

schedule_workflow(*name*, *flow*, *date*, *time*, *period=None*, ***kwargs*)
Schedule a workflow.

Parameters

- **name** – Project name.
- **flow** – Name of flow in project.
- **date** – Date of the first run (possible values: `'08/07/2014'`, `'12/11/2015'`).
- **time** – Time of the schedule (possible values: `'9,21,PM,PDT'`, `'10,30,AM,PDT'`).
- **period** – Frequency to repeat. Consists of a number and a unit (possible values: `'1s'`, `'2m'`, `'3h'`, `'2M'`). If not specified the flow will be run only once.
- ****kwargs** – See *run_workflow()* for documentation.

set_sla(*schedule_id*, *email*, *settings*)
Set SLA for a schedule.

Parameters

- **schedule_id** – Schedule ID.
- **email** – Array of emails to receive notifications.
- **settings** – Array of comma delimited strings of SLA settings consisting of:
  - job name - blank for full workflow
  - rule - SUCCESS or FINISH
  - duration - specified in hh:mm
  - email action - bool
  - kill action - bool

unschedule_workflow(*name*, *flow*)
Unschedule a workflow.

Parameters

- **name** – Project name.
- **flow** – Name of flow in project.

upload_project(*name*, *path*, *archive_name=None*, *callback=None*)
Upload project archive.

Parameters

- **name** – Project name.
- **path** – Local path to zip archive.
- **archive_name** – Filename used for the archive uploaded to Azkaban. Defaults to `basename(path)`.
- **callback** – Callback forwarded to the streaming upload.

azkaban.util

Utility module.

class azkaban.util.**Adapter**(*prefix*, *logger*, *extra=None*)
Bases: logging.LoggerAdapter

Logger adapter that includes a prefix to all messages.

Parameters

- **prefix** – Prefix string.
- **logger** – Logger instance where messages will be logged.
- **extra** – Dictionary of contextual information, passed to the formatter.

process(*msg*, *kwargs*)
Adds a prefix to each message.

Parameters

- **msg** – Original message.
- **kwargs** – Keyword arguments that will be forwarded to the formatter.

exception azkaban.util.**AzkabanError**(*message*, **args*)
Bases: exceptions.Exception

Base error class.

class azkaban.util.**Config**(*path=None*)
Bases: object

Configuration class.

Parameters **path** – path to configuration file. If no file exists at that location, the configuration parser will be empty. Defaults to ~/.azkabanrc.

get_file_handler(*command*)
Add and configure file handler.

Parameters **command** – Command the options should be looked up for.

The default path can be configured via the default.log option in the command's corresponding section.

get_option(*command*, *name*, *default=None*)
Get option value for a command.

Parameters

- **command** – Command the option should be looked up for.
- **name** – Name of the option.
- **default** – Default value to be returned if not found in the configuration file. If not provided, will raise *AzkabanError*.

save()
Save configuration parser back to file.

class azkaban.util.**MultipartForm**(*files*, *params=None*, *callback=None*, *chunksize=4096*)
Bases: object

Form allowing streaming.

Parameters

- **files** – List of filepaths. For more control, each file can also be represented as a dictionary with keys 'path', 'name', and 'type'.

- **params** – Optional dictionary of parameters that will be included in the form.
- **callback** – Arguments `cur_bytes`, `tot_bytes`, `index`.
- **chunksize** – Size of each streamed file chunk.

Usage:

```
from requests import post

form = MultipartForm(files=['README.rst'])
post('http://your.url', headers=form.headers, data=form)
```

size
Total size of all the files to be streamed.

Note that this doesn't include the bytes used for the header and parameters.

azkaban.util.**catch**(**error_classes*)
Returns a decorator that catches errors and prints messages to stderr.

Parameters

- **error_classes** – Error classes.
- **log** – Filepath to log file.

Also exits with status 1 if any errors are caught.

azkaban.util.**flatten**(*dct*, *sep='.'*)
Flatten a nested dictionary.

Parameters

- **dct** – Dictionary to flatten.
- **sep** – Separator used when concatenating keys.

azkaban.util.**human_readable**(*size*)
Transform size from bytes to human readable format (kB, MB, ...).

Parameters **size** – Size in bytes.

azkaban.util.**read_properties**(**paths*)
Read options from a properties file and return them as a dictionary.

Parameters ***paths** – Paths to properties file. In the case of multiple definitions of the same option, the latest takes precedence.

Note that not all features of `.properties` files are guaranteed to be supported.

azkaban.util.**stream_file**(*path*, *chunksize*)
Get iterator over a file's contents.

Parameters

- **path** – Path to file.
- **chunksize** – Bytes per chunk.

azkaban.util.**suppress_urllib_warnings**()
Capture urllib warnings if possible, else disable them (python 2.6).

azkaban.util.**temppath**(**args*, ***kwds*)
Create a temporary filepath.

Usage:

```
with temppath() as path:
  # do stuff
```

Any file corresponding to the path will be automatically deleted afterwards.

azkaban.util.**write_properties**(*options*, *path=None*, *header=None*)
Write options to properties file.

Parameters

- **options** – Dictionary of options.
- **path** – Path to file. Any existing file will be overwritten. Writes to stdout if no path is specified.
- **header** – Optional comment to be included at the top of the file.

Extensions

Pig

Since pig jobs are so common, azkaban comes with an extension to:

- run pig scripts directly from the command line (and view the output logs from your terminal): azkabanpig. Under the hood, this will package your script along with the appropriately generated job file and upload it to Azkaban. Running azkabanpig --help displays the list of available options (using UDFs, substituting parameters, running several scripts in order, etc.).
- integrate pig jobs easily into your project configuration via the *PigJob* class which automatically sets the job type and adds the corresponding script file to the project.

```
from azkaban import PigJob

project.add_job('baz', PigJob({'pig.script': 'baz.pig'})
```

The full API for the *PigJob* class is below.

class azkaban.ext.pig.**PigJob**(**options*)
Bases: *azkaban.job.Job*

Convenience job class for running pig scripts.

Parameters **options** – Tuple of options (cf. *Job*). These options must specify a 'pig.script' key. The corresponding file will then automatically be included in the project archive.

This class allows you to specify JVM args as a dictionary by correctly converting these to the format used by Azkaban when building the job options. For example: {'jvm.args': {'foo': 1, 'bar': 2}} will be converted to jvm.args=-Dfoo=1 -Dbar=2. Note that this enables JVM args to behave like all other Job options when defined multiple times (latest values taking precedence).

Finally, by default the job type will be set automatically to 'pig'. You can also specify a custom job type for all PigJob instances in the azkabanpig section of the ~/.azkabanrc configuration file via the default.type option.

on_add(*project*, *name*, ***kwargs*)
This handler adds the corresponding script file to the project.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index