JobControl Documentation

Release 0.1a

Samuele Santi

December 12, 2014

Contents

Application usage:

CHAPTER 1

Configuration

The main configuration file is written in YAML and pre-processed through Jinja, to allow things like defining variables, macros, etc.

1.1 Storage

Define an URL pointing to the storage (for build status).

```
storage: "postgresql://jobcontrol_test:test@localhost:5432/jc-harvester-141125"
```

1.2 Webapp

Configuration for the web application.

Uppercase names will be merged with standard Flask configuration.

```
webapp:
    PORT: 5050
    DEBUG: False
```

1.3 Celery

Configuration for Celery (the asynchronous task running library).

See all the possible configuration options here: http://docs.celeryproject.org/en/latest/configuration.html

```
celery:
    BROKER_URL: "redis://localhost:6379"
```

1.4 Jobs

Job definition is a list of objects like this:

```
id: some_job_id
title: "Some job title here"
function: mypackage.mymodule:myfunction
args:
    - spam
    - eggs
    - bacon
kwargs:
    foobar: 'Something completely different'
    blah: !retval 'some_other_job'
dependencies: ['some_other_job']
```

..which tells JobControl to run something roughly equivalent to:

```
from mypackage.mymodule import myfunction

myfunction('spam', 'eggs', 'bacon',
           foobar='Something completely different',
           blah=get_return_value('some_other_job'))
```

Where the (immaginary) `get_return_value()` function returns the return value from the latest successful build of the specified job (which *must* be amongst the job dependencies).

1.5 Planned job configuration keys

- `protect` boolean indicating whether this job must be "protected": by "protect" we mean "from accidental mistakes"; for example, it would be handy to prevent accidental builds of jobs that import things in production websites. If this flag is set, the "quick build" feature will be disabled and the build form submit button will need "arming" (by clicking another button) before being actually usable.
- `cleanup` indicate a function to be called on build deletion to clean up any data stored externally. That function requires access to the build status, eg. in order to get a pointer to the storage containing the data.

1.6 Example macros

For example, let's say we want to "crawl" and "process" a bunch of websites.

We could use a macro like this to keep repetitions at minimum:

```
{% macro process_website(name, url) %}
  - id: crawl_{{ name }}
    title: "Crawl {{ url }}"
    function: mycrawler:crawl
    kwargs:
      storage: postgresql://.../crawled_data_{{ name }}

  - id: process_{{ name }}
    title: "Process {{ url }}"
    function: mycrawler:process
    kwargs:
      input_storage: !retval crawl_{{ name }}
      storage: postgresql://.../processed_data_{{ name }}
{% endmacro %}

jobs:
```

```
{{ process_website('example_com', 'http://www.example.com') }}
{{ process_website('example_org', 'http://www.example.org') }}
{{ process_website('example_net', 'http://www.example.net') }}
```

Will get expanded to:

```
jobs:
  - id: crawl_example_com
    title: "Crawl http://www.example.com"
    function: mycrawler:crawl
    kwargs:
      storage: postgresql://.../crawled_data_example_com

  - id: process_example_com
    title: "Process http://www.example.com"
    function: mycrawler:process
    kwargs:
      input_storage: !retval crawl_example_com
      storage: postgresql://.../processed_data_example_com

  - id: crawl_example_org
    title: "Crawl http://www.example.org"
    function: mycrawler:crawl
    kwargs:
      storage: postgresql://.../crawled_data_example_org

  - id: process_example_org
    title: "Process http://www.example.org"
    function: mycrawler:process
    kwargs:
      input_storage: !retval crawl_example_org
      storage: postgresql://.../processed_data_example_org

  - id: crawl_example_net
    title: "Crawl http://www.example.net"
    function: mycrawler:crawl
    kwargs:
      storage: postgresql://.../crawled_data_example_net

  - id: process_example_net
    title: "Process http://www.example.net"
    function: mycrawler:process
    kwargs:
      input_storage: !retval crawl_example_net
      storage: postgresql://.../processed_data_example_net
```

Warning: Mind the indentation! The best way is to use the desired final indentation in the macro definition, then call the macro at "zero" indentation level.

CHAPTER 2

Command-line interface

All the operations can be run through the "jobcontrol-cli" command.

It is self-documented: running `jobcontrol-cli --help` will give information on available commands; `jobcontrol-cli <command> --help` will give usage information on a specific command.

2.1 Installing database schema

```
jobcontrol-cli --config-file myconfig.yaml install
```

2.2 Uninstalling database schema

Warning: This will drop all tables without any further warning!

```
jobcontrol-cli --config-file myconfig.yaml uninstall
```

2.3 Running the web app

Note: For production mode, the application should be run via a proper WSGI container, such as gunicorn or uWSGI.

```
jobcontrol-cli --config-file myconfig.yaml web --port 5050 --debug
```

CHAPTER 3

Deployment instructions

Requisites:

- **Python** 2.7 (2.6 should work but it's untested)
- **PostgreSQL** 9.1+ (tested on 9.4 but older 9.x versions should do)
- **Redis** (any recent version should do; tested on 2.8.17)

Steps:

- Create a PostgreSQL database for jobcontrol
- Install jobcontrol in a virtualenv:

```
virtualenv jobcontrol
pip install jobcontrol
```

- *Write a configuration file*
- Create database tables:

```
jobcontrol-cli --config-file path/to/conf.yaml install
```

- Launch the webapp:

```
jobcontrol-cli --config-file path/to/conf.yaml web --port 5050
```

- Start redis server:

```
redis-server
```

- Launch the celery worker:

```
jobcontrol-cli --config-file path/to/conf.yaml worker
```

- Visit http://127.0.0.1:5050
- Enjoy!

3.1 todo

- Give some better details for production deployment, eg.

Internals documentation:

CHAPTER 4

jobcontrol

4.1 jobcontrol.core

Objects responsible for JobControl core functionality.

Note: Important objects from this module should be imported in main __init___, in order to "abstract away" the namespace and have them in a more nicely accessible place.

class jobcontrol.core.**JobControl**(*storage*, *config*)
The main JobControl class

classmethod from_config_file(*config_file*)
Initialize JobControl by loading configuration from a file. Will also initialize storage taking values from the configuration.

Parameters **config_file** – Path to configuration file or open file descriptor

classmethod from_config(*config*)
Initialize JobControl from some configuration.

Parameters **config** – Either a jobcontrol.job_conf.JobControlConfigMgr instance, or a dict to be passed as argument to constructor.

Returns a JobControl instance

get_job(*job_id*)
Get a job, by id.

Parameters **job_id** – The job id

Returns a JobInfo class instance associated with the requested job.

iter_jobs()
Generator yielding all the jobs, one by one.

Yields for each job, a JobInfo class instance associated with the job.

get_build(*build_id*)
Get a build, by id.

Returns a BuildInfo instance.

create_build(*job_id*)
Create a build from a job configuration.

Currently, we require that all the dependencies have already been built; in the future, it will be possible to build them automatically.

Also, current implementation doesn't allow for customizations to either the job configuration nor the build one (pinning, dep/revdep building, ...).

Parameters **job_id** – Id of the job for which to start a build

Returns a `BuildInfo` instance.

build_job(*job_id*)
Create and run a new build for the specified job

run_build(*build_id*)
Actually run a build.

•take the build configuration

•make sure all the dependencies are built

•take return values from the dependencies -> pass as arguments

•run the build

•build the reverse dependencies as well, if required to do so

Parameters **build_id** – either a `BuildInfo` instance, or a build id

prune_logs(*policy=None*)

report_progress(*group_name*, *current*, *total*, *status_line=''*)
Report progress for the currently running build.

Parameters

- **group_name** – The report "group name": either a tuple representing the "path", or None for the top-level.
- **current** – Current progress
- **total** – Total progress
- **status_line** – An optional line of text, describing the currently running operation.

get_celery_app()
Return the Celery application, configured with values from the current configuration.

Note: this is a bit hackish, as we are just *updating* configuration values in the global object with ones from the jobcontrol configuration, not replacing all the configuration at once.

---

class jobcontrol.core.**JobExecutionContext**(*app*, *job_id*, *build_id*)
Class to hold "global" context during job execution.

This class can also act as a context manager for temporary context:

```
with JobExecutionContext(app, job_id, build_id):
    pass # do stuff in an execution context
```

Parameters

- **app** – The JobControl instance running jobs
- **job_id** – Id of the currently running job

- **build_id** – Id of the currently running build

push()
Push this context in the global stack

pop()
Pop this context from the global stack

current_app
Returns the currently running app

current_job
Returns a `JobInfo` instance associated with the currently running job.

current_build
Returns a `BuildInfo` instance associated with the currently running build.

class jobcontrol.core.**JobControlLogHandler**
Logging handler sending messages to the appropriate JobControl instance that will dispatch them to storage.

flush()
No-op, as we don't need to flush anything

emit(*record*)
"Emit" the log record (if there is an execution context, store the log record appropriately; otherwise, just ignore it).

class jobcontrol.core.**JobInfo**(*app*, *job_id*, *config*)
High-level interface to jobs

id

config

get_deps()
Iterate over jobs this job depends upon.

Yields `JobInfo` instances

get_status()
Return a label describing the current status of the job.

Returns

- `'not_built'` the job has no builds
- `'running'` the job has running builds
- `'success'` the job has at least a successful build
- `'failed'` the job only has failed builds
- `'outdated'` the job has at least a successful build, but older than one dependency build

get_revdeps()
Iterate over jobs depending on this one

Yields `JobInfo` instances

iter_builds(**a*, ***kw*)
Iterate over builds for this job.

Accepts the same arguments as `jobcontrol.interfaces.StorageBase.get_job_builds()`

Yields `BuildInfo` instances

get_builds(**a*, ***kw*)
DEPRECATED alias for iter_builds()

run()
Trigger run for this job (will automatically create a build, etc.)

create_build()

get_latest_successful_build()
Get latest successful build for this job, if any. Otherwise, returns `None`.

get_docs()
Get documentation for this job.

get_conf_as_yaml()
Return the job configuration as serialized YAML, mostly for displaying on user interfaces.

has_builds()
Check whether this job has any build.

has_successful_builds()
Check whether this job has any successful build.

has_running_builds()
Check whether this job has any running build.

is_outdated()
Check whether any dependency has builds more recent than the newest build for this job.

can_be_built()
Checks whether a job can be built, i.e.: whether all the dependencies have at least one successful build.

class jobcontrol.core.**BuildInfo**(*app*, *build_id*, *info=None*)
High-level interface to builds.

Parameters

- **app** – The JobControl instance this build was retrieved from
- **build_id** – The build id
- **info** – Optionally, this can be used to pre-populate the build information (useful, eg. if we are retrieving a bunch of builds from the database at once).

id
The build id

job_id
The job id

info
Property used to lazily access the build attributes.

Returns a dict with the following keys:

- 'id'
- 'job_id'
- 'start_time'
- 'end_time'
- 'started'
- 'finished'

- 'success'
- 'skipped'
- 'job_config'
- 'build_config'
- 'retval'
- 'exception'
- 'exception_tb'

job_config
Property to access the job configuration.

build_config
Property to access the build configuration.

descriptive_status
Return a label describing the current status of the build.

Returns

- 'CREATED' if the build was not started yet
- 'RUNNING' if the build was started but did not finish
- 'SUCCESSFUL' if the build run with success
- 'SKIPPED' if the build was skipped
- 'FAILED' if the build execution failed

refresh()
Refresh the build status information from database

get_progress_info()
Get information about the build progress

get_job()
Get a JobInfo associated with this build's job

delete()
Delete all information related to this build from database

run()
Calls run_build() on the main app for this build

iter_log_messages(**kw)
Iterate over log messages for this build.

Keywords are passed directly to the underlying iter_log_messages() method of the storage.

4.2 jobcontrol.exceptions

This module contains the exceptions used by JobControl.

exception jobcontrol.exceptions.**JobControlException**
Base for JobControl exceptions

exception jobcontrol.exceptions.**NotFound**
Exception used to indicate something was not found. Pretty generic, but useful for returning 404s..

exception jobcontrol.exceptions.**MissingDependencies**
Exception used to indicate a build dependency was not met (i.e. job has no successful builds).

exception jobcontrol.exceptions.**SkipBuild**
Exception raised by builds to indicate the current build should be skipped, eg. because there is no need for a rebuild.

exception jobcontrol.exceptions.**SerializationError**
Exception raised when serialization of a build's return value failed.

4.3 jobcontrol.globals

4.4 jobcontrol.interfaces

Interfaces for NEW jobcontrol objects.

Data model:

```
Build   id SERIAL
-----   job_id TEXT
        start_time TIMESTAMP
        end_time TIMESTAMP
        started BOOLEAN
        finished BOOLEAN
        success BOOLEAN
        skipped BOOLEAN
        job_config TEXT (YAML)
            Copy of the job configuration whan the build was started
        build_config TEXT (YAML)
            Extra configuration, such as dependency build "pinning"
        retval BINARY (Pickled return value)
        exception BINARY
            Pickled exception object (or None)
        exception_tb BINARY
            Pickled TracebackInfo object

Build progress
--------------
        build_id INTEGER (references Build.id)
        group_name VARCHAR(128)
            Name of the "progress group" (separated by '::')
        current INTEGER
            Current progress value
        total INTEGER
            Total progress value
        status_line TEXT
            An optional line of text describing current state
        UNIQUE constraint on (build_id, group_name)

Log     id SERIAL
---     build_id INTEGER (references Build.id)
        created TIMESTAMP
        level INTEGER
        record BINARY
            Pickled LogRecord
```

```
exception_tb BINARY
    Pickled TracebackInfo object
```

Job configuration:

The job configuration is stored as a YAML-serialized dict.

Recognised keys are:

- `function` in `module:function` format, specify the function to be called
- `args` a list of arguments to be passed to the function
- `kwargs` a dict of keyword arguments to be passed to the function
- `title` a descriptive title, to be shown on the interfaces
- `notes` notes, to be shown in interfaces (in restructured text)
- `dependencies` list of dependency job names

Additionally, args/kwargs may contain references to return value of dependency builds, by using the `!retval <name>` syntax.

Exception traceback serialization

To be used both in build records and associated with log messages containing an exception.

We want to include the following information:

- Details about the call stack, as in normal tracebacks: filename, line number, function name, line of code (plus some context)
- Local variables: we are not guaranteed we can safely pickle / unpickle arbitrary values; moreover this might result in huge fields, etc. So our better chance is to just store a dictionary mapping names to repr()s of the values (trimmed to a – large – maximum length, just to be on the safe side).

class jobcontrol.interfaces.**StorageBase**

classmethod from_url(*url*)

install()

uninstall()

get_job_builds(*job_id*, *started=None*, *finished=None*, *success=None*, *skipped=None*, *order='asc'*, *limit=100*)

Iterate over all the builds for a job, sorted by date, according to the order specified by `order`.

Parameters

- **job_id** – The job id
- **started** – If set to a boolean, filter on the "started" field
- **finished** – If set to a boolean, filter on the "finished" field
- **success** – If set to a boolean, filter on the "success" field
- **skipped** – If set to a boolean, filter on the "skipped" field
- **order** – 'asc' (default) or 'desc'
- **limit** – only return the first `limit` builds

Yield Dictionaries representing build information

create_build(*job_id*, *job_config*, *build_config*)
Create a build.

Parameters

- **job_id** – The job for which a build should be started
- **job_config** – The job configuration `(function, args, kwargs, ..)` to be copied inside the object (we will use this from now on).
- **build_config** – Build configuration, containing things like dependency build pinning, etc.
  - `dependency_builds`: dict mapping job ids to build ids, or `None` to indicate "create a new build" for this job.

Returns the build id

get_build(*build_id*)
Get information about a build.

Returns the build information, as a dict

delete_build(*build_id*)
Delete a build, by id.

start_build(*build_id*)
Register a build execution start.

finish_build(*build_id*, *success=None*, *skipped=None*, *retval=None*, *exception=None*, *exception_tb=None*)
Register a build execution end.

finish_build_with_exception(*build_id*)

update_build_progress(*build_id*, *current*, *total*)

report_build_progress(*build_id*, *current*, *total*, *group_name=''*, *status_line=''*)
Report progress for a build.

Parameters

- **build_id** – The build id for which to report progress
- **current** – The current number of "steps" done
- **total** – The total amount of "steps"
- **group_name** – Optionally, a name used to nest multiple progress "levels". A tuple (or string separated by '::' can be used to specify multiple "nesting" levels)
- **status_line** – Optionally, a line of text indicating the current build status.

get_build_progress_info(*build_id*)
Return progress information for a build.

Returns a list of tuples: `(name, current, total, status_line)`

get_latest_successful_build(*job_id*)
Helper method to retrieve the latest successful build for a given job. Calls `get_job_builds()` in the background.

Returns information about the build, as a dict

log_message(*build_id*, *record*)
Store a log record associated with a build.

prune_log_messages(*job_id=None*, *build_id=None*, *max_age=None*, *level=None*)
Delete (old) log messages.

Parameters

- **job_id** – If specified, only delete messages for this job
- **build_id** – If specified, only delete messages for this build
- **max_age** – If specified, only delete log messages with an age greater than this one (in seconds)
- **level** – If specified, only delete log messages with a level equal or minor to this one

iter_log_messages(*build_id=None*, *max_date=None*, *min_date=None*, *min_level=None*)
Iterate over log messages, applying some filters.

Parameters

- **build_id** – If specified, only return messages for this build
- **max_date** – If specified, only return messages newer than this date
- **min_date** – If specified, only return messages older than this date
- **min_level** – If specified, only return messages with a level at least equal to this one

pack(*obj*, *safe=False*)

pack_log_record(*record*)
Pack a log record.

This special-cased function is meant to gracefully handle cases of log messages not being serializable, usually due to some "attr" or the attached exception not being serializable.

pack_exception(*exception*)

unpack(*obj*, *safe=False*)

yaml_pack(*obj*)

yaml_unpack(*obj*)

4.5 jobcontrol.job_conf

Functions to manage the job configuration

The job configuration is a YAML object (dict) containing (at least) the following keys:

- module - name of the module from wich to import the function
- function - name of the function to be called
- args - arguments to the function (list)
- kwargs - keyword arguments to the function (dictionary)
- dependencies - dependencies for this job

Additional "constructors" are available:

- `!retval <n>` will be replaced with return value of latest successful build for dependency job `<n>` (and job `<n>` *must* be specified as a dependency)
- [proposed] `!cfg <name>` will be replaced with global configuration option `<name>`

- [proposed] "system" objects, such as context, job configuration, ... might be passed/accessed as well?
  - execution context
  - current job object
  - current build object
- [proposed] `!secret <name>` value from "secret" configuration, usually used for storing passwords etc, on file.

Note: job configuration widgets *need* to manipulate the configuration, if we want to expose it in a nicer way – is there any way to do so while preserving formatting / comments in other parts of the document?

Job configuration:

```
jobs:

  - name: my-job-name
    title: A descriptive title
    function: package.module:name
    args: []
    kwargs:
      storage: {url: 'mongodb://...'}
      input_storage: !retval 'other-job-name'
    dependencies: ['other-job-name']

  - name: other-job-name
    title: Another descriptive title
    function: package.module:othername
```

class jobcontrol.job_conf.**Retval**(*job_id*)
Placeholder for `!retval <n>`

jobcontrol.job_conf.**dump**(*data*)

jobcontrol.job_conf.**load**(*stream*)

jobcontrol.job_conf.**prepare_args**(*args*, *build*)
Prepare arguments / kwargs by replacing placeholders with actual values from the context.

class jobcontrol.job_conf.**JobControlConfigMgr**(*initial=None*)

classmethod from_file(*filename*)

classmethod from_string(*s*)

classmethod from_object(*data*)

config

validate(*data*)

iter_jobs()

get_job(*job_id*)

get_job_deps(*job_id*)

get_job_revdeps(*job_id*)

get_webapp_config()
Returns a dict containing configuration for the Flask app

get_secret(*name*)

get_storage()

4.6 jobcontrol.utils

4.6.1 jobcontrol.utils.depgraph

Dependency graph exploration / resolution functions.

The dependency graph is represented as a dictionary of `{<vertex>: [<dependencies>]}`.

exception jobcontrol.utils.depgraph.**DepResolutionError**

exception jobcontrol.utils.depgraph.**DepLoop**

jobcontrol.utils.depgraph.**resolve_deps**(*graph*, *start*, *with_weights=False*)

4.6.2 jobcontrol.utils.local

werkzeug.local

This module implements context-local objects.

copyright

3. 2014 by the Werkzeug Team, see AUTHORS for more details.

license BSD, see LICENSE for more details.

jobcontrol.utils.local.**implements_bool**(*cls*)

jobcontrol.utils.local.**release_local**(*local*)

Releases the contents of the local for the current context. This makes it possible to use locals without a manager.

Example:

```
>>> loc = Local()
>>> loc.foo = 42
>>> release_local(loc)
>>> hasattr(loc, 'foo')
False
```

With this function one can release `Local` objects as well as `LocalStack` objects. However it is not possible to release data held by proxies that way, one always has to retain a reference to the underlying local object in order to be able to release it.

New in version 0.6.1.

class jobcontrol.utils.local.**Local**

class jobcontrol.utils.local.**LocalStack**

This class works similar to a `Local` but keeps a stack of objects instead. This is best explained with an example:

```
>>> ls = LocalStack()
>>> ls.push(42)
>>> ls.top
42
>>> ls.push(23)
>>> ls.top
```

```
23
>>> ls.pop()
23
>>> ls.top
42
```

They can be force released by using a `LocalManager` or with the `release_local()` function but the correct way is to pop the item from the stack after using. When the stack is empty it will no longer be bound to the current context (and as such released).

By calling the stack without arguments it returns a proxy that resolves to the topmost item on the stack.

New in version 0.6.1.

push(*obj*)
Pushes a new item to the stack

pop()
Removes the topmost item from the stack, will return the old value or *None* if the stack was already empty.

top
The topmost item on the stack. If the stack is empty, *None* is returned.

class jobcontrol.utils.local.**LocalProxy**(*local*, *name=None*)
Acts as a proxy for a werkzeug local. Forwards all operations to a proxied object. The only operations not supported for forwarding are right handed operands and any kind of assignment.

Example usage:

```
from werkzeug.local import Local
l = Local()

# these are proxies
request = l('request')
user = l('user')


from werkzeug.local import LocalStack
_response_local = LocalStack()

# this is a proxy
response = _response_local()
```

Whenever something is bound to l.user / l.request the proxy objects will forward all operations. If no object is bound a `RuntimeError` will be raised.

To create proxies to `Local` or `LocalStack` objects, call the object as shown above. If you want to have a proxy to an object looked up by a function, you can (as of Werkzeug 0.6.1) pass a function to the `LocalProxy` constructor:

```
session = LocalProxy(lambda: get_current_request().session)
```

Changed in version 0.6.1: The class can be instanciated with a callable as well now.

4.6.3 jobcontrol.utils.testing

jobcontrol.utils.testing.**job_simple_echo**(**args*, ***kwargs*)

jobcontrol.utils.testing.**testing_job**(*progress_steps=None*, *retval=None*, *fail=False*, *skip=False*, *log_messages=None*, *step_duration=0*)

Job used for testing purposes.

Parameters

- **progress_steps** – A list of tuples: `(<group_name>, <steps>)`, where "group_name" is a tuple of name "levels", "steps" an integer representing how many steps should that level have.

  Progress reports will be sent in randomized order.
- **retval** – The return value for the job.
- **fail** – Whether this job should fail.
- **skip** – Whether this job should be skipped.
- **log_messages** – A list of tuples: `(level, message)`
- **step_duration** – The time to sleep between steps, in milliseconds.

jobcontrol.utils.testing.**job_with_logging**()

jobcontrol.utils.testing.**job_with_tracer_log**()

jobcontrol.utils.testing.**job_failing_once**()

This job will fail exactly once; retry will be successful

jobcontrol.utils.testing.**job_echo_config**(**args*, ***kwargs*)

Simple job, "echoing" back the current configuration.

class jobcontrol.utils.testing.**RecordingLogHandler**

Log handler that records messages

flush()

emit(*record*)

print_messages()

clear_messages()

class jobcontrol.utils.testing.**NonSerializableObject**

foo

bar

exception jobcontrol.utils.testing.**NonSerializableException**

jobcontrol.utils.testing.**job_returning_nonserializable**()

jobcontrol.utils.testing.**job_raising_nonserializable**()

4.6.4 jobcontrol.utils.web

Utilities for the RESTful API

jobcontrol.utils.web.**json_view**(*func*)

jobcontrol.utils.web.**generate_csrf_token**()

class jobcontrol.utils.**cached_property**(*func*, *name=None*, *doc=None*)
A decorator that converts a function into a lazy property. The function wrapped is called the first time to retrieve the result and then that calculated result is used the next time you access the value:

```
class Foo(object):

    @cached_property
    def foo(self):
        # calculate something important here
        return 42
```

The class has to have a *__dict__* in order for this property to work.

jobcontrol.utils.**import_object**(*name*)
Import an object from a module, by name.

Parameters **name** – The object name, in the `package.module:name` format.

Returns The imported object

jobcontrol.utils.**get_storage_from_url**(*url*)
Get a storage from URL.

Storages URLs are in the format:

- `<scheme>://`
- `<class>+<scheme>://` Load <class>, pass the URL removing `<class>+`

jobcontrol.utils.**get_storage_from_config**(*config*)
Not implemented yet

jobcontrol.utils.**short_repr**(*obj*, *maxlen=50*)
Returns a "shortened representation" of an object; that is, the return value of `repr(obj)` limited to a certain length, with a trailing ellipsis `'...'` if text was truncated.

This function is mainly used in order to provide a nice representation of local variables in `TracebackInfo` objects

jobcontrol.utils.**json_dumps**(*obj*)

jobcontrol.utils.**trim_string**(*s*, *maxlen=1024*, *ellps='...'*)
Trim a string to a maximum length, adding an "ellipsis" indicator if the string was trimmed

class jobcontrol.utils.**FrameInfo**(*filename*, *lineno*, *name*, *line*, *locs*)

class jobcontrol.utils.**TracebackInfo**
Class used to hold information about an error traceback.

This is meant to be serialized & stored in the database, instead of a full traceback object, which is *not* serializable.

It holds information about:

- the exception that caused the thing to fail
- the stack frames (with file / line number, function and exact code around the point in which the exception occurred)
- a representation of the local variables for each frame.

A textual representation of the traceback information may be retrieved by using `str()` or `unicode()` on the object instance.

classmethod **from_current_exc**()
Instantiate with traceback from `sys.exc_info()`.

classmethod **from_tb**(*tb*)
Instantiate from a traceback object.

format()
Format traceback for printing

format_color()
Format traceback for printing on 256-color terminal

class jobcontrol.utils.**ProgressReport**(*name*, *current=None*, *total=None*, *status_line=None*, *children=None*)
Class used to represent progress reports.

It supports progress reporting on a multi-level "tree" structure; each level can have its own progress status, or it will generate it automatically by summing up values from children.

current

total

percent

percent_human

progress_label

color_css_rgb

classmethod **from_table**(*table*, *base_name=None*)

Parameters **table** – a list of tuples: (name, current, total, status_line).

- If there is a tuple with `name == None` -> use as the object's current/total report
- Find all the "namespaces" and use to build progress sub-objects

class jobcontrol.utils.**NotSerializableRepr**(*obj*, *exception=None*)

class jobcontrol.utils.**ExceptionPlaceholder**(*orig*)

CHAPTER 5

Indices and tables

- *genindex*
- *modindex*
- *search*

j

B

C

D

E

F

G

H

I

J

L