helper Documentation

Release 2.1.0

Gavin M. Roy

September 24, 2013

CONTENTS

helper is a command-line/daemon application wrapper package with the aim of creating a consistent and fast way to creating applications. It is available on the Python Package Index as helper. helper supports both UNIX (Posix) and Windows applications (in process) and works with Python 2.6, 2.7, 3.2 or 3.3.

CHAPTER

ONE

DOCUMENTATION

1.1 Application Initialization Tool

helper comes with a command line tool *new-helper* which will create a stub helper application project with the following items:

- Python package for the application
- Stub application controller
- Basic configuration file
- RHEL based init.d script
- setup.py file for distributing the application

Usage:

```
usage: new-helper [-h] [--version] PROJECT
```

When you run the application, a tree resembling the following is created:

```
PROJECT/
    etc/
        PROJECT.initd
        PROJECT.yml
    PROJECT/
        __init__.py
        controller.py
    setup.py
```

Where PROJECT is the value you specify when running `new-helper`.

1.2 Getting Started

Creating your first `helper` application is a fairly straightforward process:

1. Download and install helper via pip:

   ```
   pip install helper
   ```

2. Create a new application with the *new-helper* script which will create a stub project including the package directory, configuration file, init.d script for RHEL systems, and setup.py file:

```
new-helper -p myapp
```

3. Open the controller.py file in myapp/myapp/ and you should have a file that looks similar to the following:

```
"""myapp

Helper boilerplate project

"""
import helper
import logging
from helper import parser

DESCRIPTION = 'Project Description'
LOGGER = logging.getLogger(__name__)

class Controller(helper.Controller):
    """The core application controller which is created by invoking
    helper.run().

   """

    def setup(self):
        """Place setup and initialization steps in this method."""
        LOGGER.info('setup invoked')

    def process(self):
        """This method is invoked every wake interval as specified in the
        application configuration. It is fully wrapped and you do not need to
        manage state within it.

        """
        LOGGER.info('process invoked')

    def cleanup(self):
        """Place shutdown steps in this method."""
        LOGGER.info('cleanup invoked')


def main():
    parser.description(DESCRIPTION)
    helper.start(Controller)
```

4. Extend the `Controller.proccess` method to put your core logic in place.
5. If you want to test your app without installing it, I often make a small script in the project directory, something like myapp/myapp.py that looks like the following:

```
#!/usr/bin/env
from myapp import controller
controller.main()
```

6. Change the mode of the file to u+x and run it:

```
chmod u+x myapp.py
./myapp.py -c etc/myapp.yml -f
```

That's about all there is to it. If you don't want to use the sleep/wake/process pattern but want to use an IOLoop, instead of extending `Controller.process`, extend `Controller.run`.

1.3 Configuration Format

helper uses logging.config.dictConfig module to create a flexible method for configuring the python standard logging module. If Python 2.6 is used, logutils.dictconfig.dictConfig is used instead.

YAML is used for the configuration file for helper based applications and will automatically be loaded and referenced for all the required information to start your application. The configuration may be reloaded at runtime by sending a USR1 signal to parent process.

The configuration file has three case-sensitive sections that are required: *Application*, *Daemon*, and *Logging*.

1.3.1 Application

As a generalization, this is where your application's configuration directives go. There is only one core configuration attribute for this section, *wake_interval*. The *wake_interval* value is an integer value that is used for the sleep/wake/process flow and tells helper how often to fire the `Controller.process` method.

1.3.2 Daemon

This section contains the settings required to run the application as a daemon. They are as follows:

user The username to run as when the process is daemonized

group [optional] The group name to switch to when the process is daemonized

pidfile The pidfile to write when the process is daemonized

1.3.3 Logging

As previously mentioned, the Logging section uses the Python standard library dictConfig format. The following basic example illustrates all of the required sections in the dictConfig format, implemented in YAML:

```
version: 1
formatters: []
verbose:
  format: '%(levelname) -10s %(asctime)s %(process)-6d %(processName) -15s %(name) -10s %(funcName) -
  datefmt: '%Y-%m-%d %H:%M:%S'
handlers:
  console:
    class: logging.StreamHandler
    formatter: verbose
    debug_only: True
loggers:
  helper:
    handlers: [console]
    level: INFO
    propagate: true
  myapp:
    handlers: [console]
    level: DEBUG
    propagate: true
disable_existing_loggers: true
incremental: false
```

Note: The debug_only node of the Logging > handlers > console section is not part of the standard dictConfig format. Please see the *Logging Caveats* section below for more information.

Logging Caveats

In order to allow for customizable console output when running in the foreground and no console output when daemonized, a "debug_only" node has been added to the standard dictConfig format in the handler section. This method is evaluated in the helper.Logging and removed, if present, prior to passing the dictionary to dictConfig if present.

If the value is set to true and the application is not running in the foreground, the configuration for the handler and references to it will be removed from the configuration dictionary.

Troubleshooting

If you find that your application is not logging anything or sending output to the terminal, ensure that you have created a logger section in your configuration for your controller. For example if your Controller instance is named MyController, make sure there is a MyController logger in the logging configuration.

1.4 Signal Handling

The `helper.Controller` class will automatically setup and handle signals for your application.

When the `Controller` extended application starts, helper registers handlers for four signals:

- *Handling SIGTERM*
- *Handling SIGHUP*
- *Handling SIGUSR1*
- *Handling SIGUSR2*

Signals received call registered methods within the `Controller` class. If you are using multiprocessing and have child processes, it is up to you to then signal your child processes appropriately.

1.4.1 Handling SIGTERM

In the event that your application receives a `TERM` signal, it will change the internal state of the `Controller` class indicating that the application is shutting down. This may be checked for by checking for a True value from the attribute `Controller.is_stopping` Controller.is_stopping. During this type of shutdown, `Controller.cleanup` will be invoked. This method is meant to be extended by your application for the purposes of cleanly shutting down your application.

1.4.2 Handling SIGHUP

The behavior in `HUP` is to cleanly shutdown the application and then start it back up again. It will, like with `TERM`, call the `Controller.stop` method. Once the shutdown is complete, it will clear the internal state and configuration and then invoke `Controller.run`.

1.4.3 Handling SIGUSR1

If you would like to reload the configuration, sending a `USR1` signal to the parent process of the application will invoke the `Controller.reload_configuration` method, freeing the previously help configuration data from memory and reloading the configuration file from disk. Because it may be desirable to change runtime configuration without restarting the application, it is advised to use the `Controller.config` property method to retrieve configuration values each time instead of holding config values as attributes.

1.4.4 Handling SIGUSR2

This is an unimplemented method within the `Controller` class and is registered for convenience. If have need for custom signal handling, redefine the `Controller.on_signusr2` method in your child class.

1.5 Adding Commandline Arguments

If you would like to add additional command-line options, access helper's argparse based parser adding additional command line arguments as needed. The arguments will be accessible via the *Controller.args* attribute.

Example:

```
from helper import parser

p = parser.get()
p.add_argument('-n', '--newrelic',
               action='store',
               dest='newrelic',
               help='Path to newrelic.init for enabling NewRelic '
                    'instrumentation')
p.add_argument('-p', '--path',
               action='store_true',
               dest='path',
               help='Path to prepend to the Python system path')
```

You can also override the auto-assigned application name:

```
from helper import parser

parser.name('my-app')
```

And the default description:

```
from helper import parser

parser.description('My application rocks!')
```

1.6 clihelper API

1.6.1 Controller

Extend the `Controller` class with your own application implementing the `Controller.process` method. If you do not want to use sleep based looping but rather an IOLoop or some other long-lived blocking loop, redefine the `Controller.run` method.

`Controller` maintains an internal state which is handy for ensuring the proper things are happening at the proper times. The following are the constants used for state transitions:

- `Initializing`
- `Active`
- `Idle`
- `Sleeping`
- `Stop Requested`
- `Stopping`
- `Stopped`

When extending `Controller`, if your class requires initialization or setup setups, extend the `Controller.setup` method.

If your application requires cleanup steps prior to stopping, extend the `Controller.cleanup` method.

class `helper.`**`Controller`**(*args*)

Extend this class to implement your core application controller. Key methods to implement are Controller.setup, Controller.process and Controller.cleanup.

If you do not want to use the sleep/wake structure but rather something like a blocking IOLoop, overwrite the Controller.run method.

APPNAME = 'sphinx-build'

SLEEP_UNIT = 0.5

When shutting down, how long should sleeping block the interpreter while waiting for the state to indicate the class is no longer active.

STATE_ACTIVE = 4

The active state should be set whenever the implementing class is performing a task that can not be interrupted.

STATE_IDLE = 3

The idle state is available to implementing classes to indicate that while they are not actively performing tasks, they are not sleeping. Objects in the idle state can be shutdown immediately.

STATE_INITIALIZING = 1

Initializing state is only set during initial object creation

STATE_SLEEPING = 2

When helper has set the signal timer and is paused, it will be in the sleeping state.

STATE_STOPPED = 7

Once the application has fully stopped, the state is set to stopped.

STATE_STOPPING = 6

Once the application has started to shutdown, it will set the state to stopping and then invoke the `Controller.stopping()` method.

STATE_STOP_REQUESTED = 5

The stop requested state is set when a signal is received indicating the process should stop. The app will invoke the `Controller.stop()` method which will wait for the process state to change from STATE_ACTIVE

VERSION = '2.1.0'

WAKE_INTERVAL = 60

How often should `Controller.process()` be invoked

cleanup()
Override this method to cleanly shutdown the application.

configuration_reloaded()
Override to provide any steps when the configuration is reloaded.

current_state
Property method that return the string description of the runtime state.

Return type str

is_active
Property method that returns a bool specifying if the process is currently active.

Return type bool

is_idle
Property method that returns a bool specifying if the process is currently idle.

Return type bool

is_initializing
Property method that returns a bool specifying if the process is currently initializing.

Return type bool

is_running
Property method that returns a bool specifying if the process is currently running. This will return true if the state is active, idle or initializing.

Return type bool

is_sleeping
Property method that returns a bool specifying if the process is currently sleeping.

Return type bool

is_stopped
Property method that returns a bool specifying if the process is stopped.

Return type bool

is_stopping
Property method that returns a bool specifying if the process is stopping.

Return type bool

is_waiting_to_stop
Property method that returns a bool specifying if the process is waiting for the current process to finish so it can stop.

Return type bool

on_sighup(*signum_unused*, *frame_unused*)
Called when SIGHUP is received, shutdown internal runtime state, reloads configuration and then calls Controller.run(). Can be extended to implement other behaviors.

on_sigterm(*signum_unused*, *frame_unused*)
Called when SIGTERM is received, calling self.stop(). Override to implement a different behavior.

on_sigusr1(*signum_unused*, *frame_unused*)
Called when SIGUSR1 is received, does not have any attached behavior. Override to implement a behavior for this signal.

on_sigusr2(*signum_unused*, *frame_unused*)
Called when SIGUSR2 is received, does not have any attached behavior. Override to implement a behavior for this signal.

process()
To be implemented by the extending class. Is called after every sleep interval in the main application loop.

run()
The core method for starting the application. Will setup logging, toggle the runtime state flag, block on loop, then call shutdown.

Redefine this method if you intend to use an IO Loop or some other long running process.

set_state(*state*)
Set the runtime state of the Controller. Use the internal constants to ensure proper state values:

- `Controller.STATE_INITIALIZING`
- `Controller.STATE_ACTIVE`
- `Controller.STATE_IDLE`
- `Controller.STATE_SLEEPING`
- `Controller.STATE_STOP_REQUESTED`
- `Controller.STATE_STOPPING`
- `Controller.STATE_STOPPED`

Parameters **state** (*int*) – The runtime state

Raises ValueError

setup()
Override to provide any required setup steps.

setup_signals()

shutdown()
Override to provide any required shutdown steps.

start()
Important:

Do not extend this method, rather redefine Controller.run

stop()
Override to implement shutdown steps.

wake_interval
Property method that returns the wake interval in seconds.

Return type int

1.6.2 Logging

The `Logging` class is included as a convenient wrapper to handle Python 2.6 and Python 2.7 dictConfig differences as well as to manage the `helper` specific `debug_only` Handler setting.

If you want to use the default console only logging for helper, you do not need to implement this configuration section. Any configuration you specify merges with the default configuration.

Default Configuration

```
disable_existing_loggers: true
filters: {}
formatters:
  verbose:
    datefmt: '%Y-%m-%d %H:%M:%S'
    format: '%(levelname) -10s %(asctime)s %(process)-6d %(processName) -15s %(threadName)-10s %(name
handlers:
  console:
    class: logging.StreamHandler
    debug_only: true
    formatter: verbose
incremental: false
loggers:
  helper:
    handlers: [console]
    level: INFO
    propagate: true
root:
  handlers: []
  level: 50
  propagate: true
version: 1
```

class helper.config.**LoggingConfig**(*configuration*, *debug=None*)

The Logging class is used for abstracting away dictConfig logging semantics and can be used by sub-processes to ensure consistent logging rule application.

configure()

Configure the Python stdlib logger

update(*configuration*, *debug=None*)

Update the internal configuration values, removing debug_only handlers if debug is False. Returns True if the configuration has changed from previous configuration values.

Parameters

- **configuration** (*dict*) – The logging configuration
- **debug** (*bool*) – Toggles use of debug_only loggers

Return type bool

1.7 Troubleshooting

If you find that you start your application and it immediately dies without any output on the screen, be sure to check for *Unhandled Exceptions*.

1.7.1 Unhandled Exceptions

By default helper will write any unhandled exceptions to a file in one of the following paths:

UNIX: - /var/log/<APPNAME>.errors - /var/tmp/<APPNAME>.errors - /tmp/<APPNAME>.errors

Windows: - Not implemented yet.

1.8 Version History

2.1.0 - 2013-09-24 - Bugfixes: Use pidfile from configuration if specified, don't show warning about not having a logger in helper.unix if no logger is defined, config obj default/value assignment methodology 2.0.2 - 2013-08-28 - Fix a bug where wake_interval default was not used if wake_interval was not provided in the config. Make logging config an overlay of the default logging config. 2.0.1 - 2013-08-28 - setup.py bugfix 2.0.0 - 2013-08-28 - clihelper renamed to helper with a major refactor. Windows support still pending.

Version: 2.1.0

Github Page: https://github.com/gmr/helper

Author: Gavin M. Roy <gavinmroy@gmail.com>

License: Released under the BSD license

CHAPTER

TWO

INDICES AND TABLES

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