python-service Documentation

Release 0.5.2

Florian Brucker

Apr 28, 2019

Contents

CHAPTER 1

Easy Implementation of Background Services

This package makes it easy to write Unix services, i.e. background processes ("daemons") that are controlled by a foreground application (e.g. a console script).

The package is built around the python-daemon module, which provides the means for creating well-behaved daemon processes. The `service` package adds a control infrastructure for easily starting, stopping, querying and killing the background process from a foreground application.

CHAPTER 2

Installation

The `service` package is available from PyPI and can be installed via pip:

```
pip install service
```

Supported Python versions are 2.7 as well as 3.4 and later.

CHAPTER 3

Quickstart

```
import logging
from logging.handlers import SysLogHandler
import time

from service import find_syslog, Service

class MyService(Service):
    def __init__(self, *args, **kwargs):
        super(MyService, self).__init__(*args, **kwargs)
        self.logger.addHandler(SysLogHandler(address=find_syslog(),
                               facility=SysLogHandler.LOG_DAEMON))
        self.logger.setLevel(logging.INFO)

    def run(self):
        while not self.got_sigterm():
            self.logger.info("I'm working...")
            time.sleep(5)

if __name__ == '__main__':
    import sys

    if len(sys.argv) != 2:
        sys.exit('Syntax: %s COMMAND' % sys.argv[0])

    cmd = sys.argv[1].lower()
    service = MyService('my_service', pid_dir='/tmp')

    if cmd == 'start':
        service.start()
    elif cmd == 'stop':
        service.stop()
    elif cmd == 'status':
        if service.is_running():
            print "Service is running."
```

(continues on next page)

(continued from previous page)

```
        else:
            print "Service is not running."
    else:
        sys.exit('Unknown command "%s".' % cmd)
```

CHAPTER 4

Control Interface

The *Service* class has a dual interface: Some methods control the daemon and are intended to be called from the controlling process while others implement the actual daemon functionality or utilities for it.

The control methods are:

- *start()* to start the daemon
- *stop()* to ask the daemon to stop
- *kill()* to kill the daemon
- *is_running()* to check whether the daemon is running
- *get_pid()* to get the daemon's process ID
- *send_signal()* to send arbitrary signals to the daemon

Subclasses usually do not need to override any of these.

CHAPTER 5

Daemon Functionality

To provide the actual daemon functionality, subclasses override *run()*, which is executed in a separate daemon process when *start()* is called. Once *run()* exits, the daemon process stops.

When *stop()* is called, the SIGTERM signal is sent to the daemon process, which can check for its reception using *got_sigterm()* or wait for it using *wait_for_sigterm()*.

Further signals to control the daemon can be specified using the `signals` constructor argument. These signals can then be sent to the daemon process using *send_signal()*. The daemon process can use *got_signal()*, *wait_for_signal()*, and *clear_signal()* to react to signals.

CHAPTER 6

Logging

Instances of *Service* provide a built-in logger via their *logger* attribute. By default the logger only has a logging.NullHandler attached, so all messages are discarded. Attach your own handler to output log messages to files or syslog (see the handlers provided by the logging and logging.handlers modules).

Any uncaught exceptions from *run()* are automatically logged via that logger. To avoid error messages during startup being lost make sure to attach your logging handlers before calling *start()*.

If you want use syslog for logging take a look at *find_syslog()*, which provides a portable way of locating syslog.

CHAPTER 7

Preserving File Handles

By default, all open file handles are released by the daemon process. If you need to preserve some of them add them to the *files_preserve* list attribute. Note that file handles used by any built-in Python logging handlers attached to *logger* are automatically preserved.

CHAPTER 8

Exiting the Service

From the outside, a service can be stopped gracefully by calling *stop()* or, as a last resort, by calling *kill()*.

From the inside, i.e. from within *run()*, the easiest way is to just `return` from the method. From version 0.5 on you can also call `sys.exit` and it will be handled correctly (in earlier versions that would prevent a correct clean up). Note that you should never use `os._exit`, since that skips all clean up.

CHAPTER 9

API Reference

service.**find_syslog**()
Find Syslog.

Returns Syslog's location on the current system in a form that can be passed on to `logging.handlers.SysLogHandler`:

```
handler = SysLogHandler(address=find_syslog(),
                        facility=SysLogHandler.LOG_DAEMON)
```

class service.**Service**
A background service.

This class provides the basic framework for running and controlling a background daemon. This includes methods for starting the daemon (including things like proper setup of a detached deamon process), checking whether the daemon is running, asking the daemon to terminate and for killing the daemon should that become necessary.

logger
A `logging.Logger` instance.

files_preserve
A list of file handles that should be preserved by the daemon process. File handles of built-in Python logging handlers attached to *`logger`* are automatically preserved.

__init__(*name*, *pid_dir='/var/run'*, *signals=None*)
Constructor.

`name` is a string that identifies the daemon. The name is used for the name of the daemon process, the PID file and for the messages to syslog.

`pid_dir` is the directory in which the PID file is stored.

`signals` list of operating signals, that should be available for use with *`send_signal()`*, *`got_signal()`*, *`wait_for_signal()`*, and `check_signal()`. Note that SIGTERM is always supported, and that SIGTTIN, SIGTTOU, and SIGTSTP are never supported.

clear_signal(*s*)
Clears the state of a signal.

The signal must have been enabled using the `signals` parameter of *Service.__init__()*. Otherwise, a `ValueError` is raised.

get_pid()
Get PID of daemon process or `None` if daemon is not running.

got_signal(*s*)
Check if a signal was received.

The signal must have been enabled using the `signals` parameter of *Service.__init__()*. Otherwise, a `ValueError` is raised.

Returns `True` if the daemon process has received the signal (for example because *stop()* was called in case of SIGTERM, or because *send_signal()* was used) and `False` otherwise.

Note: This function always returns `False` for enabled signals when it is not called from the daemon process.

---

got_sigterm()
Check if SIGTERM signal was received.

Returns `True` if the daemon process has received the SIGTERM signal (for example because *stop()* was called).

Note: This function always returns `False` when it is not called from the daemon process.

---

is_running()
Check if the daemon is running.

kill(*block=False*)
Kill the daemon process.

Sends the SIGKILL signal to the daemon process, killing it. You probably want to try *stop()* first.

If `block` is true then the call blocks until the daemon process has exited. `block` can either be `True` (in which case it blocks indefinitely) or a timeout in seconds.

Returns `True` if the daemon process has (already) exited and `False` otherwise.

The PID file is always removed, whether the process has already exited or not. Note that this means that subsequent calls to *is_running()* and *get_pid()* will behave as if the process has exited. If you need to be sure that the process has already exited, set `block` to `True`.

New in version 0.5.1: The `block` parameter

run()
Main daemon method.

This method is called once the daemon is initialized and running. Subclasses should override this method and provide the implementation of the daemon's functionality. The default implementation does nothing and immediately returns.

Once this method returns the daemon process automatically exits. Typical implementations therefore contain some kind of loop.

The daemon may also be terminated by sending it the SIGTERM signal, in which case *run()* should terminate after performing any necessary clean up routines. You can use *got_sigterm()* and *wait_for_sigterm()* to check whether SIGTERM has been received.

send_signal(*s*)
Send a signal to the daemon process.

The signal must have been enabled using the `signals` parameter of *Service.__init__()*. Otherwise, a `ValueError` is raised.

start(*block=False*)
Start the daemon process.

The daemon process is started in the background and the calling process returns.

Once the daemon process is initialized it calls the *run()* method.

If `block` is true then the call blocks until the daemon process has started. `block` can either be `True` (in which case it blocks indefinitely) or a timeout in seconds.

The return value is `True` if the daemon process has been started and `False` otherwise.

New in version 0.3: The `block` parameter

stop(*block=False*)
Tell the daemon process to stop.

Sends the SIGTERM signal to the daemon process, requesting it to terminate.

If `block` is true then the call blocks until the daemon process has exited. This may take some time since the daemon process will complete its on-going backup activities before shutting down. `block` can either be `True` (in which case it blocks indefinitely) or a timeout in seconds.

The return value is `True` if the daemon process has been stopped and `False` otherwise.

New in version 0.3: The `block` parameter

wait_for_signal(*s*, *timeout=None*)
Wait until a signal has been received.

The signal must have been enabled using the `signals` parameter of *Service.__init__()*. Otherwise, a `ValueError` is raised.

This function blocks until the daemon process has received the signal (for example because *stop()* was called in case of SIGTERM, or because *send_signal()* was used).

If `timeout` is given and not `None` it specifies a timeout for the block.

The return value is `True` if the signal was received and `False` otherwise (the latter occurs if a timeout was given and the signal was not received).

> **Warning:** This function blocks indefinitely (or until the given timeout) for enabled signals when it is not called from the daemon process.

wait_for_sigterm(*timeout=None*)
Wait until a SIGTERM signal has been received.

This function blocks until the daemon process has received the SIGTERM signal (for example because *stop()* was called).

If `timeout` is given and not `None` it specifies a timeout for the block.

The return value is `True` if SIGTERM was received and `False` otherwise (the latter only occurs if a timeout was given and the signal was not received).

> **Warning:** This function blocks indefinitely (or until the given timeout) when it is not called from the daemon process.

CHAPTER 10

Development

The code for this package can be found on GitHub. It is available under the MIT license.

CHAPTER 11

Change Log

See the file CHANGELOG.md.

Python Module Index

S

Index

Symbols

C

F

G

I

K

L

R

S

W