PyTest Salt Factories

Release 1.0.0

VMware, Inc.

Mar 21, 2024

CONTENTS

CHAPTER

ONE

INSTALL SALT FACTORIES

Installing Salt Factories is as simple as:

```
python -m pip install pytest-salt-factories
```

And, that's honestly it.

1.1 Salt

Salt factories does not define Salt as a hard requirement because that would create a chicken and egg problem while testing Salt itself. This is not a problem while testing code outside of the Salt repository.

To install Salt along with Salt Factories:

```
python -m pip install 'pytest-salt-factories[salt]'
```

1.2 Docker

Salt factories also supports container factories using docker containers. To have container support enabled, install Salt Factories along with docker:

```
python -m pip install 'pytest-salt-factories[docker]'
```

1.3 Multiple Optional Dependencies

Installing salt-factories with multiple optional dependencies is also simple.

```
python -m pip install 'pytest-salt-factories[salt,docker]'
```

CHAPTER

TWO

USING SALT FACTORIES

Salt factories simplifies testing Salt related code outside of Salt's source tree. A great example is a salt-extension.

Let's consider this `echo-extension` example.

The `echo-extension` provides an execution module:

Listing 1: examples/echo-extension/src/echoext/modules/echo_mod.py

```
__virtualname__ = "echo"


def __virtual__():
    return __virtualname__


def text(string):
    """
    This function just returns any text that it's given.

    CLI Example:

    .. code-block:: bash

        salt '*' echo.text 'foo bar baz quo qux'
    """
    return __salt__["test.echo"](string)


def reverse(string):
    """
    This function just returns any text that it's given, reversed.

    CLI Example:

    .. code-block:: bash

        salt '*' echo.reverse 'foo bar baz quo qux'
    """
    return __salt__["test.echo"](string)[::-1]
```

And also a state module:

Listing 2: examples/echo-extension/src/echoext/states/echo_mod.py

```
__virtualname__ = "echo"


def __virtual__():
    if "echo.text" not in __salt__:
        return False, "The 'echo' execution module is not available"
    return __virtualname__


def echoed(name):
    ret = {"name": name, "changes": {}, "result": False, "comment": ""}
    value = __salt__["echo.text"](name)
    if value == name:
        ret["result"] = True
        ret["comment"] = f"The 'echo.echoed' returned: '{value}'"
    return ret


def reversed(name):
    """
    This example function should be replaced
    """
    ret = {"name": name, "changes": {}, "result": False, "comment": ""}
    value = __salt__["echo.reverse"](name)
    if value == name[::-1]:
        ret["result"] = True
        ret["comment"] = f"The 'echo.reversed' returned: '{value}'"
    return ret
```

One could start off with something simple like unit testing the extension's code.

2.1 Unit Tests

Listing 3: examples/echo-extension/tests/unit/modules/test_echo.py

```
import pytest
import salt.modules.test as testmod

import echoext.modules.echo_mod as echo_module


@pytest.fixture
def configure_loader_modules():
    module_globals = {
        "__salt__": {"test.echo": testmod.echo},
    }
    return {
        echo_module: module_globals,
    }
```

(continues on next page)

(continued from previous page)

```
def test_text():
    echo_str = "Echoed!"
    assert echo_module.text(echo_str) == echo_str


def test_reverse():
    echo_str = "Echoed!"
    expected = echo_str[::-1]
    assert echo_module.reverse(echo_str) == expected
```

Listing 4: examples/echo-extension/tests/unit/states/test_echo.py

```
import pytest
import salt.modules.test as testmod

import echoext.modules.echo_mod as echo_module
import echoext.states.echo_mod as echo_state


@pytest.fixture
def configure_loader_modules():
    return {
        echo_module: {
            "__salt__": {
                "test.echo": testmod.echo,
            },
        },
        echo_state: {
            "__salt__": {
                "echo.text": echo_module.text,
                "echo.reverse": echo_module.reverse,
            },
        },
    }


def test_echoed():
    echo_str = "Echoed!"
    expected = {
        "name": echo_str,
        "changes": {},
        "result": True,
        "comment": f"The 'echo.echoed' returned: '{echo_str}'",
    }
    assert echo_state.echoed(echo_str) == expected


def test_reversed():
    echo_str = "Echoed!"
    expected_str = echo_str[::-1]
```

(continues on next page)

(continued from previous page)

```
    expected = {
        "name": echo_str,
        "changes": {},
        "result": True,
        "comment": f"The 'echo.reversed' returned: '{expected_str}'",
    }
    assert echo_state.reversed(echo_str) == expected
```

The *magical* piece of code in the above example is the *configure_loader_modules* fixture.

2.2 Integration Tests

Listing 5: examples/echo-extension/tests/conftest.py

```
import os

import pytest

from echoext import PACKAGE_ROOT
from saltfactories.utils import random_string


@pytest.fixture(scope="session")
def salt_factories_config():
    """
    Return a dictionary with the keyword arguments for FactoriesManager
    """
    coverage_rc_path = os.environ.get("COVERAGE_PROCESS_START")
    if coverage_rc_path:
        coverage_db_path = PACKAGE_ROOT / ".coverage"
    else:
        coverage_db_path = None
    return {
        "code_dir": str(PACKAGE_ROOT),
        "coverage_rc_path": coverage_rc_path,
        "coverage_db_path": coverage_db_path,
        "inject_sitecustomize": "COVERAGE_PROCESS_START" in os.environ,
        "start_timeout": 120 if os.environ.get("CI") else 60,
    }


@pytest.fixture(scope="package")
def master(salt_factories):
    return salt_factories.salt_master_daemon(random_string("master-"))


@pytest.fixture(scope="package")
def minion(master):
    return master.salt_minion_daemon(random_string("minion-"))
```

Listing 6: examples/echo-extension/tests/integration/conftest.py

```
import pytest


@pytest.fixture(scope="package")
def master(master):
    with master.started():
        yield master


@pytest.fixture(scope="package")
def minion(minion):
    with minion.started():
        yield minion


@pytest.fixture
def salt_run_cli(master):
    return master.salt_run_cli()


@pytest.fixture
def salt_cli(master):
    return master.salt_cli()


@pytest.fixture
def salt_call_cli(minion):
    return minion.salt_call_cli()
```

Listing 7: examples/echo-extension/tests/integration/modules/test_echo.py

```
import pytest

pytestmark = [
    pytest.mark.requires_salt_modules("echo.text"),
]


def test_text(salt_call_cli):
    echo_str = "Echoed!"
    ret = salt_call_cli.run("echo.text", echo_str)
    assert ret.returncode == 0
    assert ret.data
    assert ret.data == echo_str


def test_reverse(salt_call_cli):
    echo_str = "Echoed!"
    expected = echo_str[::-1]
    ret = salt_call_cli.run("echo.reverse", echo_str)
```

(continues on next page)

(continued from previous page)

```
    assert ret.returncode == 0
    assert ret.data
    assert ret.data == expected
```

Listing 8: examples/echo-extension/tests/integration/states/test_echo.py

```
import pytest

pytestmark = [
    pytest.mark.requires_salt_states("echo.text"),
]


def test_echoed(salt_call_cli):
    echo_str = "Echoed!"
    ret = salt_call_cli.run("state.single", "echo.echoed", echo_str)
    assert ret.returncode == 0
    assert ret.data
    assert ret.data == echo_str


def test_reversed(salt_call_cli):
    echo_str = "Echoed!"
    expected = echo_str[::-1]
    ret = salt_call_cli.run("state.single", "echo.reversed", echo_str)
    assert ret.returncode == 0
    assert ret.data
    assert ret.data == expected
```

What happened above?

1. We started a salt master
2. We started a salt minion
3. The minion connects to the master
4. The master accepted the minion's key automatically
5. We pinged the minion

A litle suggestion

Not all tests should be integration tests, in fact, only a small set of the test suite should be an integration test.

CHAPTER

THREE

MARKERS

Salt factories ships with a few markers, skip markers. Additional markers used in Salt's test suite are provided by the skip-markers pytest plugin.

3.1 requires_salt_modules

@pytest.mark.**requires_salt_modules**(**modules*)

Parameters

modules (*str*) – Each argument passed to the marker should be a salt execution module that will need to be loaded by salt, or the test will be skipped. Allowed values are the module name, for example `cmd`, or the module name with the function name, `cmd.run`.

```
@pytest.mark.requires_salt_modules("cmd", "archive.tar")
def test_func():
    assert True
```

3.2 requires_salt_states

@pytest.mark.**requires_salt_states**(**modules*)

Parameters

modules (*str*) – Each argument passed to the marker should be a salt state module that will need to be loaded by salt, or the test will be skipped. Allowed values are the state module name, for example `pkg`, or the state module name with the function name, `pkg.installed`.

```
@pytest.mark.requires_salt_states("pkg", "archive.extracted")
def test_func():
    assert True
```

3.3 skip_on_salt_system_service

@pytest.mark.**skip_on_salt_system_service**(*reason=None*)

Parameters

reason (*str*) – Custom skip reason. Defaults to "Test should not run against system install of Salt".

```
@pytest.mark.skip_on_salt_system_service
def test_func():
    assert True
```

CHAPTER

FOUR

FIXTURES

saltfactories.plugins.event_listener.**event_listener**()

Event listener session scoped fixture.

All started daemons will forward their events into an instance of *EventListener*.

This fixture can be used to wait for events:

```
def test_send(event_listener, salt_master, salt_minion, salt_call_cli):
    event_tag = random_string("salt/test/event/")
    data = {"event.fire": "just test it!!!!"}
    start_time = time.time()
    ret = salt_call_cli.run("event.send", event_tag, data=data)
    assert ret.returncode == 0
    assert ret.data
    assert ret.data is True

    event_pattern = (salt_master.id, event_tag)
    matched_events = event_listener.wait_for_events(
        [event_pattern], after_time=start_time, timeout=30
    )
    assert matched_events.found_all_events
    # At this stage, we got all the events we were waiting for
```

And assert against those events events:

```
def test_send(event_listener, salt_master, salt_minion, salt_call_cli):
    # ... check the example above for the initial code ...
    assert matched_events.found_all_events
    # At this stage, we got all the events we were waiting for
    for event in matched_events:
        assert event.data["id"] == salt_minion.id
        assert event.data["cmd"] == "_minion_event"
        assert "event.fire" in event.data["data"]
```

4.1 configure_loader_modules

Note

The `configure_loader_modules` fixture is meant to be used on unit-tests, the `pytest-salt-factories` plugin does not define it anywhere. Instead, the user must define it on the test module.

The fixture **must** return a dictionary, where the keys are the salt modules that need to be patched, and the values are dictionaries. These dictionaries should have the salt dunders as keys. These dunders are dictionaries that the salt loader injects at runtime, so, they are not available outside of Salt's runtime.

CHAPTER

FIVE

SALT FACTORIES PACKAGE

5.1 `salt-factories` CLI Script

The `salt-factories` CLI script is meant to be used to get an absolute path to the directory containing `sitecustomize.py` so that it can be injected into `PYTHONPATH` when running tests to track subprocesses code coverage.

Example:

```
export PYTHONPATH="$(salt-factories --coverage);$PYTHONPATH"
```

Please check the coverage documentation on the additional requirements to track code coverage on subprocesses.

5.2 Plugins

5.2.1 Event Listener

Salt Factories Event Listener.

A salt events store for all daemons started by salt-factories

class saltfactories.plugins.event_listener.**Event**(*, *daemon_id*, *tag*, *stamp*, *data*, *full_data*, *expire_seconds*)

Bases: `object`

Event wrapper class.

The `Event` class is a container for a salt event which will live on the *`EventListener`* store.

Parameters

- **daemon_id** (*str*) – The daemon ID which received this event.
- **tag** (*str*) – The event tag of the event.
- **stamp** (*datetime*) – When the event occurred
- **data** (*dict*) – The event payload, filtered of all of Salt's private keys like `_stamp` which prevents proper assertions against it.
- **full_data** (*dict*) – The full event payload, as received by the daemon, including all of Salt's private keys.
- **expire_seconds** (*int*, *float*) – The time, in seconds, after which the event should be considered as expired and removed from the store.

property expired

Property to identify if the event has expired, at which time it should be removed from the store.

class saltfactories.plugins.event_listener.**MatchedEvents**(*, *matches*, *missed*)

Bases: object

MatchedEvents implementation.

The MatchedEvents class is a container which is returned by *wait_for_events()*.

Parameters

- **matches** (set) – A set of *Event* instances that matched.
- **missed** (set) – A set of *Event* instances that remained unmatched.

One can also easily iterate through all matched events of this class:

```
matched_events = MatchedEvents(..., ...)
for event in matched_events:
    print(event.tag)
```

property found_all_events

Return bool

True if all events were matched, or False otherwise.

class saltfactories.plugins.event_listener.**EventListenerServer**(*_event_listener*, **args*, ***kwargs*)

Bases: Protocol

TCP Server to receive events forwarded.

connection_made(*transport*)

Connection established.

data_received(*data*)

Received data.

connection_lost(*exc*)

Called when the connection is lost or closed.

The argument is an exception object or None (the latter meaning a regular EOF is received or the connection was aborted or closed).

eof_received()

Called when the other end calls write_eof() or equivalent.

If this returns a false value (including None), the transport will close itself. If it returns a true value, closing the transport is up to the protocol.

pause_writing()

Called when the transport's buffer goes over the high-water mark.

Pause and resume calls are paired – pause_writing() is called once when the buffer goes strictly over the high-water mark (even if subsequent writes increases the buffer size even more), and eventually resume_writing() is called once when the buffer size reaches the low-water mark.

Note that if the buffer size equals the high-water mark, pause_writing() is not called – it must go strictly over. Conversely, resume_writing() is called when the buffer size is equal or lower than the low-water mark. These end conditions are important to ensure that things go as expected when either mark is zero.

NOTE: This is the only Protocol callback that is not called through EventLoop.call_soon() – if it were, it would have no effect when it's most needed (when the app keeps writing without yielding until pause_writing() is called).

resume_writing()

Called when the transport's buffer drains below the low-water mark.

See pause_writing() for details.

class saltfactories.plugins.event_listener.**EventListener**(*, *timeout=120*)

Bases: object

EventListener implementation.

The EventListener is a service started by salt-factories which receives all the events of all the salt masters that it starts. The service runs throughout the whole pytest session.

Parameters

timeout (*int*) – How long, in seconds, should a forwarded event stay in the store, after which, it will be deleted.

start_server()

Start the TCP server.

start()

Start the event listener.

stop()

Stop the event listener.

get_events(*patterns*, *after_time=None*)

Get events from the internal store.

Parameters

- **pattern** (*Sequence*) – An iterable of tuples in the form of ("<daemon-id>", "<event-tag-pattern>"), ie, which daemon ID we're targeting and the event tag pattern which will be passed to fnmatch() to assert a match.
- **after_time** (*datetime*,*float*) – After which time to start matching events.

Return set

A set of matched events

wait_for_events(*patterns*, *timeout=30*, *after_time=None*)

Wait for a set of patterns to match or until timeout is reached.

Parameters

- **pattern** (*Sequence*) – An iterable of tuples in the form of ("<daemon-id>", "<event-tag-pattern>"), ie, which daemon ID we're targeting and the event tag pattern which will be passed to fnmatch() to assert a match.
- **timeout** (*int*,*float*) – The amount of time to wait for the events, in seconds.
- **after_time** (*datetime*,*float*) – After which time to start matching events.

Returns

An instance of *MatchedEvents*.

Rtype ~saltfactories.plugins.event_listener.MatchedEvents

register_auth_event_handler(*master_id*, *callback*)

Register a callback to run for every authentication event, to accept or reject the minion authenticating.

Parameters

- **master_id** (*str*) – The master ID for which the callback should run
- **callback** (*Callable*) – The function while should be called

unregister_auth_event_handler(*master_id*)

Un-register the authentication event callback, if any, for the provided master ID.

Parameters

master_id (*str*) – The master ID for which the callback is registered

saltfactories.plugins.event_listener.**event_listener**()

Event listener session scoped fixture.

All started daemons will forward their events into an instance of *EventListener*.

This fixture can be used to wait for events:

```
def test_send(event_listener, salt_master, salt_minion, salt_call_cli):
    event_tag = random_string("salt/test/event/")
    data = {"event.fire": "just test it!!!!"}
    start_time = time.time()
    ret = salt_call_cli.run("event.send", event_tag, data=data)
    assert ret.returncode == 0
    assert ret.data
    assert ret.data is True

    event_pattern = (salt_master.id, event_tag)
    matched_events = event_listener.wait_for_events(
        [event_pattern], after_time=start_time, timeout=30
    )
    assert matched_events.found_all_events
    # At this stage, we got all the events we were waiting for
```

And assert against those events events:

```
def test_send(event_listener, salt_master, salt_minion, salt_call_cli):
    # ... check the example above for the initial code ...
    assert matched_events.found_all_events
    # At this stage, we got all the events we were waiting for
    for event in matched_events:
        assert event.data["id"] == salt_minion.id
        assert event.data["cmd"] == "_minion_event"
        assert "event.fire" in event.data["data"]
```

5.2.2 Loader

Salt loader mock support for tests.

saltfactories.plugins.loader.**pytest_collection_modifyitems**(*items*)

Modify the collected items.

Iterate through the collected items, in particular their test modules, to see if there's a function named `configure_loader_modules`. If there is, assert that it's a fixture. If not, raise an error.

saltfactories.plugins.loader.**setup_loader_mock**(*request*)

Setup Salt's loader mocking/patching if the test module defines a `configure_loader_modules` fixture.

5.2.3 System Information

The system information plugin can be enabled by passing `--sys-info` to pytest.

When enabled it will include some output sections when starting pytest. Here's an example of the output(partial, for brevity):

```
>>>>>>>>>>>>>>>>>>>>>>>>>>> System Information >>>>>>>>>>>>>>>>>>>>>>>>>>>
------------------------- Salt Versions Report -------------------------
  Salt Version:
            Salt: 3003

  Dependency Versions:
            cffi: Not Installed
        cherrypy: Not Installed
        dateutil: Not Installed
       docker-py: 5.0.0
           gitdb: Not Installed
       gitpython: Not Installed
          Jinja2: 3.0.1
         libgit2: Not Installed
        M2Crypto: Not Installed
            Mako: Not Installed
         msgpack: 1.0.2
    msgpack-pure: Not Installed
    mysql-python: Not Installed
       pycparser: Not Installed
        pycrypto: Not Installed
    pycryptodome: 3.10.1
          pygit2: Not Installed
          Python: 3.7.7 (default, Oct 24 2021, 07:30:53)
    python-gnupg: Not Installed
          PyYAML: 5.4.1
           PyZMQ: 22.1.0
           smmap: Not Installed
         timelib: Not Installed
         Tornado: 4.5.3
             ZMQ: 4.3.4

  System Versions:
            dist: arch rolling n/a
```

(continues on next page)

(continued from previous page)

```
          locale: UTF-8
         machine: x86_64
         release: 5.16.2-arch1-1
          system: Linux
         version: Arch Linux rolling n/a
------------------------- System Grains Report -------------------------
  biosreleasedate: 12/06/2019
  biosversion: N1EET87W (1.60 )
  cpu_flags:
  - fpu
  - vme
  gpus:
   - model: HD Graphics 530
     vendor: intel
   - model: GM107GLM [Quadro M1000M]
     vendor: nvidia
  kernelrelease: 5.16.2-arch1-1
  kernelversion: '#1 SMP PREEMPT Thu, 20 Jan 2022 16:18:29 +0000'
  locale_info:
    defaultencoding: UTF-8
    defaultlanguage: pt_PT
    detectedencoding: UTF-8
    timezone: unknown
  mem_total: 64137
  num_cpus: 8
  num_gpus: 2
  os: Arch
  os_family: Arch
  osarch: x86_64
  oscodename: n/a
  osfinger: Arch-rolling
  osfullname: Arch
  osrelease: rolling
  virtual: physical
  zfs_feature_flags: false
  zfs_support: false
  zmqversion: 4.3.4
<<<<<<<<<<<<<<<<<<<<<<<<<<< System Information <<<<<<<<<<<<<<<<<<<<<<<<<<<
```

5.2.4 Log Server

The Salt Factories Log Server is responsible to receive log records from the salt daemons.

Because all of Salt's daemons and CLI tools are started in subprocesses, there's really no easy way to get those logs into the main process where the test suite is running.

However, salt is extensible by nature, and it provides a way to attach custom log handlers into python's logging machinery.

We take advantage of that and add a custom logging handler into subprocesses we start for salt. That logging handler will then forward **all** log records into this log server, which in turn, injects them into the logging machinery running in the test suite.

This allows one to use PyTest's `caplog fixture` to assert against log messages.

As an example:

```
def test_baz(caplog):
    func_under_test()
    for record in caplog.records:
        assert record.levelname != "CRITICAL"
    assert "wally" not in caplog.text
```

5.2.5 Daemon & CLI Factories

Salt Daemon Factories PyTest Plugin.

saltfactories.plugins.factories.**salt_factories_config**()

Default salt factories configuration fixture.

saltfactories.plugins.factories.**salt_factories_default_root_dir**()

The root directory from where to base all paths.

For example, in a salt system installation, this would be `/`.

Attention

If *root_dir* is returned on the *salt_factories_config()* fixture dictionary, then that's the value used, and not the one returned by this fixture.

---

saltfactories.plugins.factories.**salt_factories**(*event_listener*, *stats_processes*, *salt_factories_default_root_dir*, *salt_factories_config*, *_salt_factories_config*)

Instantiate the salt factories manager.

saltfactories.plugins.factories.**pytest_addoption**(*parser*)

Register argparse-style options and ini-style config values.

5.3 Utils

Utility functions.

saltfactories.utils.**random_string**(*prefix*, *size=6*, *uppercase=True*, *lowercase=True*, *digits=True*)

Generates a random string.

Parameters

- **prefix** (*str*) – The prefix for the random string
- **size** (*int*) – The size of the random string
- **uppercase** (*bool*) – If true, include upper-cased ascii chars in choice sample
- **lowercase** (*bool*) – If true, include lower-cased ascii chars in choice sample
- **digits** (*bool*) – If true, include digits in choice sample

Return str

The random string

saltfactories.utils.**running_username**()

Return the username that is running the code.

saltfactories.utils.**cast_to_pathlib_path**(*value*)

Cast the passed value to an instance of `pathlib.Path`.

saltfactories.utils.**warn_until**(*version*, *message*, *category=<class 'DeprecationWarning'>*, *stacklevel=None*, *_dont_call_warnings=False*, *_pkg_version_=None*)

Show a deprecation warning.

Helper function to raise a warning, by default, a `DeprecationWarning`, until the provided `version`, after which, a `RuntimeError` will be raised to remind the developers to remove the warning because the target version has been reached.

Parameters

- **version** (*str*) – The version string after which the warning becomes a `RuntimeError`. For example `2.1`.
- **message** (*str*) – The warning message to be displayed.
- **category** (*Type[Warning]*) – The warning class to be thrown, by default `DeprecationWarning`
- **stacklevel** (*int | None*) – There should be no need to set the value of `stacklevel`.
- **_dont_call_warnings** (*bool*) – This parameter is used just to get the functionality until the actual error is to be issued. When we're only after the version checks to raise a `RuntimeError`.
- **_pkg_version_** (*str | None*) –

Return type
None

5.3.1 saltfactories.utils.cli_scripts

Code to generate Salt CLI scripts for test runs.

saltfactories.utils.cli_scripts.**generate_script**(*bin_dir*, *script_name*, *code_dir=None*, *inject_sitecustomize=False*, *coverage_db_path=None*, *coverage_rc_path=None*)

Generate a CLI script.

Parameters

- **bin_dir** (*Path*) – The path to the directory which will contain the CLI scripts
- **script_name** (*str*) – The CLI script name
- **code_dir** (*Path*) – The project's being tested root directory path
- **inject_sitecustomize** (*bool*) – Inject code to support code coverage in subprocesses
- **coverage_db_path** (*Path*) – The path to the *.coverage* DB file
- **coverage_rc_path** (*Path*) – The path to the *.coveragerc* file

5.3.2 saltfactories.utils.functional

Salt functional testing support.

class saltfactories.utils.functional.**Loaders**(*opts*, *loaded_base_name=None*)

Bases: object

This class provides the required functionality for functional testing against the salt loaders.

Parameters
opts (*dict*) – The options dictionary to load the salt loaders.

Example usage:

```
import salt.config
from saltfactories.utils.functional import Loaders


@pytest.fixture(scope="module")
def minion_opts():
    return salt.config.minion_config(None)


@pytest.fixture(scope="module")
def loaders(minion_opts):
    return Loaders(minion_opts)


@pytest.fixture(autouse=True)
def reset_loaders_state(loaders):
    try:
        # Run the tests
        yield
    finally:
        # Reset the loaders state
        loaders.reset_state()
```

reset_state()

Reset the state functions state.

reload_all()

Reload all loaders.

property grains

The grains loaded by the salt loader.

property utils

The utils loaded by the salt loader.

property modules

The execution modules loaded by the salt loader.

property serializers

The serializers loaded by the salt loader.

property states

The state modules loaded by the salt loader.

property pillar

The pillar loaded by the salt loader.

refresh_pillar()

Refresh the pillar.

class saltfactories.utils.functional.**StateResult**(*raw*)

Bases: object

This class wraps a single salt state return into a more pythonic object in order to simplify assertions.

Parameters

raw (*dict*) – A single salt state return result

```
def test_user_absent(loaders):
    ret = loaders.states.user.absent(name=random_string("account-",␣
↪uppercase=False))
    assert ret.result is True
```

property run_num

The __run_num__ key on the full state return dictionary.

property id

The __id__ key on the full state return dictionary.

property name

The name key on the full state return dictionary.

property result

The result key on the full state return dictionary.

property changes

The changes key on the full state return dictionary.

property comment: MatchString

The comment key on the full state return dictionary.

property warnings

The warnings key on the full state return dictionary.

class saltfactories.utils.functional.**StateFunction**(*proxy_func*, *state_func*)

Bases: object

Salt state module functions wrapper.

Simple wrapper around Salt's state execution module functions which actually proxies the call through Salt's state.single execution module

class saltfactories.utils.functional.**MultiStateResult**(*raw*)

Bases: object

Multiple state returns wrapper class.

This class wraps multiple salt state returns, for example, running the state.sls execution module, into a more pythonic object in order to simplify assertions

Parameters

raw (*dict*,*list*) – The multiple salt state returns result, a dictionary on success or a list on failure

Example usage on the test suite:

```
def test_issue_1876_syntax_error(loaders, state_tree, tmp_path):
    testfile = tmp_path / "issue-1876.txt"
    sls_contents = """
    {}:
      file:
        - managed
        - source: salt://testfile

      file.append:
        - text: foo
    """.format(
        testfile
    )
    with pytest.helpers.temp_file("issue-1876.sls", sls_contents, state_tree):
        ret = loaders.modules.state.sls("issue-1876")
        assert ret.failed
        errmsg = (
            "ID '{}' in SLS 'issue-1876' contains multiple state declarations of the
↪"
            " same type".format(testfile)
        )
        assert errmsg in ret.errors


def test_pydsl(loaders, state_tree, tmp_path):
    testfile = tmp_path / "testfile"
    sls_contents = """
    #!pydsl

    state("{}").file("touch")
    """.format(
        testfile
    )
    with pytest.helpers.temp_file("pydsl.sls", sls_contents, state_tree):
        ret = loaders.modules.state.sls("pydsl")
        for staterun in ret:
            assert staterun.result is True
        assert testfile.exists()
```

property failed

Return `True` or `False` if the multiple state run was not successful.

property errors

Return the list of errors in case the multiple state run was not successful.

class saltfactories.utils.functional.**StateModuleFuncWrapper**(*func*, *wrapper*)

Bases: `object`

This class simply wraps a single or multiple state returns into a more pythonic object.

`StateResult` or py:class:*~saltfactories.utils.functional.MultiStateResult*

Parameters

- **func** (*callable*) – A salt loader function

- **wrapper** (StateResult,MultiStateResult) – The wrapper to use for the return of the salt loader function's return

5.3.3 saltfactories.utils.loader

Salt's Loader PyTest Mock Support.

class saltfactories.utils.loader.**LoaderModuleMock**(*setup_loader_modules*, *, *salt_module_dunders=('__opts__', '__salt__', '__runner__', '__context__', '__utils__', '__ext_pillar__', '__thorium__', '__states__', '__serializers__', '__ret__', '__grains__', '__pillar__', '__sdb__')*, *salt_module_dunders_optional=('__proxy__',)*, *salt_module_dunder_attributes=('__env__', '__low__', '__instance_id__', '__orchestration_jid__', '__jid_event__', '__active_provider_name__', '__proxyenabled__')*)

Bases: object

Salt Loader mock class.

start()

Start mocks.

stop()

Stop mocks.

addfinalizer(*func*, **args*, ***kwargs*)

Register a function to run when stopping.

5.3.4 PyTest Markers related utilities

Markers related utilities.

saltfactories.utils.markers.**check_required_loader_attributes**(*loader_instance*, *loader_attr*, *required_items*)

Check if the salt loaders has the passed required items.

Parameters

- **loader_instance** (Loaders) – An instance of *Loaders*
- **loader_attr** (*str*) – The name of the minion attribute to check, such as 'modules' or 'states'
- **required_items** (*tuple*) – The items that must be part of the loader attribute for the decorated test

Returns

The modules that are not available

Return type

set

saltfactories.utils.markers.**evaluate_markers**(*item*)

Fixtures injection based on markers or test skips based on CLI arguments.

5.3.5 Temporary files utilities

Temporary files helpers.

saltfactories.utils.tempfiles.**temp_directory**(*name=None*, *basepath=None*)

This helper creates a temporary directory.

It should be used as a context manager which returns the temporary directory path, and, once out of context, deletes it.

Parameters

- **name** (*basepath*) – The name of the directory to create
- **name** – The base path of where to create the directory. Defaults to `gettempdir()`

Return type
pathlib.Path

Can be directly imported and used:

```
from saltfactories.utils.tempfiles import temp_directory


def test_func():
    with temp_directory() as temp_path:
        assert temp_path.is_dir()

    assert not temp_path.is_dir() is False
```

Or, it can be used as a pytest helper function:

```
import pytest


def test_blah():
    with pytest.helpers.temp_directory() as temp_path:
        assert temp_path.is_dir()

    assert not temp_path.is_dir() is False
```

saltfactories.utils.tempfiles.**temp_file**(*name=None*, *contents=None*, *directory=None*, *strip_first_newline=True*)

Create a temporary file as a context manager.

This helper creates a temporary file. It should be used as a context manager which returns the temporary file path, and, once out of context, deletes it.

Parameters

- **name** (*str*) – The temporary file name
- **contents** (*str*) – The contents of the temporary file
- **directory** (*str*, *pathlib.Path*) – The directory where to create the temporary file. Defaults to the value of `gettempdir()`

- **strip_first_newline** (*bool*) – Either strip the initial first new line char or not.

Return type
pathlib.Path

Can be directly imported and used:

```
from saltfactories.utils.tempfiles import temp_file


def test_func():
    with temp_file(name="blah.txt") as temp_path:
        assert temp_path.is_file()

    assert temp_path.is_file() is False
```

Or, it can be used as a pytest helper function:

```
import pytest


def test_blah():
    with pytest.helpers.temp_file("blah.txt") as temp_path:
        assert temp_path.is_file()

    assert temp_path.is_file() is False
```

To create files under a sub-directory, one has two choices:

```
import pytest


def test_relative_subdirectory():
    with pytest.helpers.temp_file("foo/blah.txt") as temp_path:
        assert temp_path.is_file()
        assert temp_path.parent.is_dir()
        assert temp_path.parent.name == "foo"

    assert not temp_path.is_file() is False
    assert not temp_path.parent.is_dir() is False
```

```
import os
import pytest
import tempfile

ROOT_DIR = tempfile.gettempdir()


def test_absolute_subdirectory_1():
    destpath = os.path.join(ROOT_DIR, "foo")
    with pytest.helpers.temp_file("blah.txt", directory=destpath) as temp_path:
        assert temp_path.is_file()
        assert temp_path.parent.is_dir()
        assert temp_path.parent.name == "foo"
```

(continues on next page)

(continued from previous page)

```
    assert not temp_path.is_file() is False
    assert not temp_path.parent.is_dir() is False


def test_absolute_subdirectory_2():
    destpath = os.path.join(ROOT_DIR, "foo", "blah.txt")
    with pytest.helpers.temp_file(destpath) as temp_path:
        assert temp_path.is_file()
        assert temp_path.parent.is_dir()
        assert temp_path.parent.name == "foo"

    assert temp_path.is_file() is False
    assert temp_path.parent.is_dir() is False
```

class saltfactories.utils.tempfiles.**SaltEnv**(*, *name*, *paths=_Nothing.NOTHING*)

Bases: object

This helper class represent a Salt Environment, either for states or pillar.

It's base purpose it to handle temporary file creation/deletion during testing.

Parameters

- **name** (*str*) – The salt environment name, commonly, 'base' or 'prod'
- **paths** (*list*) – The salt environment list of paths.

Note

The first entry in this list, is the path that will get used to create temporary files in, ie, the return value of the *saltfactories.utils.tempfiles.SaltEnv.write_path* attribute.

property write_path

The path where temporary files are created.

temp_file(*name*, *contents=None*, *strip_first_newline=True*)

Create a temporary file within this saltenv.

Please check *saltfactories.utils.tempfiles.temp_file()* for documentation.

Note

The `directory` keyword is not supported(since the directory used will be the value of *saltfactories.utils.tempfiles.SaltEnv.write_path*. To place a file within a sub-directory, give path with directory for file, for example: "mydir/myfile".

as_dict()

Returns a dictionary of the right types to update the salt configuration.

Return dict

class saltfactories.utils.tempfiles.**SaltEnvs**(*, *envs*)

Bases: object

This class serves as a container for multiple salt environments for states or pillar.

Parameters

envs (*dict*) – The *envs* dictionary should be a mapping of a string as key, the *saltenv*, commonly 'base' or 'prod', and the value an instance of *SaltEnv* or a list of strings(paths). In the case where a list of strings(paths) is passed, it is converted to an instance of *SaltEnv*

To provide a better user experience, the salt environments can be accessed as attributes of this class.

```
envs = SaltEnvs(
    {
        "base": [
            "/path/to/base/env",
        ],
        "prod": [
            "/path/to/prod/env",
        ],
    }
)
with envs.base.temp_file("foo.txt", "foo contents") as base_foo_path:
    ...
with envs.prod.temp_file("foo.txt", "foo contents") as prod_foo_path:
    ...
```

as_dict()

Returns a dictionary of the right types to update the salt configuration.

Return dict

class saltfactories.utils.tempfiles.**SaltStateTree**(*, *envs*)

Bases: *SaltEnvs*

Helper class which handles temporary file creation within the state tree.

Parameters

envs (*dict*) – A mapping of a `saltenv` to a list of paths.

```
envs = {
    "base": [
        "/path/to/base/env",
        "/another/path/to/base/env",
    ],
    "prod": [
        "/path/to/prod/env",
        "/another/path/to/prod/env",
    ],
}
```

The state tree environments can be accessed by attribute:

```
# See example of envs definition above
state_tree = SaltStateTree(envs=envs)

# To access the base saltenv
base = state_tree.envs["base"]

# Alternatively, in a simpler form
base = state_tree.base
```

When setting up the Salt configuration to use an instance of *SaltStateTree*, the following pseudo code can be followed.

```
# Using the state_tree defined above:
salt_config = {
    # ... other salt config entries ...
    "file_roots": state_tree.as_dict()
    # ... other salt config entries ...
}
```

Attention

The temporary files created by the `temp_file()` are written to the first path passed when instantiating the `SaltStateTree`, ie, the return value of the `saltfactories.utils.tempfiles.SaltStateTree.write_path` attribute.

```
# Given the example mapping shown above ...

with state_tree.base.temp_file("foo.sls") as path:
    assert str(path) == "/path/to/base/env/foo.sls"
```

---

as_dict()

Returns a dictionary of the right types to update the salt configuration.

Return dict

class saltfactories.utils.tempfiles.**SaltPillarTree**(*, *envs*)

Bases: *SaltEnvs*

Helper class which handles temporary file creation within the pillar tree.

Parameters

envs (*dict*) – A mapping of a `saltenv` to a list of paths.

```
envs = {
    "base": [
        "/path/to/base/env",
        "/another/path/to/base/env",
    ],
    "prod": [
        "/path/to/prod/env",
        "/another/path/to/prod/env",
    ],
}
```

The pillar tree environments can be accessed by attribute:

```
# See example of envs definition above
pillar_tree = SaltPillarTree(envs=envs)

# To access the base saltenv
base = pillar_tree.envs["base"]

# Alternatively, in a simpler form
base = pillar_tree.base
```

When setting up the Salt configuration to use an instance of *SaltPillarTree*, the following pseudo code can be followed.

```
# Using the pillar_tree defined above:
salt_config = {
    # ... other salt config entries ...
    "pillar_roots": pillar_tree.as_dict()
    # ... other salt config entries ...
}
```

Attention

The temporary files created by the `temp_file()` are written to the first path passed when instantiating the `SaltPillarTree`, ie, the return value of the `saltfactories.utils.tempfiles.SaltPillarTree.write_path` attribute.

```
# Given the example mapping shown above ...

with state_tree.base.temp_file("foo.sls") as path:
    assert str(path) == "/path/to/base/env/foo.sls"
```

as_dict()

Returns a dictionary of the right types to update the salt configuration.

Return dict

5.3.6 Salt Extensions

saltfactories.utils.saltext.**get_engines_dirs**()

Return a list of directories for Salt to look for engine extensions.

saltfactories.utils.saltext.**get_log_handlers_dirs**()

Return a list of directories for Salt to look for log handlers extensions.

PyTest Salt Engine

Salt Factories Engine For Salt.

Simple salt engine which will setup a socket to accept connections allowing us to know when a daemon is up and running

saltfactories.utils.saltext.engines.pytest_engine.**start**()

Method to start the engine.

saltfactories.utils.saltext.engines.pytest_engine.**ext_type_encoder**(*obj*)

Convert any types that msgpack cannot handle on it's own.

class saltfactories.utils.saltext.engines.pytest_engine.**PyTestEventForwardClient**(*queue*, *client_running_event*)

Bases: `Protocol`

TCP Client to forward events.

connection_made(*transport*)

Connection established.

connection_lost(*exc*)

Connection lost.

async wait_connected()

Wait until a connection to the server is successful.

async wait_disconnected()

Wait until disconnected from the server.

data_received(*data*)

Called when some data is received.

The argument is a bytes object.

eof_received()

Called when the other end calls write_eof() or equivalent.

If this returns a false value (including None), the transport will close itself. If it returns a true value, closing the transport is up to the protocol.

pause_writing()

Called when the transport's buffer goes over the high-water mark.

Pause and resume calls are paired – pause_writing() is called once when the buffer goes strictly over the high-water mark (even if subsequent writes increases the buffer size even more), and eventually resume_writing() is called once when the buffer size reaches the low-water mark.

Note that if the buffer size equals the high-water mark, pause_writing() is not called – it must go strictly over. Conversely, resume_writing() is called when the buffer size is equal or lower than the low-water mark. These end conditions are important to ensure that things go as expected when either mark is zero.

NOTE: This is the only Protocol callback that is not called through EventLoop.call_soon() – if it were, it would have no effect when it's most needed (when the app keeps writing without yielding until pause_writing() is called).

resume_writing()

Called when the transport's buffer drains below the low-water mark.

See pause_writing() for details.

class saltfactories.utils.saltext.engines.pytest_engine.**PyTestEventForwardEngine**(*opts*)

Bases: object

Salt Engine instance.

start()

Start the engine.

stop()

Stop the engine.

PyTest Salt Log Handler

Salt External Logging Handler.

saltfactories.utils.saltext.log_handlers.pytest_log_handler.**setup_handlers**()

Setup the handlers.

class saltfactories.utils.saltext.log_handlers.pytest_log_handler.**ZMQHandler**(**args*, ***kwargs*)

Bases: ExcInfoOnLogLevelFormatMixin, Handler

ZMQ logging handler implementation.

setFormatter(_)

Overridden method to show an error.

start()

Start the handler.

stop(*flush=True*)

Stop the handler.

format(*record*)

Format the log record.

prepare(*record*)

Prepare the log record.

emit(*record*)

Emit a record.

Writes the LogRecord to the queue, preparing it for pickling first.

close()

Tidy up any resources used by the handler.

acquire()

Acquire the I/O thread lock.

addFilter(*filter*)

Add the specified filter to this handler.

createLock()

Acquire a thread lock for serializing access to the underlying I/O.

filter(*record*)

Determine if a record is loggable by consulting all the filters.

The default is to allow the record to be logged; any filter can veto this and the record is then dropped. Returns a zero value if a record is to be dropped, else non-zero.

Changed in version 3.2: Allow filters to be just callables.

flush()

Ensure all logging output has been flushed.

This version does nothing and is intended to be implemented by subclasses.

handle(*record*)

Conditionally emit the specified logging record.

Emission depends on filters which may have been added to the handler. Wrap the actual emission of the record with acquisition/release of the I/O thread lock. Returns whether the filter passed the record for emission.

handleError(*record*)

Handle errors which occur during an emit() call.

This method should be called from handlers when an exception is encountered during an emit() call. If raiseExceptions is false, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The record which was being processed is passed in to this method.

release()

Release the I/O thread lock.

removeFilter(*filter*)

Remove the specified filter from this handler.

setLevel(*level*)

Set the logging level of this handler. level must be an int or a str.

5.4 Exceptions

PyTest Salt Factories related exceptions.

5.5 Factories

5.5.1 Base Classes

Factories base classes.

class saltfactories.bases.**SaltMixin**(*, *config*, *config_dir=_Nothing.NOTHING*, *python_executable=None*, *system_service=False*)

Bases: `object`

Base factory for salt cli's and daemon's.

Parameters

- **config** (*dict*) – The Salt config dictionary
- **python_executable** (*str*) – The path to the python executable to use
- **system_service** (*bool*) – If true, the daemons and CLI's are run against a system installed salt setup, ie, the default salt system paths apply.

get_display_name()

Returns a human readable name for the factory.

class saltfactories.bases.**SaltCliImpl**(*, *factory*)

Bases: SubprocessImpl

Salt CLI's subprocess interaction implementation.

Please look at SubprocessImpl for the additional supported keyword arguments documentation.

Parameters
factory (*Factory | Subprocess | ScriptSubprocess*) –

cmdline(**args*, *minion_tgt=None*, ***kwargs*)

Construct a list of arguments to use when starting the subprocess.

Parameters

- **args** (*str*) – Additional arguments to use when starting the subprocess
- **minion_tgt** (*str*) – The minion ID to target
- **kwargs** – Additional keyword arguments will be converted into `key=value` pairs to be consumed by the salt CLI's

init_terminal(*cmdline*, *shell=False*, *env=None*, *cwd=None*)

Instantiate a terminal with the passed command line(`cmdline`) and return it.

Additionally, it sets a reference to it in `self._terminal` and also collects an initial listing of child processes which will be used when terminating the terminal

Arguments:

cmdline:
List of strings to pass as `args` to Popen

Keyword Arguments:

shell:
Pass the value of `shell` to Popen

env:
A dictionary of `key`, `value` pairs to add to the `pytestshellutils.shell.Factory.environ`.

cwd:
A path for the CWD when running the process.

Returns:
A Popen instance.

Parameters

- **cmdline** (*List[str]*) –
- **shell** (*bool*) –
- **env** (*EnvironDict | None*) –
- **cwd** (*str | Path | None*) –

Return type
Popen[*Any*]

is_running()

Returns true if the sub-process is alive.

Returns:
Returns true if the sub-process is alive

Return type
bool

property pid: int | None

The pid of the running process. None if not running.

run(**args*, *shell=False*, *env=None*, *cwd=None*, ***kwargs*)

Run the given command synchronously.

Arguments:

args:
The command to run.

Keyword Arguments:

shell:
Pass the value of *shell* to `pytestshellutils.shell.Factory.init_terminal()`

env:
A dictionary of `key`, `value` pairs to add to the `pytestshellutils.shell.Factory.environ`.

cwd:
A path for the CWD when running the process.

Returns:
A Popen instance.

Parameters

- **args** (*str*) –
- **shell** (*bool*) –
- **env** (*EnvironDict | None*) –
- **cwd** (*str | Path | None*) –
- **kwargs** (*Any*) –

Return type
Popen[*Any*]

terminate()

Terminate the started subprocess.

Return type
ProcessResult

class saltfactories.bases.SaltCli(*, *config*, *config_dir=_Nothing.NOTHING*, *python_executable=None*, *system_service=False*, *cwd=_Nothing.NOTHING*, *environ=_Nothing.NOTHING*, *slow_stop=True*, *system_encoding=_Nothing.NOTHING*, *timeout=_Nothing.NOTHING*, *script_name*, *base_script_args=_Nothing.NOTHING*, *hard_crash=False*, *merge_json_output=True*)

Bases: *SaltMixin*, ScriptSubprocess

Base factory for salt cli's.

Parameters

- **hard_crash** (*bool*) – Pass `--hard-crash` to Salt's CLI's

- **cwd** (*Path*) –
- **environ** (*EnvironDict*) –
- **slow_stop** (*bool*) –
- **system_encoding** (*str*) –
- **timeout** (*int | float*) –
- **script_name** (*str*) –
- **base_script_args** (*List[str]*) –

Please look at `Salt` and `ScriptSubprocess` for the additional supported keyword arguments documentation.

`get_script_args()`

Returns any additional arguments to pass to the CLI script.

`get_minion_tgt`(*minion_tgt=None*)

Return the minion target ID.

`cmdline`(**args*, *minion_tgt=None*, *merge_json_output=None*, ***kwargs*)

Construct a list of arguments to use when starting the subprocess.

Parameters

- **args** (*str*) – Additional arguments to use when starting the subprocess
- **minion_tgt** (*str*) – The minion ID to target
- **merge_json_output** (*bool*) – The default behavior of salt outputters is to print one line per minion return, which makes parsing the whole output as JSON impossible when targeting multiple minions. If this value is `True`, an attempt is made to merge each JSON line into a single dictionary.
- **kwargs** – Additional keyword arguments will be converted into `key=value` pairs to be consumed by the salt CLI's

`process_output`(*stdout*, *stderr*, *cmdline=None*)

Process the output. When possible JSON is loaded from the output.

Returns
Returns a tuple in the form of `(stdout, stderr, loaded_json)`

Return type
tuple

`get_base_script_args()`

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

`get_display_name()`

Returns a human readable name for the factory.

`get_script_path()`

Returns the path to the script to run.

Return type
str

`is_running()`

Returns true if the sub-process is alive.

Return type
bool

`property pid: int | None`

The pid of the running process. None if not running.

`run(*args, env=None, _timeout=None, **kwargs)`

Run the given command synchronously.

Keyword Arguments:

args:
The list of arguments to pass to `cmdline()` to construct the command to run

env:
Pass a dictionary of environment key, value pairs to inject into the subprocess.

_timeout:
The timeout value for this particular `run()` call. If this value is not `None`, it will be used instead of `timeout`, the default timeout.

Parameters

- **args** (*str*) –
- **env** (*EnvironDict | None*) –
- **_timeout** (*int | float | None*) –
- **kwargs** (*Any*) –

Return type
ProcessResult

`terminate()`

Terminate the started subprocess.

Return type
ProcessResult

`class saltfactories.bases.SystemdSaltDaemonImpl(*, factory, before_start_callbacks=_Nothing.NOTHING, after_start_callbacks=_Nothing.NOTHING, before_terminate_callbacks=_Nothing.NOTHING, after_terminate_callbacks=_Nothing.NOTHING)`

Bases: `DaemonImpl`

Daemon systemd interaction implementation.

Please look at `DaemonImpl` for the additional supported keyword arguments documentation.

Parameters

- **factory** (*Daemon*) –
- **before_start_callbacks** (*List[Callback]*) –
- **after_start_callbacks** (*List[Callback]*) –
- **before_terminate_callbacks** (*List[Callback]*) –

- **after_terminate_callbacks** (*List[Callback]*) –

cmdline(**args*)

Construct a list of arguments to use when starting the subprocess.

Parameters
args (*str*) – Additional arguments to use when starting the subprocess

get_service_name()

Return the systemd service name.

is_running()

Returns true if the sub-process is alive.

property pid

Return the `pid` of the running process.

start(**extra_cli_arguments*, *max_start_attempts=None*, *start_timeout=None*)

Start the daemon.

after_start(*callback*, **args*, ***kwargs*)

Register a function callback to run after the daemon starts.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

after_terminate(*callback*, **args*, ***kwargs*)

Register a function callback to run after the daemon terminates.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

before_start(*callback*, **args*, ***kwargs*)

Register a function callback to run before the daemon starts.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

before_terminate(*callback*, **args*, ***kwargs*)

Register a function callback to run before the daemon terminates.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

`get_start_arguments()`

Return the arguments and keyword arguments used when starting the daemon.

Return type
StartDaemonCallArguments

`init_terminal(cmdline, shell=False, env=None, cwd=None)`

Instantiate a terminal with the passed command line(`cmdline`) and return it.

Additionally, it sets a reference to it in `self._terminal` and also collects an initial listing of child processes which will be used when terminating the terminal

Arguments:

cmdline:
List of strings to pass as `args` to `Popen`

Keyword Arguments:

shell:
Pass the value of `shell` to `Popen`

env:
A dictionary of `key`, `value` pairs to add to the `pytestshellutils.shell.Factory.environ`.

cwd:
A path for the CWD when running the process.

Returns:
A `Popen` instance.

Parameters

- **cmdline** (*List[str]*) –
- **shell** (*bool*) –
- **env** (*EnvironDict | None*) –
- **cwd** (*str | Path | None*) –

Return type
Popen[*Any*]

`run(*args, shell=False, env=None, cwd=None, **kwargs)`

Run the given command synchronously.

Arguments:

args:
The command to run.

Keyword Arguments:

shell:
Pass the value of *shell* to `pytestshellutils.shell.Factory.init_terminal()`

env:
A dictionary of `key`, `value` pairs to add to the `pytestshellutils.shell.Factory.environ`.

cwd:
A path for the CWD when running the process.

Returns:
A `Popen` instance.

Parameters

- **args** (*str*) –
- **shell** (*bool*) –
- **env** (*EnvironDict | None*) –
- **cwd** (*str | Path | None*) –
- **kwargs** (*Any*) –

Return type
Popen[*Any*]

`terminate()`

Terminate the daemon.

Return type
ProcessResult

`class saltfactories.bases.SaltDaemon(*, config, config_dir=_Nothing.NOTHING, python_executable=None, system_service=False, cwd=_Nothing.NOTHING, environ=_Nothing.NOTHING, slow_stop=True, system_encoding=_Nothing.NOTHING, timeout=_Nothing.NOTHING, script_name, base_script_args=_Nothing.NOTHING, check_ports=_Nothing.NOTHING, stats_processes=None, start_timeout, max_start_attempts=3, extra_cli_arguments_after_first_start_failure=_Nothing.NOTHING, start_checks_callbacks=_Nothing.NOTHING, event_listener=None, factories_manager=None, started_at=None)`

Bases: *SaltMixin*, `Daemon`

Base factory for salt daemon's.

Please look at *SaltMixin* and `Daemon` for the additional supported keyword arguments documentation.

Parameters

- **cwd** (*str | Path*) –
- **environ** (*EnvironDict*) –
- **slow_stop** (*bool*) –

- **system_encoding** (*str*) –
- **timeout** (*int | float*) –
- **script_name** (*str*) –
- **base_script_args** (*List[str]*) –
- **check_ports** (*List[int]*) –
- **stats_processes** (*StatsProcesses*) –
- **start_timeout** (*int | float*) –
- **max_start_attempts** (*int*) –
- **extra_cli_arguments_after_first_start_failure** (*List[str]*) –
- **start_checks_callbacks** (*List[Callback]*) –

classmethod configure(*factories_manager*, *daemon_id*, *root_dir=None*, *defaults=None*, *overrides=None*, ***configure_kwargs*)

Configure the salt daemon.

classmethod verify_config(*config*)

Verify the configuration dictionary.

classmethod write_config(*config*)

Write the configuration to file.

classmethod load_config(*config_file*, *config*)

Return the loaded configuration.

Should return the configuration as the daemon would have loaded after parsing the CLI

get_check_events()

Return salt events to check.

Returns list of tuples in the form of *(master_id, event_tag)* check against to ensure the daemon is running.

cmdline(**args*)

Construct a list of arguments to use when starting the subprocess.

Parameters
args (*str*) – Additional arguments to use when starting the subprocess

after_start(*callback*, **args*, ***kwargs*)

Register a function callback to run after the daemon starts.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

`after_terminate`(*callback*, **args*, ***kwargs*)

Register a function callback to run after the daemon terminates.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

`before_start`(*callback*, **args*, ***kwargs*)

Register a function callback to run before the daemon starts.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

`before_terminate`(*callback*, **args*, ***kwargs*)

Register a function callback to run before the daemon terminates.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

`get_base_script_args`()

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

`get_check_ports`()

Return a list of ports to check against to ensure the daemon is running.

Return type
List[int]

`get_display_name`()

Returns a human readable name for the factory.

`get_script_args`()

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

`get_script_path`()

Returns the path to the script to run.

Return type
str

`get_start_check_callbacks()`

Return a list of the start check callbacks.

Return type
List[*Callback*]

`is_running()`

Returns true if the sub-process is alive.

Return type
bool

`property pid: int | None`

The pid of the running process. None if not running.

`process_output(stdout, stderr, cmdline=None)`

Process the output. When possible JSON is loaded from the output.

Returns:
Returns a tuple in the form of `(stdout, stderr, loaded_json)`

Parameters

- **stdout** (*str*) –
- **stderr** (*str*) –
- **cmdline** (*List[str] | None*) –

Return type
Tuple[str, str, *Dict*[*Any*, *Any*] | None]

`run(*args, env=None, _timeout=None, **kwargs)`

Run the given command synchronously.

Keyword Arguments:

args:
The list of arguments to pass to `cmdline()` to construct the command to run

env:
Pass a dictionary of environment key, value pairs to inject into the subprocess.

_timeout:
The timeout value for this particular `run()` call. If this value is not `None`, it will be used instead of `timeout`, the default timeout.

Parameters

- **args** (*str*) –
- **env** (*EnvironDict | None*) –
- **_timeout** (*int | float | None*) –
- **kwargs** (*Any*) –

Return type
ProcessResult

run_start_checks(*started_at*, *timeout_at*)

Run checks to confirm that the daemon has started.

Parameters

- **started_at** (*float*) –
- **timeout_at** (*float*) –

Return type
bool

start(**extra_cli_arguments*, *max_start_attempts=None*, *start_timeout=None*)

Start the daemon.

Parameters

- **extra_cli_arguments** (*str*) –
- **max_start_attempts** (*int | None*) –
- **start_timeout** (*int | float | None*) –

Return type
bool

start_check(*callback*, **args*, ***kwargs*)

Register a function to run after the daemon starts to confirm readiness for work.

The callback must accept as the first argument `timeout_at` which is a float. The callback must stop trying to confirm running behavior once `time.time() > timeout_at`. The callback should return `True` to confirm that the daemon is ready for work.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Example:

```
def check_running_state(timeout_at: float) -> bool:
    while time.time() <= timeout_at:
        # run some checks
        ...
        # if all is good
        break
    else:
        return False
    return True
```

Parameters

- **callback** (*Callable[[...], bool]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

started(**extra_cli_arguments*, *max_start_attempts=None*, *start_timeout=None*)

Start the daemon and return it's instance so it can be used as a context manager.

Parameters

- **extra_cli_arguments** (*str*) –
- **max_start_attempts** (*int | None*) –
- **start_timeout** (*int | float | None*) –

Return type
Generator[*Daemon*, None, None]

stopped(*before_stop_callback=None*, *after_stop_callback=None*, *before_start_callback=None*, *after_start_callback=None*)

Stop the daemon and return it's instance so it can be used as a context manager.

Keyword Arguments:

before_stop_callback:
A callable to run before stopping the daemon. The callback must accept one argument, the daemon instance.

after_stop_callback:
A callable to run after stopping the daemon. The callback must accept one argument, the daemon instance.

before_start_callback:
A callable to run before starting the daemon. The callback must accept one argument, the daemon instance.

after_start_callback:
A callable to run after starting the daemon. The callback must accept one argument, the daemon instance.

This context manager will stop the factory while the context is in place, it re-starts it once out of context.

Example:

```
assert factory.is_running() is True

with factory.stopped():
    assert factory.is_running() is False

assert factory.is_running() is True
```

Parameters

- **before_stop_callback** (*Callable[[Daemon], None] | None*) –
- **after_stop_callback** (*Callable[[Daemon], None] | None*) –
- **before_start_callback** (*Callable[[Daemon], None] | None*) –

- **after_start_callback** (*Callable[[Daemon], None] | None*) –

Return type
Generator[*Daemon*, None, None]

terminate()

Terminate the started subprocess.

Return type
ProcessResult

5.5.2 Salt Factories Manager

Salt Factories Manager.

class saltfactories.manager.**FactoriesManager**(*, *root_dir*, *log_server_port*, *log_server_level*, *log_server_host*, *python_executable=None*, *scripts_dir=None*, *code_dir=None*, *coverage_db_path=None*, *coverage_rc_path=None*, *inject_sitecustomize=False*, *cwd=_Nothing.NOTHING*, *environ=_Nothing.NOTHING*, *slow_stop=True*, *start_timeout=None*, *stats_processes=None*, *system_service=False*, *event_listener*)

Bases: object

Factories manager implementation.

The *FactoriesManager* is responsible for configuring and spawning Salt Daemons and making sure that any salt CLI tools are "targeted" to the right daemon.

It also keeps track of which daemons were started and adds their termination routines to PyTest's request finalization routines.

If process statistics are enabled, it also adds the started daemons to those statistics.

Arguments:

root_dir:
The root directory from where to base all paths. For example, in a salt system installation, this would be `/`.

log_server_port:
The port the log server should listen at

log_server_level:
The level of the log server

log_server_host:
The hostname/ip address of the host running the logs server. Defaults to "localhost".

Keyword Arguments:

python_executable:
The python executable to use, where needed. If `scripts_dir` is not `None`, then `python_executable` will default to `None`, otherwise, defaults to py:attr:*sys.executable*.

scripts_dir:
The paths to the directory containing the Salt CLI scripts. The several scripts to the Salt daemons and CLI's **must** exist. Passing this option will also make salt-factories **NOT** generate said scripts and set `python_executable` to `None`.

code_dir:
The path to the code root directory of the project being tested. This is important for proper code-coverage paths.

coverage_db_path:
The path to the *.coverage* DB file

coverage_rc_path:
The path to the *.coveragerc* file

inject_sitecustomize:
Inject code in the generated CLI scripts in order for our *sitecustomise.py* to be loaded by subprocesses.

cwd:
The path to the current working directory

environ:
A dictionary of *key*, *value* pairs to add to the environment.

slow_stop:
Whether to terminate the processes by sending a `SIGTERM` signal or by calling `terminate()` on the sub-process. When code coverage is enabled, one will want *slow_stop* set to *True* so that coverage data can be written down to disk.

start_timeout:
The amount of time, in seconds, to wait, until a subprocess is considered as not started.

stats_processes:
This will be an `pytestsysstats.plugin.StatsProcesses` class instantiated on the `pytest_sessionstart()` hook accessible as a session scoped *stats_processes* fixture.

system_service:
If true, the daemons and CLI's are run against a system installed salt setup, ie, the default salt system paths apply and the daemon and CLI scripts will be searched for in `$PATH`.

`static get_salt_log_handlers_path()`
Returns the path to the Salt log handler this plugin provides.

`static get_salt_engines_path()`
Returns the path to the Salt engine this plugin provides.

`final_minion_config_tweaks(config)`
Final tweaks to the minion configuration.

`final_master_config_tweaks(config)`
Final tweaks to the master configuration.

`final_syndic_config_tweaks(config)`
Final tweaks to the syndic configuration.

`final_proxy_minion_config_tweaks(config)`
Final tweaks to the proxy-minion configuration.

`final_cloud_config_tweaks(config)`
Final tweaks to the cloud configuration.

`final_spm_config_tweaks(config)`
Final tweaks to the spm configuration.

`final_common_config_tweaks`(*config*, *role*)

Final common tweaks to the configuration.

`salt_master_daemon`(*master_id*, *order_masters=False*, *master_of_masters=None*, *defaults=None*, *overrides=None*, *max_start_attempts=3*, *start_timeout=None*, *factory_class=<class 'saltfactories.daemons.master.SaltMaster'>*, ***factory_class_kwargs*)

Return a salt-master instance.

Args:

master_id(str):
The master ID

order_masters(bool):
Boolean flag to set if this master is going to control other masters(ie, master of masters), like, for example, in a Syndic topology scenario

master_of_masters(*saltfactories.daemons.master.SaltMaster*):
A *saltfactories.daemons.master.SaltMaster* instance, like, for example, in a Syndic topology scenario

defaults(dict):
A dictionary of default configuration to use when configuring the master

overrides(dict):
A dictionary of configuration overrides to use when configuring the master

max_start_attempts(int):
How many attempts should be made to start the master in case of failure to validate that its running

factory_class_kwargs(dict):
Extra keyword arguments to pass to *saltfactories.daemons.master.SaltMaster*

Returns:

***saltfactories.daemons.master.SaltMaster*:**
The master process class instance

`salt_minion_daemon`(*minion_id*, *master=None*, *defaults=None*, *overrides=None*, *max_start_attempts=3*, *start_timeout=None*, *factory_class=<class 'saltfactories.daemons.minion.SaltMinion'>*, ***factory_class_kwargs*)

Return a salt-minion instance.

Args:

minion_id(str):
The minion ID

master(*saltfactories.daemons.master.SaltMaster*):
An instance of *saltfactories.daemons.master.SaltMaster* that this minion will connect to.

defaults(dict):
A dictionary of default configuration to use when configuring the minion

overrides(dict):
A dictionary of configuration overrides to use when configuring the minion

max_start_attempts(int):
How many attempts should be made to start the minion in case of failure to validate that its running

factory_class_kwargs(dict):
Extra keyword arguments to pass to *SaltMinion*

Returns:

SaltMinion:
The minion process class instance

salt_syndic_daemon(*syndic_id*, *master_of_masters=None*, *defaults=None*, *overrides=None*, *max_start_attempts=3*, *start_timeout=None*, *factory_class=<class 'saltfactories.daemons.syndic.SaltSyndic'>*, *master_defaults=None*, *master_overrides=None*, *master_factory_class=<class 'saltfactories.daemons.master.SaltMaster'>*, *minion_defaults=None*, *minion_overrides=None*, *minion_factory_class=<class 'saltfactories.daemons.minion.SaltMinion'>*, ***factory_class_kwargs*)

Return a salt-syndic instance.

Args:

syndic_id(str):
The Syndic ID. This ID will be shared by the `salt-master`, `salt-minion` and `salt-syndic` processes.

master_of_masters(*saltfactories.daemons.master.SaltMaster*):
An instance of *saltfactories.daemons.master.SaltMaster* that the master configured in this Syndic topology scenario shall connect to.

defaults(dict):
A dictionary of default configurations with three top level keys, `master`, `minion` and `syndic`, to use when configuring the `salt-master`, `salt-minion` and `salt-syndic` respectively.

overrides(dict):
A dictionary of configuration overrides with three top level keys, `master`, `minion` and `syndic`, to use when configuring the `salt-master`, `salt-minion` and `salt-syndic` respectively.

max_start_attempts(int):
How many attempts should be made to start the syndic in case of failure to validate that its running

factory_class_kwargs(dict):
Extra keyword arguments to pass to `SaltSyndic`

Returns:

`SaltSyndic`:
The syndic process class instance

salt_proxy_minion_daemon(*proxy_minion_id*, *master=None*, *defaults=None*, *overrides=None*, *max_start_attempts=3*, *start_timeout=None*, *factory_class=<class 'saltfactories.daemons.proxy.SaltProxyMinion'>*, ***factory_class_kwargs*)

Return a salt proxy-minion instance.

Args:

proxy_minion_id(str):
The proxy minion ID

master(*saltfactories.daemons.master.SaltMaster*):
An instance of *saltfactories.daemons.master.SaltMaster* that this minion will connect to.

defaults(dict):
A dictionary of default configuration to use when configuring the proxy minion

overrides(dict):
A dictionary of configuration overrides to use when configuring the proxy minion

max_start_attempts(int):
How many attempts should be made to start the proxy minion in case of failure to validate that its running

factory_class_kwargs(dict):
Extra keyword arguments to pass to *SaltProxyMinion*

Returns:

***SaltProxyMinion*:**
The proxy minion process class instance

salt_api_daemon(*master*, *max_start_attempts=3*, *start_timeout=None*, *factory_class=<class 'saltfactories.daemons.api.SaltApi'>*, ***factory_class_kwargs*)

Return a salt-api instance.

Please see py:class:*~saltfactories.manager.FactoriesManager.salt_master_daemon* for argument documentation.

Returns:

***SaltApi*:**
The salt-api process class instance

get_sshd_daemon(*config_dir=None*, *listen_address=None*, *listen_port=None*, *sshd_config_dict=None*, *display_name=None*, *script_name='sshd'*, *max_start_attempts=3*, *start_timeout=None*, *factory_class=<class 'saltfactories.daemons.sshd.Sshd'>*, ***factory_class_kwargs*)

Return an SSHD daemon instance.

Args:

max_start_attempts(int):
How many attempts should be made to start the proxy minion in case of failure to validate that its running

config_dir(pathlib.Path):
The path to the sshd config directory

listen_address(str):
The address where the sshd server will listen to connections. Defaults to 127.0.0.1

listen_port(int):
The port where the sshd server will listen to connections

sshd_config_dict(dict):
A dictionary of key-value pairs to construct the sshd config file

script_name(str):
The name or path to the binary to run. Defaults to `sshd`.

factory_class_kwargs(dict):
Extra keyword arguments to pass to *Sshd*

Returns:

***Sshd*:**
The sshd process class instance

get_container(*container_name*, *image_name*, *display_name=None*, *factory_class=<class 'saltfactories.daemons.container.Container'>*, *max_start_attempts=3*, *start_timeout=None*, ***factory_class_kwargs*)

Return a container instance.

Args:

container_name(str):
The name to give the container

image_name(str):
The image to use

display_name(str):
Human readable name for the factory

factory_class:
A factory class. (Default *Container*)

max_start_attempts(int):
How many attempts should be made to start the container in case of failure to validate that its running.

start_timeout(int):
The amount of time, in seconds, to wait, until the container is considered as not started.

factory_class_kwargs(dict):
Extra keyword arguments to pass to *Container*

Returns:

Container:
The factory instance

`get_salt_script_path`(*script_name*)

Return the path to the customized script path, generating one if needed.

`get_root_dir_for_daemon`(*daemon_id*, *defaults=None*, *factory_class=None*)

Return a root directory for the passed daemon.

5.5.3 Daemons

salt-master

Salt Master Factory.

`class saltfactories.daemons.master.SaltMaster`(*, *config*, *config_dir=_Nothing.NOTHING*, *python_executable=None*, *system_service=False*, *cwd=_Nothing.NOTHING*, *environ=_Nothing.NOTHING*, *slow_stop=True*, *system_encoding=_Nothing.NOTHING*, *timeout=_Nothing.NOTHING*, *script_name*, *base_script_args=_Nothing.NOTHING*, *check_ports=_Nothing.NOTHING*, *stats_processes=None*, *start_timeout*, *max_start_attempts=3*, *extra_cli_arguments_after_first_start_failure=_Nothing.NOTHING*, *start_checks_callbacks=_Nothing.NOTHING*, *event_listener=None*, *factories_manager=None*, *started_at=None*, *on_auth_event_callback=None*)

Bases: *SaltDaemon*

salt-master daemon factory.

Parameters

- **cwd** (*str | Path*) –
- **environ** (*EnvironDict*) –
- **slow_stop** (*bool*) –
- **system_encoding** (*str*) –
- **timeout** (*int | float*) –
- **script_name** (*str*) –
- **base_script_args** (*List[str]*) –
- **check_ports** (*List[int]*) –
- **stats_processes** (*StatsProcesses*) –
- **start_timeout** (*int | float*) –
- **max_start_attempts** (*int*) –
- **extra_cli_arguments_after_first_start_failure** (*List[str]*) –
- **start_checks_callbacks** (*List[Callback]*) –

classmethod default_config(*root_dir*, *master_id*, *defaults=None*, *overrides=None*, *order_masters=False*, *master_of_masters=None*, *system_service=False*)

Return the default configuration.

classmethod load_config(*config_file*, *config*)

Return the loaded configuration.

get_check_events()

Return salt events to check.

Return a list of tuples in the form of *(master_id, event_tag)* check against to ensure the daemon is running

salt_master_daemon(*master_id*, ***kwargs*)

This method will configure a master under a master-of-masters.

Please see the documentation in *salt_master_daemon*

salt_minion_daemon(*minion_id*, ***kwargs*)

Please see the documentation in `configure_salt_minion`.

salt_proxy_minion_daemon(*minion_id*, ***kwargs*)

Please see the documentation in *salt_proxy_minion_daemon*.

salt_api_daemon(***kwargs*)

Please see the documentation in *salt_api_daemon*.

salt_syndic_daemon(*syndic_id*, ***kwargs*)

Please see the documentation in *salt_syndic_daemon*.

salt_cloud_cli(*defaults=None*, *overrides=None*, *factory_class=<class 'saltfactories.cli.cloud.SaltCloud'>*, ***factory_class_kwargs*)

Return a salt-cloud CLI instance.

Args:

defaults(dict):
A dictionary of default configuration to use when configuring the minion

overrides(dict):
A dictionary of configuration overrides to use when configuring the minion

Returns:

SaltCloud:
The salt-cloud CLI script process class instance

`salt_cli`(*factory_class=<class 'saltfactories.cli.salt.Salt'>*, ***factory_class_kwargs*)

Return a *salt* CLI process for this master instance.

`salt_cp_cli`(*factory_class=<class 'saltfactories.cli.cp.SaltCp'>*, ***factory_class_kwargs*)

Return a *salt-cp* CLI process for this master instance.

`salt_key_cli`(*factory_class=<class 'saltfactories.cli.key.SaltKey'>*, ***factory_class_kwargs*)

Return a *salt-key* CLI process for this master instance.

`salt_run_cli`(*factory_class=<class 'saltfactories.cli.run.SaltRun'>*, ***factory_class_kwargs*)

Return a *salt-run* CLI process for this master instance.

`salt_spm_cli`(*defaults=None*, *overrides=None*, *factory_class=<class 'saltfactories.cli.spm.Spm'>*, ***factory_class_kwargs*)

Return a *spm* CLI process for this master instance.

`salt_ssh_cli`(*factory_class=<class 'saltfactories.cli.ssh.SaltSsh'>*, *roster_file=None*, *target_host=None*, *client_key=None*, *ssh_user=None*, ***factory_class_kwargs*)

Return a *salt-ssh* CLI process for this master instance.

Args:

roster_file(str):
The roster file to use

target_host(str):
The target host address to connect to

client_key(str):
The path to the private ssh key to use to connect

ssh_user(str):
The remote username to connect as

`salt_client`(*functions_known_to_return_none=None*, *factory_class=<class 'saltfactories.client.LocalClient'>*)

Return a local salt client object.

`after_start`(*callback*, **args*, ***kwargs*)

Register a function callback to run after the daemon starts.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

`after_terminate`(*callback*, **args*, ***kwargs*)

Register a function callback to run after the daemon terminates.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

`before_start`(*callback*, **args*, ***kwargs*)

Register a function callback to run before the daemon starts.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

before_terminate(*callback*, **args*, ***kwargs*)

Register a function callback to run before the daemon terminates.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

cmdline(**args*)

Construct a list of arguments to use when starting the subprocess.

Parameters
args (*str*) – Additional arguments to use when starting the subprocess

classmethod configure(*factories_manager*, *daemon_id*, *root_dir=None*, *defaults=None*, *overrides=None*, ***configure_kwargs*)

Configure the salt daemon.

get_base_script_args()

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

get_check_ports()

Return a list of ports to check against to ensure the daemon is running.

Return type
List[int]

`get_display_name()`

Returns a human readable name for the factory.

`get_script_args()`

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

`get_script_path()`

Returns the path to the script to run.

Return type
str

`get_start_check_callbacks()`

Return a list of the start check callbacks.

Return type
List[*Callback*]

`is_running()`

Returns true if the sub-process is alive.

Return type
bool

`property pid: int | None`

The pid of the running process. None if not running.

`process_output(stdout, stderr, cmdline=None)`

Process the output. When possible JSON is loaded from the output.

Returns:
Returns a tuple in the form of `(stdout, stderr, loaded_json)`

Parameters

- **stdout** (*str*) –
- **stderr** (*str*) –
- **cmdline** (*List[str] | None*) –

Return type
Tuple[str, str, *Dict*[*Any*, *Any*] | None]

`run(*args, env=None, _timeout=None, **kwargs)`

Run the given command synchronously.

Keyword Arguments:

args:
The list of arguments to pass to `cmdline()` to construct the command to run

env:
Pass a dictionary of environment key, value pairs to inject into the subprocess.

_timeout:
The timeout value for this particular `run()` call. If this value is not `None`, it will be used instead of `timeout`, the default timeout.

Parameters

- **args** (*str*) –
- **env** (*EnvironDict | None*) –
- **_timeout** (*int | float | None*) –
- **kwargs** (*Any*) –

Return type
ProcessResult

run_start_checks(*started_at*, *timeout_at*)

Run checks to confirm that the daemon has started.

Parameters

- **started_at** (*float*) –
- **timeout_at** (*float*) –

Return type
bool

start(**extra_cli_arguments*, *max_start_attempts=None*, *start_timeout=None*)

Start the daemon.

Parameters

- **extra_cli_arguments** (*str*) –
- **max_start_attempts** (*int | None*) –
- **start_timeout** (*int | float | None*) –

Return type
bool

start_check(*callback*, **args*, ***kwargs*)

Register a function to run after the daemon starts to confirm readiness for work.

The callback must accept as the first argument `timeout_at` which is a float. The callback must stop trying to confirm running behavior once `time.time() > timeout_at`. The callback should return `True` to confirm that the daemon is ready for work.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Example:

```
def check_running_state(timeout_at: float) -> bool:
    while time.time() <= timeout_at:
        # run some checks
        ...
        # if all is good
        break
    else:
        return False
    return True
```

Parameters

- **callback** (*Callable[[...], bool]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

started(**extra_cli_arguments*, *max_start_attempts=None*, *start_timeout=None*)

Start the daemon and return it's instance so it can be used as a context manager.

Parameters

- **extra_cli_arguments** (*str*) –
- **max_start_attempts** (*int | None*) –
- **start_timeout** (*int | float | None*) –

Return type
Generator[*Daemon*, None, None]

stopped(*before_stop_callback=None*, *after_stop_callback=None*, *before_start_callback=None*, *after_start_callback=None*)

Stop the daemon and return it's instance so it can be used as a context manager.

Keyword Arguments:

before_stop_callback:
A callable to run before stopping the daemon. The callback must accept one argument, the daemon instance.

after_stop_callback:
A callable to run after stopping the daemon. The callback must accept one argument, the daemon instance.

before_start_callback:
A callable to run before starting the daemon. The callback must accept one argument, the daemon instance.

after_start_callback:
A callable to run after starting the daemon. The callback must accept one argument, the daemon instance.

This context manager will stop the factory while the context is in place, it re-starts it once out of context.

Example:

```
assert factory.is_running() is True

with factory.stopped():
    assert factory.is_running() is False

assert factory.is_running() is True
```

Parameters

- **before_stop_callback** (*Callable[[Daemon], None] | None*) –
- **after_stop_callback** (*Callable[[Daemon], None] | None*) –
- **before_start_callback** (*Callable[[Daemon], None] | None*) –
- **after_start_callback** (*Callable[[Daemon], None] | None*) –

Return type
Generator[*Daemon*, None, None]

terminate()

Terminate the started subprocess.

Return type
ProcessResult

classmethod verify_config(*config*)

Verify the configuration dictionary.

classmethod write_config(*config*)

Write the configuration to file.

salt-minion

Salt Minion Factory.

class saltfactories.daemons.minion.**SaltMinion**(*, *config*, *config_dir=_Nothing.NOTHING*, *python_executable=None*, *system_service=False*, *cwd=_Nothing.NOTHING*, *environ=_Nothing.NOTHING*, *slow_stop=True*, *system_encoding=_Nothing.NOTHING*, *timeout=_Nothing.NOTHING*, *script_name*, *base_script_args=_Nothing.NOTHING*, *check_ports=_Nothing.NOTHING*, *stats_processes=None*, *start_timeout*, *max_start_attempts=3*, *extra_cli_arguments_after_first_start_failure=_Nothing.NOTHING*, *start_checks_callbacks=_Nothing.NOTHING*, *event_listener=None*, *factories_manager=None*, *started_at=None*)

Bases: *SaltDaemon*

salt-minion daemon factory.

Parameters

- **cwd** (*str | Path*) –

- **environ** (*EnvironDict*) –
- **slow_stop** (*bool*) –
- **system_encoding** (*str*) –
- **timeout** (*int | float*) –
- **script_name** (*str*) –
- **base_script_args** (*List[str]*) –
- **check_ports** (*List[int]*) –
- **stats_processes** (*StatsProcesses*) –
- **start_timeout** (*int | float*) –
- **max_start_attempts** (*int*) –
- **extra_cli_arguments_after_first_start_failure** (*List[str]*) –
- **start_checks_callbacks** (*List[Callback]*) –

classmethod default_config(*root_dir*, *minion_id*, *defaults=None*, *overrides=None*, *master=None*, *system_service=False*)

Return the default configuration.

classmethod load_config(*config_file*, *config*)

Return the loaded configuration.

get_script_args()

Return the script arguments.

get_check_events()

Return salt events to check.

Return a list of tuples in the form of *(master_id, event_tag)* check against to ensure the daemon is running

salt_call_cli(*factory_class=<class 'saltfactories.cli.call.SaltCall'>*, ***factory_class_kwargs*)

Return a *salt-call* CLI process for this minion instance.

after_start(*callback*, **args*, ***kwargs*)

Register a function callback to run after the daemon starts.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

after_terminate(*callback*, **args*, ***kwargs*)

Register a function callback to run after the daemon terminates.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

before_start(*callback*, **args*, ***kwargs*)

Register a function callback to run before the daemon starts.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

`before_terminate`(*callback*, **args*, ***kwargs*)

Register a function callback to run before the daemon terminates.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

`cmdline`(**args*)

Construct a list of arguments to use when starting the subprocess.

Parameters
args (*str*) – Additional arguments to use when starting the subprocess

`classmethod configure`(*factories_manager*, *daemon_id*, *root_dir=None*, *defaults=None*, *overrides=None*, ***configure_kwargs*)

Configure the salt daemon.

`get_base_script_args`()

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

`get_check_ports`()

Return a list of ports to check against to ensure the daemon is running.

Return type
List[int]

`get_display_name`()

Returns a human readable name for the factory.

`get_script_path()`

Returns the path to the script to run.

Return type
str

`get_start_check_callbacks()`

Return a list of the start check callbacks.

Return type
List[*Callback*]

`is_running()`

Returns true if the sub-process is alive.

Return type
bool

`property pid: int | None`

The pid of the running process. None if not running.

`process_output(stdout, stderr, cmdline=None)`

Process the output. When possible JSON is loaded from the output.

Returns:
Returns a tuple in the form of `(stdout, stderr, loaded_json)`

Parameters

- **stdout** (*str*) –
- **stderr** (*str*) –
- **cmdline** (*List[str] | None*) –

Return type
Tuple[str, str, *Dict*[*Any*, *Any*] | None]

`run(*args, env=None, _timeout=None, **kwargs)`

Run the given command synchronously.

Keyword Arguments:

args:
The list of arguments to pass to `cmdline()` to construct the command to run

env:
Pass a dictionary of environment key, value pairs to inject into the subprocess.

_timeout:
The timeout value for this particular `run()` call. If this value is not `None`, it will be used instead of `timeout`, the default timeout.

Parameters

- **args** (*str*) –
- **env** (*EnvironDict | None*) –
- **_timeout** (*int | float | None*) –
- **kwargs** (*Any*) –

Return type
ProcessResult

`run_start_checks`(*started_at*, *timeout_at*)

Run checks to confirm that the daemon has started.

Parameters

- **started_at** (*float*) –
- **timeout_at** (*float*) –

Return type
bool

`start`(**extra_cli_arguments*, *max_start_attempts=None*, *start_timeout=None*)

Start the daemon.

Parameters

- **extra_cli_arguments** (*str*) –
- **max_start_attempts** (*int | None*) –
- **start_timeout** (*int | float | None*) –

Return type
bool

`start_check`(*callback*, **args*, ***kwargs*)

Register a function to run after the daemon starts to confirm readiness for work.

The callback must accept as the first argument `timeout_at` which is a float. The callback must stop trying to confirm running behavior once `time.time() > timeout_at`. The callback should return `True` to confirm that the daemon is ready for work.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Example:

```
def check_running_state(timeout_at: float) -> bool:
    while time.time() <= timeout_at:
        # run some checks
        ...
        # if all is good
        break
    else:
        return False
    return True
```

Parameters

- **callback** (*Callable[[...], bool]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

started(**extra_cli_arguments*, *max_start_attempts=None*, *start_timeout=None*)

Start the daemon and return it's instance so it can be used as a context manager.

Parameters

- **extra_cli_arguments** (*str*) –
- **max_start_attempts** (*int | None*) –
- **start_timeout** (*int | float | None*) –

Return type
Generator[*Daemon*, None, None]

stopped(*before_stop_callback=None*, *after_stop_callback=None*, *before_start_callback=None*, *after_start_callback=None*)

Stop the daemon and return it's instance so it can be used as a context manager.

Keyword Arguments:

before_stop_callback:
A callable to run before stopping the daemon. The callback must accept one argument, the daemon instance.

after_stop_callback:
A callable to run after stopping the daemon. The callback must accept one argument, the daemon instance.

before_start_callback:
A callable to run before starting the daemon. The callback must accept one argument, the daemon instance.

after_start_callback:
A callable to run after starting the daemon. The callback must accept one argument, the daemon instance.

This context manager will stop the factory while the context is in place, it re-starts it once out of context.

Example:

```
assert factory.is_running() is True

with factory.stopped():
    assert factory.is_running() is False

assert factory.is_running() is True
```

Parameters

- **before_stop_callback** (*Callable[[Daemon], None] | None*) –
- **after_stop_callback** (*Callable[[Daemon], None] | None*) –

- **before_start_callback** (*Callable[[Daemon], None] | None*) –
- **after_start_callback** (*Callable[[Daemon], None] | None*) –

Return type
Generator[*Daemon*, None, None]

terminate()

Terminate the started subprocess.

Return type
ProcessResult

classmethod verify_config(*config*)

Verify the configuration dictionary.

classmethod write_config(*config*)

Write the configuration to file.

salt-proxy

Salt Proxy Minion Factory.

class saltfactories.daemons.proxy.**SystemdSaltProxyImpl**(*, *factory*, *before_start_callbacks=_Nothing.NOTHING*, *after_start_callbacks=_Nothing.NOTHING*, *before_terminate_callbacks=_Nothing.NOTHING*, *after_terminate_callbacks=_Nothing.NOTHING*)

Bases: *SystemdSaltDaemonImpl*

systemd salt-proxy daemon factory.

Parameters

- **factory** (*Daemon*) –
- **before_start_callbacks** (*List[Callback]*) –
- **after_start_callbacks** (*List[Callback]*) –
- **before_terminate_callbacks** (*List[Callback]*) –
- **after_terminate_callbacks** (*List[Callback]*) –

get_service_name()

Return the systemd service name.

after_start(*callback*, **args*, ***kwargs*)

Register a function callback to run after the daemon starts.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

after_terminate(*callback*, **args*, ***kwargs*)

Register a function callback to run after the daemon terminates.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

before_start(*callback*, **args*, ***kwargs*)

Register a function callback to run before the daemon starts.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

before_terminate(*callback*, **args*, ***kwargs*)

Register a function callback to run before the daemon terminates.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

cmdline(**args*)

Construct a list of arguments to use when starting the subprocess.

Parameters
args (*str*) – Additional arguments to use when starting the subprocess

get_start_arguments()

Return the arguments and keyword arguments used when starting the daemon.

Return type
StartDaemonCallArguments

init_terminal(*cmdline*, *shell=False*, *env=None*, *cwd=None*)

Instantiate a terminal with the passed command line(`cmdline`) and return it.

Additionally, it sets a reference to it in `self._terminal` and also collects an initial listing of child processes which will be used when terminating the terminal

Arguments:

cmdline:
List of strings to pass as `args` to `Popen`

Keyword Arguments:

shell:
Pass the value of `shell` to `Popen`

env:
A dictionary of `key`, `value` pairs to add to the `pytestshellutils.shell.Factory.environ`.

cwd:
A path for the CWD when running the process.

Returns:
A `Popen` instance.

Parameters

- **cmdline** (*List[str]*) –
- **shell** (*bool*) –
- **env** (*EnvironDict | None*) –
- **cwd** (*str | Path | None*) –

Return type
Popen[*Any*]

`is_running()`
Returns true if the sub-process is alive.

`property pid`
Return the `pid` of the running process.

`run`(**args*, *shell=False*, *env=None*, *cwd=None*, ***kwargs*)
Run the given command synchronously.

Arguments:

args:
The command to run.

Keyword Arguments:

shell:
Pass the value of *shell* to `pytestshellutils.shell.Factory.init_terminal()`

env:
A dictionary of `key`, `value` pairs to add to the `pytestshellutils.shell.Factory.environ`.

cwd:
A path for the CWD when running the process.

Returns:
A `Popen` instance.

Parameters

- **args** (*str*) –
- **shell** (*bool*) –

- **env** (*EnvironDict | None*) –
- **cwd** (*str | Path | None*) –
- **kwargs** (*Any*) –

Return type
Popen[*Any*]

start(**extra_cli_arguments*, *max_start_attempts=None*, *start_timeout=None*)

Start the daemon.

terminate()

Terminate the daemon.

Return type
ProcessResult

class saltfactories.daemons.proxy.**SaltProxyMinion**(***, *config*, *config_dir=_Nothing.NOTHING*, *python_executable=None*, *system_service=False*, *cwd=_Nothing.NOTHING*, *environ=_Nothing.NOTHING*, *slow_stop=True*, *system_encoding=_Nothing.NOTHING*, *timeout=_Nothing.NOTHING*, *script_name*, *base_script_args=_Nothing.NOTHING*, *check_ports=_Nothing.NOTHING*, *stats_processes=None*, *start_timeout*, *max_start_attempts=3*, *extra_cli_arguments_after_first_start_failure=_Nothing.NOTHING*, *start_checks_callbacks=_Nothing.NOTHING*, *event_listener=None*, *factories_manager=None*, *started_at=None*, *include_proxyid_cli_flag=True*)

Bases: *SaltDaemon*

salt-proxy daemon factory.

Parameters

- **cwd** (*str | Path*) –
- **environ** (*EnvironDict*) –
- **slow_stop** (*bool*) –
- **system_encoding** (*str*) –
- **timeout** (*int | float*) –
- **script_name** (*str*) –
- **base_script_args** (*List[str]*) –
- **check_ports** (*List[int]*) –
- **stats_processes** (*StatsProcesses*) –
- **start_timeout** (*int | float*) –
- **max_start_attempts** (*int*) –
- **extra_cli_arguments_after_first_start_failure** (*List[str]*) –
- **start_checks_callbacks** (*List[Callback]*) –

classmethod default_config(*root_dir*, *proxy_minion_id*, *defaults=None*, *overrides=None*, *master=None*, *system_service=False*)

Return the default configuration.

classmethod load_config(*config_file*, *config*)

Return the loaded configuration.

get_base_script_args()

Return the base arguments for the daemon.

cmdline(**args*)

Construct a list of arguments to use when starting the daemon.

Parameters
args (*str*) – Additional arguments to use when starting the daemon

get_check_events()

Return salt events to check.

Return a list of tuples in the form of *(master_id, event_tag)* check against to ensure the daemon is running

salt_call_cli(*factory_class=<class 'saltfactories.cli.call.SaltCall'>*, ***factory_class_kwargs*)

Return a *salt-call* CLI process for this minion instance.

after_start(*callback*, **args*, ***kwargs*)

Register a function callback to run after the daemon starts.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

after_terminate(*callback*, **args*, ***kwargs*)

Register a function callback to run after the daemon terminates.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

before_start(*callback*, **args*, ***kwargs*)

Register a function callback to run before the daemon starts.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

before_terminate(*callback*, **args*, ***kwargs*)

Register a function callback to run before the daemon terminates.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

classmethod configure(*factories_manager*, *daemon_id*, *root_dir=None*, *defaults=None*, *overrides=None*, ***configure_kwargs*)

Configure the salt daemon.

get_check_ports()

Return a list of ports to check against to ensure the daemon is running.

Return type
List[int]

get_display_name()

Returns a human readable name for the factory.

get_script_args()

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

get_script_path()

Returns the path to the script to run.

Return type
str

get_start_check_callbacks()

Return a list of the start check callbacks.

Return type
List[*Callback*]

is_running()

Returns true if the sub-process is alive.

Return type
bool

property pid: int | None

The pid of the running process. None if not running.

process_output(*stdout*, *stderr*, *cmdline=None*)

Process the output. When possible JSON is loaded from the output.

Returns:
Returns a tuple in the form of `(stdout, stderr, loaded_json)`

Parameters

- **stdout** (*str*) –
- **stderr** (*str*) –
- **cmdline** (*List[str] | None*) –

Return type
Tuple[str, str, *Dict*[*Any*, *Any*] | None]

run(**args*, *env=None*, *_timeout=None*, ***kwargs*)

Run the given command synchronously.

Keyword Arguments:

args:
The list of arguments to pass to `cmdline()` to construct the command to run

env:
Pass a dictionary of environment key, value pairs to inject into the subprocess.

_timeout:
The timeout value for this particular `run()` call. If this value is not `None`, it will be used instead of `timeout`, the default timeout.

Parameters

- **args** (*str*) –
- **env** (*EnvironDict | None*) –
- **_timeout** (*int | float | None*) –
- **kwargs** (*Any*) –

Return type
ProcessResult

run_start_checks(*started_at*, *timeout_at*)

Run checks to confirm that the daemon has started.

Parameters

- **started_at** (*float*) –
- **timeout_at** (*float*) –

Return type
bool

start(**extra_cli_arguments*, *max_start_attempts=None*, *start_timeout=None*)

Start the daemon.

Parameters

- **extra_cli_arguments** (*str*) –
- **max_start_attempts** (*int | None*) –
- **start_timeout** (*int | float | None*) –

Return type
bool

start_check(*callback*, **args*, ***kwargs*)

Register a function to run after the daemon starts to confirm readiness for work.

The callback must accept as the first argument `timeout_at` which is a float. The callback must stop trying to confirm running behavior once `time.time() > timeout_at`. The callback should return `True` to confirm that the daemon is ready for work.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Example:

```
def check_running_state(timeout_at: float) -> bool:
    while time.time() <= timeout_at:
        # run some checks
        ...
        # if all is good
        break
    else:
        return False
    return True
```

Parameters

- **callback** (*Callable[[...], bool]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

started(**extra_cli_arguments*, *max_start_attempts=None*, *start_timeout=None*)

Start the daemon and return it's instance so it can be used as a context manager.

Parameters

- **extra_cli_arguments** (*str*) –
- **max_start_attempts** (*int | None*) –
- **start_timeout** (*int | float | None*) –

Return type
Generator[*Daemon*, None, None]

stopped(*before_stop_callback=None*, *after_stop_callback=None*, *before_start_callback=None*, *after_start_callback=None*)

Stop the daemon and return it's instance so it can be used as a context manager.

Keyword Arguments:

before_stop_callback:
A callable to run before stopping the daemon. The callback must accept one argument, the daemon instance.

after_stop_callback:
A callable to run after stopping the daemon. The callback must accept one argument, the daemon instance.

before_start_callback:
A callable to run before starting the daemon. The callback must accept one argument, the daemon instance.

after_start_callback:
A callable to run after starting the daemon. The callback must accept one argument, the daemon instance.

This context manager will stop the factory while the context is in place, it re-starts it once out of context.

Example:

```
assert factory.is_running() is True

with factory.stopped():
    assert factory.is_running() is False

assert factory.is_running() is True
```

Parameters

- **before_stop_callback** (*Callable[[Daemon], None] | None*) –
- **after_stop_callback** (*Callable[[Daemon], None] | None*) –
- **before_start_callback** (*Callable[[Daemon], None] | None*) –
- **after_start_callback** (*Callable[[Daemon], None] | None*) –

Return type
Generator[*Daemon*, None, None]

terminate()

Terminate the started subprocess.

Return type
ProcessResult

classmethod verify_config(*config*)

Verify the configuration dictionary.

classmethod write_config(*config*)

Write the configuration to file.

salt-api

Salt API Factory.

class saltfactories.daemons.api.**SaltApi**(*, *config*, *config_dir=_Nothing.NOTHING*, *python_executable=None*, *system_service=False*, *cwd=_Nothing.NOTHING*, *environ=_Nothing.NOTHING*, *slow_stop=True*, *system_encoding=_Nothing.NOTHING*, *timeout=_Nothing.NOTHING*, *script_name*, *base_script_args=_Nothing.NOTHING*, *check_ports=_Nothing.NOTHING*, *stats_processes=None*, *start_timeout*, *max_start_attempts=3*, *extra_cli_arguments_after_first_start_failure=_Nothing.NOTHING*, *start_checks_callbacks=_Nothing.NOTHING*, *event_listener=None*, *factories_manager=None*, *started_at=None*)

Bases: *SaltDaemon*

salt-api daemon factory.

Parameters

- **cwd** (*str | Path*) –
- **environ** (*EnvironDict*) –
- **slow_stop** (*bool*) –
- **system_encoding** (*str*) –
- **timeout** (*int | float*) –
- **script_name** (*str*) –
- **base_script_args** (*List[str]*) –
- **check_ports** (*List[int]*) –
- **stats_processes** (*StatsProcesses*) –
- **start_timeout** (*int | float*) –
- **max_start_attempts** (*int*) –
- **extra_cli_arguments_after_first_start_failure** (*List[str]*) –
- **start_checks_callbacks** (*List[Callback]*) –

classmethod load_config(*config_file*, *config*)

Return the loaded configuration.

get_check_events()

Return salt events to check.

Return a list of tuples in the form of *(master_id, event_tag)* check against to ensure the daemon is running

after_start(*callback*, **args*, ***kwargs*)

Register a function callback to run after the daemon starts.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

`after_terminate`(*callback*, **args*, ***kwargs*)

Register a function callback to run after the daemon terminates.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

`before_start`(*callback*, **args*, ***kwargs*)

Register a function callback to run before the daemon starts.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

before_terminate(*callback*, **args*, ***kwargs*)

Register a function callback to run before the daemon terminates.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

cmdline(**args*)

Construct a list of arguments to use when starting the subprocess.

Parameters
args (*str*) – Additional arguments to use when starting the subprocess

classmethod configure(*factories_manager*, *daemon_id*, *root_dir=None*, *defaults=None*, *overrides=None*, ***configure_kwargs*)

Configure the salt daemon.

get_base_script_args()

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

`get_check_ports`()

Return a list of ports to check against to ensure the daemon is running.

Return type
List[int]

`get_display_name`()

Returns a human readable name for the factory.

`get_script_args`()

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

`get_script_path`()

Returns the path to the script to run.

Return type
str

`get_start_check_callbacks`()

Return a list of the start check callbacks.

Return type
List[*Callback*]

`is_running`()

Returns true if the sub-process is alive.

Return type
bool

`property pid: int | None`

The pid of the running process. None if not running.

`process_output`(*stdout*, *stderr*, *cmdline=None*)

Process the output. When possible JSON is loaded from the output.

Returns:
Returns a tuple in the form of `(stdout, stderr, loaded_json)`

Parameters

- **stdout** (*str*) –
- **stderr** (*str*) –
- **cmdline** (*List[str] | None*) –

Return type
Tuple[str, str, *Dict*[*Any*, *Any*] | None]

`run`(**args*, *env=None*, *_timeout=None*, ***kwargs*)

Run the given command synchronously.

Keyword Arguments:

args:
The list of arguments to pass to `cmdline()` to construct the command to run

env:
Pass a dictionary of environment key, value pairs to inject into the subprocess.

_timeout:
The timeout value for this particular `run()` call. If this value is not `None`, it will be used instead of `timeout`, the default timeout.

Parameters

- **args** (*str*) –
- **env** (*EnvironDict | None*) –
- **_timeout** (*int | float | None*) –
- **kwargs** (*Any*) –

Return type
ProcessResult

run_start_checks(*started_at*, *timeout_at*)

Run checks to confirm that the daemon has started.

Parameters

- **started_at** (*float*) –
- **timeout_at** (*float*) –

Return type
bool

start(**extra_cli_arguments*, *max_start_attempts=None*, *start_timeout=None*)

Start the daemon.

Parameters

- **extra_cli_arguments** (*str*) –
- **max_start_attempts** (*int | None*) –
- **start_timeout** (*int | float | None*) –

Return type
bool

start_check(*callback*, **args*, ***kwargs*)

Register a function to run after the daemon starts to confirm readiness for work.

The callback must accept as the first argument `timeout_at` which is a float. The callback must stop trying to confirm running behavior once `time.time() > timeout_at`. The callback should return `True` to confirm that the daemon is ready for work.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Example:

```
def check_running_state(timeout_at: float) -> bool:
    while time.time() <= timeout_at:
        # run some checks
        ...
        # if all is good
        break
    else:
        return False
    return True
```

Parameters

- **callback** (*Callable[[...], bool]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

started(**extra_cli_arguments*, *max_start_attempts=None*, *start_timeout=None*)

Start the daemon and return it's instance so it can be used as a context manager.

Parameters

- **extra_cli_arguments** (*str*) –
- **max_start_attempts** (*int | None*) –
- **start_timeout** (*int | float | None*) –

Return type
Generator[*Daemon*, None, None]

stopped(*before_stop_callback=None*, *after_stop_callback=None*, *before_start_callback=None*, *after_start_callback=None*)

Stop the daemon and return it's instance so it can be used as a context manager.

Keyword Arguments:

before_stop_callback:
A callable to run before stopping the daemon. The callback must accept one argument, the daemon instance.

after_stop_callback:
A callable to run after stopping the daemon. The callback must accept one argument, the daemon instance.

before_start_callback:
A callable to run before starting the daemon. The callback must accept one argument, the daemon instance.

after_start_callback:
A callable to run after starting the daemon. The callback must accept one argument, the daemon instance.

This context manager will stop the factory while the context is in place, it re-starts it once out of context.

Example:

```
assert factory.is_running() is True

with factory.stopped():
    assert factory.is_running() is False

assert factory.is_running() is True
```

Parameters

- **before_stop_callback** (*Callable[[Daemon], None] | None*) –
- **after_stop_callback** (*Callable[[Daemon], None] | None*) –
- **before_start_callback** (*Callable[[Daemon], None] | None*) –
- **after_start_callback** (*Callable[[Daemon], None] | None*) –

Return type
Generator[*Daemon*, None, None]

terminate()

Terminate the started subprocess.

Return type
ProcessResult

classmethod verify_config(*config*)

Verify the configuration dictionary.

classmethod write_config(*config*)

Write the configuration to file.

sshd

SSHD daemon factory implementation.

class saltfactories.daemons.sshd.**Sshd**(*, *cwd=_Nothing.NOTHING*, *environ=_Nothing.NOTHING*, *slow_stop=True*, *system_encoding=_Nothing.NOTHING*, *timeout=_Nothing.NOTHING*, *script_name*, *base_script_args=_Nothing.NOTHING*, *check_ports=_Nothing.NOTHING*, *stats_processes=None*, *start_timeout*, *max_start_attempts=3*, *extra_cli_arguments_after_first_start_failure=_Nothing.NOTHING*, *start_checks_callbacks=_Nothing.NOTHING*, *config_dir*, *listen_address=None*, *listen_port=None*, *authorized_keys=None*, *sshd_config_dict=None*, *display_name=None*, *ssh_keygen_path='/usr/bin/ssh-keygen'*, *ssh_keyscan_path='/usr/bin/ssh-keyscan'*)

Bases: Daemon

SSHD implementation.

Parameters

- **cwd** (*str | Path*) –
- **environ** (*EnvironDict*) –
- **slow_stop** (*bool*) –
- **system_encoding** (*str*) –
- **timeout** (*int | float*) –
- **script_name** (*str*) –
- **base_script_args** (*List[str]*) –
- **check_ports** (*List[int]*) –
- **stats_processes** (*StatsProcesses*) –
- **start_timeout** (*int | float*) –
- **max_start_attempts** (*int*) –
- **extra_cli_arguments_after_first_start_failure** (*List[str]*) –
- **start_checks_callbacks** (*List[Callback]*) –

get_display_name()

Returns a human readable name for the factory.

get_base_script_args()

Returns any additional arguments to pass to the CLI script.

get_host_keys()

Return a list of strings which are the SSHD server host keys.

after_start(*callback*, **args*, ***kwargs*)

Register a function callback to run after the daemon starts.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

`after_terminate`(*callback*, **args*, ***kwargs*)

Register a function callback to run after the daemon terminates.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

`before_start`(*callback*, **args*, ***kwargs*)

Register a function callback to run before the daemon starts.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

`before_terminate`(*callback*, **args*, ***kwargs*)

Register a function callback to run before the daemon terminates.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Parameters

- **callback** (*Callable[[], None]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

cmdline(**args*)

Construct a list of arguments to use when starting the subprocess.

Arguments:

args:
Additional arguments to use when starting the subprocess

Parameters
args (*str*) –

Return type
List[str]

get_check_ports()

Return a list of ports to check against to ensure the daemon is running.

Return type
List[int]

get_script_args()

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

get_script_path()

Returns the path to the script to run.

Return type
str

get_start_check_callbacks()

Return a list of the start check callbacks.

Return type
List[*Callback*]

`is_running`()

Returns true if the sub-process is alive.

Return type
bool

`property pid: int | None`

The pid of the running process. None if not running.

`process_output`(*stdout*, *stderr*, *cmdline=None*)

Process the output. When possible JSON is loaded from the output.

Returns:
Returns a tuple in the form of `(stdout, stderr, loaded_json)`

Parameters

- **stdout** (*str*) –
- **stderr** (*str*) –
- **cmdline** (*List[str] | None*) –

Return type
Tuple[str, str, *Dict*[*Any*, *Any*] | None]

`run`(**args*, *env=None*, *_timeout=None*, ***kwargs*)

Run the given command synchronously.

Keyword Arguments:

args:
The list of arguments to pass to `cmdline()` to construct the command to run

env:
Pass a dictionary of environment key, value pairs to inject into the subprocess.

_timeout:
The timeout value for this particular `run()` call. If this value is not `None`, it will be used instead of `timeout`, the default timeout.

Parameters

- **args** (*str*) –
- **env** (*EnvironDict | None*) –
- **_timeout** (*int | float | None*) –
- **kwargs** (*Any*) –

Return type
ProcessResult

`run_start_checks`(*started_at*, *timeout_at*)

Run checks to confirm that the daemon has started.

Parameters

- **started_at** (*float*) –
- **timeout_at** (*float*) –

Return type
bool

start(**extra_cli_arguments*, *max_start_attempts=None*, *start_timeout=None*)

Start the daemon.

Parameters

- **extra_cli_arguments** (*str*) –
- **max_start_attempts** (*int | None*) –
- **start_timeout** (*int | float | None*) –

Return type
bool

start_check(*callback*, **args*, ***kwargs*)

Register a function to run after the daemon starts to confirm readiness for work.

The callback must accept as the first argument `timeout_at` which is a float. The callback must stop trying to confirm running behavior once `time.time() > timeout_at`. The callback should return `True` to confirm that the daemon is ready for work.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Example:

```
def check_running_state(timeout_at: float) -> bool:
    while time.time() <= timeout_at:
        # run some checks
        ...
        # if all is good
        break
    else:
        return False
    return True
```

Parameters

- **callback** (*Callable[[...], bool]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

started(**extra_cli_arguments*, *max_start_attempts=None*, *start_timeout=None*)

Start the daemon and return it's instance so it can be used as a context manager.

Parameters

- **extra_cli_arguments** (*str*) –
- **max_start_attempts** (*int | None*) –
- **start_timeout** (*int | float | None*) –

Return type

Generator[*Daemon*, None, None]

stopped(*before_stop_callback=None*, *after_stop_callback=None*, *before_start_callback=None*, *after_start_callback=None*)

Stop the daemon and return it's instance so it can be used as a context manager.

Keyword Arguments:

before_stop_callback:
A callable to run before stopping the daemon. The callback must accept one argument, the daemon instance.

after_stop_callback:
A callable to run after stopping the daemon. The callback must accept one argument, the daemon instance.

before_start_callback:
A callable to run before starting the daemon. The callback must accept one argument, the daemon instance.

after_start_callback:
A callable to run after starting the daemon. The callback must accept one argument, the daemon instance.

This context manager will stop the factory while the context is in place, it re-starts it once out of context.

Example:

```
assert factory.is_running() is True

with factory.stopped():
    assert factory.is_running() is False

assert factory.is_running() is True
```

Parameters

- **before_stop_callback** (*Callable[[Daemon], None] | None*) –
- **after_stop_callback** (*Callable[[Daemon], None] | None*) –
- **before_start_callback** (*Callable[[Daemon], None] | None*) –
- **after_start_callback** (*Callable[[Daemon], None] | None*) –

Return type

Generator[*Daemon*, None, None]

terminate()

Terminate the started subprocess.

Return type
ProcessResult

Containers

Container based factories.

exception saltfactories.daemons.container.**PyWinTypesError**

Bases: Exception

Define PyWinTypesError to avoid NameError.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class saltfactories.daemons.container.**Container**(*, *cwd=_Nothing.NOTHING*, *environ=_Nothing.NOTHING*, *image*, *name=_Nothing.NOTHING*, *display_name=None*, *check_ports=_Nothing.NOTHING*, *container_run_kwargs=_Nothing.NOTHING*, *start_timeout=30*, *max_start_attempts=3*, *pull_before_start=True*, *skip_on_pull_failure=False*, *skip_if_docker_client_not_connectable=False*, *docker_client=_Nothing.NOTHING*, *before_start_callbacks=_Nothing.NOTHING*, *before_terminate_callbacks=_Nothing.NOTHING*, *after_start_callbacks=_Nothing.NOTHING*, *after_terminate_callbacks=_Nothing.NOTHING*, *container_start_checks_callbacks=_Nothing.NOTHING*)

Bases: BaseFactory

Docker containers daemon implementation.

Args:

param str image
The container image to use, for example 'centos:7'

param str name
The name to give to the started container.

keyword dict check_ports
This dictionary is a mapping where the keys are the container port bindings and the values are the host port bindings.

If this mapping is empty, the container class will inspect the `container_run_kwargs` for a `ports` key to build this mapping.

Take as an example the following `container_run_kwargs`:

```
container_run_kwargs = {
    "ports": {
        "5000/tcp": None,
        "12345/tcp": 54321,
```

(continues on next page)

(continued from previous page)

```
    }
}
```

This would build the following check ports mapping:

```
{5000: None, 12345: 54321}
```

At runtime, the *Container* class would query docker for the host port binding to the container port binding of 5000.

keyword dict container_run_kwargs
This mapping will be passed directly to the python docker library:

```
container = self.docker_client.containers.run(
    self.image,
    name=self.name,
    detach=True,
    stdin_open=True,
    command=list(command) or None,
    **self.container_run_kwargs,
)
```

keyword int start_timeout
The maximum number of seconds we should wait until the container is running.

keyword int max_start_attempts
The maximum number of attempts to try and start the container

keyword bool pull_before_start
When `True`, the image is pulled before trying to start it

keyword bool skip_on_pull_failure
When `True`, and there's a failure when pulling the image, the test is skipped.

keyword bool skip_if_docker_client_not_connectable
When `True`, it skips the test if there's a failure when connecting to docker

keyword Docker docker_client
An instance of the python docker client to use. When nothing is passed, a default docker client is instantiated.

Parameters

- **cwd** (*Path*) –
- **environ** (*EnvironDict*) –

before_start(*callback*, **args*, ***kwargs*)

Register a function callback to run before the container starts.

Parameters

- **callback** (*Callable*) – The function to call back
- **args** – The arguments to pass to the callback
- **kwargs** – The keyword arguments to pass to the callback

after_start(*callback*, **args*, ***kwargs*)

Register a function callback to run after the container starts.

Parameters

- **callback** (*Callable*) – The function to call back
- **args** – The arguments to pass to the callback
- **kwargs** – The keyword arguments to pass to the callback

before_terminate(*callback*, **args*, ***kwargs*)

Register a function callback to run before the container terminates.

Parameters

- **callback** (*Callable*) – The function to call back
- **args** – The arguments to pass to the callback
- **kwargs** – The keyword arguments to pass to the callback

after_terminate(*callback*, **args*, ***kwargs*)

Register a function callback to run after the container terminates.

Parameters

- **callback** (*Callable*) – The function to call back
- **args** – The arguments to pass to the callback
- **kwargs** – The keyword arguments to pass to the callback

container_start_check(*callback*, **args*, ***kwargs*)

Register a function to run after the container starts to confirm readiness for work.

The callback must accept as the first argument `timeout_at` which is a float. The callback must stop trying to confirm running behavior once `time.time() > timeout_at`. The callback should return `True` to confirm that the daemon is ready for work.

For example:

```
def check_running_state(timeout_at: float) -> bool:
    while time.time() <= timeout_at:
        # run some checks
        ...
        # if all is good
        break
    else:
        return False
    return True
```

Parameters

- **callback** (*Callable*) – The function to call back
- **args** – The arguments to pass to the callback
- **kwargs** – The keyword arguments to pass to the callback

get_display_name()

Returns a human readable name for the factory.

start(**command*, *max_start_attempts=None*, *start_timeout=None*)

Start the container.

started(**command*, *max_start_attempts=None*, *start_timeout=None*)

Start the container and return it's instance so it can be used as a context manager.

terminate()

Terminate the container.

get_check_ports()

Return a list of TCP ports to check against to ensure the daemon is running.

get_host_port_binding(*port*, *protocol='tcp'*, *ipv6=False*)

Return the host binding for a port on the container.

Parameters

- **port** (*int*) – The port.
- **protocol** (*str*) – The port protocol. Defaults to `tcp`.
- **ipv6** (*bool*) – If true, return the ipv6 port binding.

Returns

int: The matched port binding on the host. None: When not port binding was matched.

get_container_start_check_callbacks()

Return a list of the start check callbacks.

is_running()

Returns true if the container is running.

run(**cmd*, ***kwargs*)

Run a command inside the container.

static client_connectable(*docker_client*)

Check if the docker client can connect to the docker daemon.

run_container_start_checks(*started_at*, *timeout_at*)

Run startup checks.

class saltfactories.daemons.container.**SaltDaemon**(*, *cwd=_Nothing.NOTHING*, *environ=_Nothing.NOTHING*, *image*, *name=_Nothing.NOTHING*, *display_name=None*, *check_ports=_Nothing.NOTHING*, *container_run_kwargs=_Nothing.NOTHING*, *start_timeout=30*, *max_start_attempts=3*, *pull_before_start=True*, *skip_on_pull_failure=False*, *skip_if_docker_client_not_connectable=False*, *docker_client=_Nothing.NOTHING*, *before_start_callbacks=_Nothing.NOTHING*, *before_terminate_callbacks=_Nothing.NOTHING*, *after_start_callbacks=_Nothing.NOTHING*, *after_terminate_callbacks=_Nothing.NOTHING*, *container_start_checks_callbacks=_Nothing.NOTHING*, *config*, *config_dir=_Nothing.NOTHING*, *python_executable=None*, *system_service=False*, *slow_stop=True*, *system_encoding=_Nothing.NOTHING*, *timeout=_Nothing.NOTHING*, *script_name*, *base_script_args=_Nothing.NOTHING*, *stats_processes=None*, *extra_cli_arguments_after_first_start_failure=_Nothing.NOTHING*, *start_checks_callbacks=_Nothing.NOTHING*, *event_listener=None*, *factories_manager=None*, *started_at=None*)

Bases: *Container*, *SaltDaemon*

Salt Daemon inside a container implementation.

Parameters

- **cwd** (*str | Path*) –
- **environ** (*EnvironDict*) –
- **slow_stop** (*bool*) –
- **system_encoding** (*str*) –
- **timeout** (*int | float*) –
- **script_name** (*str*) –
- **base_script_args** (*List[str]*) –
- **stats_processes** (*StatsProcesses*) –
- **extra_cli_arguments_after_first_start_failure** (*List[str]*) –
- **start_checks_callbacks** (*List[Callback]*) –

get_display_name()

Returns a human readable name for the factory.

run(**cmd*, ***kwargs*)

Run a command inside the container.

cmdline(**args*)

Construct a list of arguments to use when starting the container.

Parameters
args (*str*) – Additional arguments to use when starting the container

start(**extra_cli_arguments*, *max_start_attempts=None*, *start_timeout=None*)
Start the daemon.

terminate()
Terminate the container.

is_running()
Returns true if the container is running.

get_check_ports()
Return a list of ports to check against to ensure the daemon is running.

before_start(*callback*, **args*, *on_container=False*, ***kwargs*)
Register a function callback to run before the daemon starts.

Parameters
- **callback** (*Callable*) – The function to call back
- **on_container** (*bool*) – If true, the callback will be registered on the container and not the daemon.
- **args** – The arguments to pass to the callback
- **kwargs** – The keyword arguments to pass to the callback

after_start(*callback*, **args*, *on_container=False*, ***kwargs*)
Register a function callback to run after the daemon starts.

Parameters
- **callback** (*Callable*) – The function to call back
- **on_container** (*bool*) – If true, the callback will be registered on the container and not the daemon.
- **args** – The arguments to pass to the callback
- **kwargs** – The keyword arguments to pass to the callback

before_terminate(*callback*, **args*, *on_container=False*, ***kwargs*)
Register a function callback to run before the daemon terminates.

Parameters
- **callback** (*Callable*) – The function to call back
- **on_container** (*bool*) – If true, the callback will be registered on the container and not the daemon.
- **args** – The arguments to pass to the callback
- **kwargs** – The keyword arguments to pass to the callback

after_terminate(*callback*, **args*, *on_container=False*, ***kwargs*)
Register a function callback to run after the daemon terminates.

Parameters
- **callback** (*Callable*) – The function to call back

- **on_container** (*bool*) – If true, the callback will be registered on the container and not the daemon.
- **args** – The arguments to pass to the callback
- **kwargs** – The keyword arguments to pass to the callback

started(**extra_cli_arguments*, *max_start_attempts=None*, *start_timeout=None*)

Start the daemon and return it's instance so it can be used as a context manager.

get_check_events()

Return salt events to check.

Return a list of tuples in the form of *(master_id, event_tag)* check against to ensure the daemon is running

static client_connectable(*docker_client*)

Check if the docker client can connect to the docker daemon.

classmethod configure(*factories_manager*, *daemon_id*, *root_dir=None*, *defaults=None*, *overrides=None*, ***configure_kwargs*)

Configure the salt daemon.

container_start_check(*callback*, **args*, ***kwargs*)

Register a function to run after the container starts to confirm readiness for work.

The callback must accept as the first argument `timeout_at` which is a float. The callback must stop trying to confirm running behavior once `time.time() > timeout_at`. The callback should return `True` to confirm that the daemon is ready for work.

For example:

```
def check_running_state(timeout_at: float) -> bool:
    while time.time() <= timeout_at:
        # run some checks
        ...
        # if all is good
        break
    else:
        return False
    return True
```

Parameters

- **callback** (*Callable*) – The function to call back
- **args** – The arguments to pass to the callback
- **kwargs** – The keyword arguments to pass to the callback

get_base_script_args()

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

get_container_start_check_callbacks()

Return a list of the start check callbacks.

get_host_port_binding(*port*, *protocol='tcp'*, *ipv6=False*)

Return the host binding for a port on the container.

Parameters

- **port** (*int*) – The port.
- **protocol** (*str*) – The port protocol. Defaults to `tcp`.
- **ipv6** (*bool*) – If true, return the ipv6 port binding.

Returns
int: The matched port binding on the host. None: When not port binding was matched.

get_script_args()

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

get_script_path()

Returns the path to the script to run.

Return type
str

get_start_check_callbacks()

Return a list of the start check callbacks.

Return type
List[*Callback*]

classmethod load_config(*config_file*, *config*)

Return the loaded configuration.

Should return the configuration as the daemon would have loaded after parsing the CLI

property pid: int | None

The pid of the running process. None if not running.

process_output(*stdout*, *stderr*, *cmdline=None*)

Process the output. When possible JSON is loaded from the output.

Returns:
Returns a tuple in the form of `(stdout, stderr, loaded_json)`

Parameters

- **stdout** (*str*) –
- **stderr** (*str*) –
- **cmdline** (*List[str] | None*) –

Return type
Tuple[str, str, *Dict*[*Any*, *Any*] | None]

run_container_start_checks(*started_at*, *timeout_at*)

Run startup checks.

run_start_checks(*started_at*, *timeout_at*)

Run checks to confirm that the daemon has started.

Parameters

- **started_at** (*float*) –
- **timeout_at** (*float*) –

Return type
bool

start_check(*callback*, **args*, ***kwargs*)

Register a function to run after the daemon starts to confirm readiness for work.

The callback must accept as the first argument `timeout_at` which is a float. The callback must stop trying to confirm running behavior once `time.time() > timeout_at`. The callback should return `True` to confirm that the daemon is ready for work.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Example:

```
def check_running_state(timeout_at: float) -> bool:
    while time.time() <= timeout_at:
        # run some checks
        ...
        # if all is good
        break
    else:
        return False
    return True
```

Parameters

- **callback** (*Callable[[...], bool]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

stopped(*before_stop_callback=None*, *after_stop_callback=None*, *before_start_callback=None*, *after_start_callback=None*)

Stop the daemon and return it's instance so it can be used as a context manager.

Keyword Arguments:

before_stop_callback:
A callable to run before stopping the daemon. The callback must accept one argument, the daemon instance.

after_stop_callback:
A callable to run after stopping the daemon. The callback must accept one argument, the daemon instance.

before_start_callback:
A callable to run before starting the daemon. The callback must accept one argument, the daemon instance.

after_start_callback:
A callable to run after starting the daemon. The callback must accept one argument, the daemon instance.

This context manager will stop the factory while the context is in place, it re-starts it once out of context.

Example:

```
assert factory.is_running() is True

with factory.stopped():
    assert factory.is_running() is False

assert factory.is_running() is True
```

Parameters

- **before_stop_callback** (*Callable[[Daemon], None] | None*) –
- **after_stop_callback** (*Callable[[Daemon], None] | None*) –
- **before_start_callback** (*Callable[[Daemon], None] | None*) –
- **after_start_callback** (*Callable[[Daemon], None] | None*) –

Return type
Generator[*Daemon*, None, None]

classmethod verify_config(*config*)
Verify the configuration dictionary.

classmethod write_config(*config*)
Write the configuration to file.

class saltfactories.daemons.container.**SaltMinion**(*, *cwd=_Nothing.NOTHING*, *environ=_Nothing.NOTHING*, *image*, *name=_Nothing.NOTHING*, *display_name=None*, *check_ports=_Nothing.NOTHING*, *container_run_kwargs=_Nothing.NOTHING*, *start_timeout=30*, *max_start_attempts=3*, *pull_before_start=True*, *skip_on_pull_failure=False*, *skip_if_docker_client_not_connectable=False*, *docker_client=_Nothing.NOTHING*, *before_start_callbacks=_Nothing.NOTHING*, *before_terminate_callbacks=_Nothing.NOTHING*, *after_start_callbacks=_Nothing.NOTHING*, *after_terminate_callbacks=_Nothing.NOTHING*, *container_start_checks_callbacks=_Nothing.NOTHING*, *config*, *config_dir=_Nothing.NOTHING*, *python_executable=None*, *system_service=False*, *slow_stop=True*, *system_encoding=_Nothing.NOTHING*, *timeout=_Nothing.NOTHING*, *script_name*, *base_script_args=_Nothing.NOTHING*, *stats_processes=None*, *extra_cli_arguments_after_first_start_failure=_Nothing.NOTHING*, *start_checks_callbacks=_Nothing.NOTHING*, *event_listener=None*, *factories_manager=None*, *started_at=None*)

Bases: *SaltDaemon*, *SaltMinion*

Salt minion daemon implementation running in a docker container.

Parameters

- **cwd** (*str | Path*) –
- **environ** (*EnvironDict*) –
- **slow_stop** (*bool*) –
- **system_encoding** (*str*) –
- **timeout** (*int | float*) –
- **script_name** (*str*) –
- **base_script_args** (*List[str]*) –
- **stats_processes** (*StatsProcesses*) –
- **extra_cli_arguments_after_first_start_failure** (*List[str]*) –
- **start_checks_callbacks** (*List[Callback]*) –

get_display_name()

Returns a human readable name for the factory.

get_check_events()

Return salt events to check.

Return a list of tuples in the form of *(master_id, event_tag)* check against to ensure the daemon is running

after_start(*callback*, **args*, *on_container=False*, ***kwargs*)

Register a function callback to run after the daemon starts.

Parameters

- **callback** (*Callable*) – The function to call back
- **on_container** (*bool*) – If true, the callback will be registered on the container and not the daemon.
- **args** – The arguments to pass to the callback
- **kwargs** – The keyword arguments to pass to the callback

after_terminate(*callback*, **args*, *on_container=False*, ***kwargs*)

Register a function callback to run after the daemon terminates.

Parameters

- **callback** (*Callable*) – The function to call back
- **on_container** (*bool*) – If true, the callback will be registered on the container and not the daemon.
- **args** – The arguments to pass to the callback
- **kwargs** – The keyword arguments to pass to the callback

before_start(*callback*, **args*, *on_container=False*, ***kwargs*)

Register a function callback to run before the daemon starts.

Parameters

- **callback** (*Callable*) – The function to call back
- **on_container** (*bool*) – If true, the callback will be registered on the container and not the daemon.
- **args** – The arguments to pass to the callback
- **kwargs** – The keyword arguments to pass to the callback

before_terminate(*callback*, **args*, *on_container=False*, ***kwargs*)

Register a function callback to run before the daemon terminates.

Parameters

- **callback** (*Callable*) – The function to call back
- **on_container** (*bool*) – If true, the callback will be registered on the container and not the daemon.
- **args** – The arguments to pass to the callback
- **kwargs** – The keyword arguments to pass to the callback

static client_connectable(*docker_client*)

Check if the docker client can connect to the docker daemon.

cmdline(**args*)

Construct a list of arguments to use when starting the container.

Parameters

args (*str*) – Additional arguments to use when starting the container

classmethod configure(*factories_manager*, *daemon_id*, *root_dir=None*, *defaults=None*, *overrides=None*, ***configure_kwargs*)

Configure the salt daemon.

container_start_check(*callback*, **args*, ***kwargs*)

Register a function to run after the container starts to confirm readiness for work.

The callback must accept as the first argument `timeout_at` which is a float. The callback must stop trying to confirm running behavior once `time.time() > timeout_at`. The callback should return `True` to confirm that the daemon is ready for work.

For example:

```
def check_running_state(timeout_at: float) -> bool:
    while time.time() <= timeout_at:
        # run some checks
        ...
        # if all is good
        break
    else:
        return False
    return True
```

Parameters

- **callback** (*Callable*) – The function to call back
- **args** – The arguments to pass to the callback
- **kwargs** – The keyword arguments to pass to the callback

classmethod default_config(*root_dir*, *minion_id*, *defaults=None*, *overrides=None*, *master=None*, *system_service=False*)

Return the default configuration.

get_base_script_args()

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

get_check_ports()

Return a list of ports to check against to ensure the daemon is running.

get_container_start_check_callbacks()

Return a list of the start check callbacks.

get_host_port_binding(*port*, *protocol='tcp'*, *ipv6=False*)

Return the host binding for a port on the container.

Parameters

- **port** (*int*) – The port.
- **protocol** (*str*) – The port protocol. Defaults to `tcp`.
- **ipv6** (*bool*) – If true, return the ipv6 port binding.

Returns
int: The matched port binding on the host. None: When not port binding was matched.

get_script_args()

Return the script arguments.

get_script_path()

Returns the path to the script to run.

Return type
str

get_start_check_callbacks()

Return a list of the start check callbacks.

Return type
List[*Callback*]

is_running()

Returns true if the container is running.

classmethod load_config(*config_file*, *config*)

Return the loaded configuration.

property pid: int | None

The pid of the running process. None if not running.

process_output(*stdout*, *stderr*, *cmdline=None*)

Process the output. When possible JSON is loaded from the output.

Returns:
Returns a tuple in the form of `(stdout, stderr, loaded_json)`

Parameters

- **stdout** (*str*) –
- **stderr** (*str*) –
- **cmdline** (*List[str] | None*) –

Return type
Tuple[str, str, *Dict*[*Any*, *Any*] | None]

run(**cmd*, ***kwargs*)

Run a command inside the container.

run_container_start_checks(*started_at*, *timeout_at*)

Run startup checks.

run_start_checks(*started_at*, *timeout_at*)

Run checks to confirm that the daemon has started.

Parameters

- **started_at** (*float*) –
- **timeout_at** (*float*) –

Return type
bool

salt_call_cli(*factory_class=<class 'saltfactories.cli.call.SaltCall'>*, ***factory_class_kwargs*)

Return a *salt-call* CLI process for this minion instance.

start(**extra_cli_arguments*, *max_start_attempts=None*, *start_timeout=None*)

Start the daemon.

start_check(*callback*, **args*, ***kwargs*)

Register a function to run after the daemon starts to confirm readiness for work.

The callback must accept as the first argument `timeout_at` which is a float. The callback must stop trying to confirm running behavior once `time.time() > timeout_at`. The callback should return `True` to confirm that the daemon is ready for work.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Example:

```
def check_running_state(timeout_at: float) -> bool:
    while time.time() <= timeout_at:
        # run some checks
        ...
        # if all is good
        break
    else:
        return False
    return True
```

Parameters

- **callback** (*Callable[[...], bool]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

started(**extra_cli_arguments*, *max_start_attempts=None*, *start_timeout=None*)

Start the daemon and return it's instance so it can be used as a context manager.

stopped(*before_stop_callback=None*, *after_stop_callback=None*, *before_start_callback=None*, *after_start_callback=None*)

Stop the daemon and return it's instance so it can be used as a context manager.

Keyword Arguments:

before_stop_callback:
A callable to run before stopping the daemon. The callback must accept one argument, the daemon instance.

after_stop_callback:
A callable to run after stopping the daemon. The callback must accept one argument, the daemon instance.

before_start_callback:
A callable to run before starting the daemon. The callback must accept one argument, the daemon instance.

after_start_callback:
A callable to run after starting the daemon. The callback must accept one argument, the daemon instance.

This context manager will stop the factory while the context is in place, it re-starts it once out of context.

Example:

```
assert factory.is_running() is True

with factory.stopped():
    assert factory.is_running() is False

assert factory.is_running() is True
```

Parameters

- **before_stop_callback** (*Callable[[Daemon], None] | None*) –
- **after_stop_callback** (*Callable[[Daemon], None] | None*) –
- **before_start_callback** (*Callable[[Daemon], None] | None*) –
- **after_start_callback** (*Callable[[Daemon], None] | None*) –

Return type
Generator[*Daemon*, None, None]

terminate()
Terminate the container.

classmethod verify_config(*config*)
Verify the configuration dictionary.

classmethod write_config(*config*)
Write the configuration to file.

class saltfactories.daemons.container.**SaltMaster**(*, *cwd=_Nothing.NOTHING*, *environ=_Nothing.NOTHING*, *image*, *name=_Nothing.NOTHING*, *display_name=None*, *check_ports=_Nothing.NOTHING*, *container_run_kwargs=_Nothing.NOTHING*, *start_timeout=30*, *max_start_attempts=3*, *pull_before_start=True*, *skip_on_pull_failure=False*, *skip_if_docker_client_not_connectable=False*, *docker_client=_Nothing.NOTHING*, *before_start_callbacks=_Nothing.NOTHING*, *before_terminate_callbacks=_Nothing.NOTHING*, *after_start_callbacks=_Nothing.NOTHING*, *after_terminate_callbacks=_Nothing.NOTHING*, *container_start_checks_callbacks=_Nothing.NOTHING*, *config*, *config_dir=_Nothing.NOTHING*, *python_executable=None*, *system_service=False*, *slow_stop=True*, *system_encoding=_Nothing.NOTHING*, *timeout=_Nothing.NOTHING*, *script_name*, *base_script_args=_Nothing.NOTHING*, *stats_processes=None*, *extra_cli_arguments_after_first_start_failure=_Nothing.NOTHING*, *start_checks_callbacks=_Nothing.NOTHING*, *event_listener=None*, *factories_manager=None*, *started_at=None*, *on_auth_event_callback=None*)

Bases: *SaltDaemon*, *SaltMaster*

Salt master daemon implementation running in a docker container.

Parameters

- **cwd** (*str | Path*) –
- **environ** (*EnvironDict*) –
- **slow_stop** (*bool*) –
- **system_encoding** (*str*) –
- **timeout** (*int | float*) –
- **script_name** (*str*) –
- **base_script_args** (*List[str]*) –
- **stats_processes** (*StatsProcesses*) –
- **extra_cli_arguments_after_first_start_failure** (*List[str]*) –
- **start_checks_callbacks** (*List[Callback]*) –

get_display_name()

Returns a human readable name for the factory.

get_check_events()

Return salt events to check.

Return a list of tuples in the form of *(master_id, event_tag)* check against to ensure the daemon is running

after_start(*callback*, **args*, *on_container=False*, ***kwargs*)

Register a function callback to run after the daemon starts.

Parameters

- **callback** (*Callable*) – The function to call back
- **on_container** (*bool*) – If true, the callback will be registered on the container and not the daemon.
- **args** – The arguments to pass to the callback
- **kwargs** – The keyword arguments to pass to the callback

after_terminate(*callback*, **args*, *on_container=False*, ***kwargs*)

Register a function callback to run after the daemon terminates.

Parameters

- **callback** (*Callable*) – The function to call back
- **on_container** (*bool*) – If true, the callback will be registered on the container and not the daemon.
- **args** – The arguments to pass to the callback
- **kwargs** – The keyword arguments to pass to the callback

before_start(*callback*, **args*, *on_container=False*, ***kwargs*)

Register a function callback to run before the daemon starts.

Parameters

- **callback** (*Callable*) – The function to call back
- **on_container** (*bool*) – If true, the callback will be registered on the container and not the daemon.
- **args** – The arguments to pass to the callback
- **kwargs** – The keyword arguments to pass to the callback

before_terminate(*callback*, **args*, *on_container=False*, ***kwargs*)

Register a function callback to run before the daemon terminates.

Parameters

- **callback** (*Callable*) – The function to call back
- **on_container** (*bool*) – If true, the callback will be registered on the container and not the daemon.
- **args** – The arguments to pass to the callback
- **kwargs** – The keyword arguments to pass to the callback

static client_connectable(*docker_client*)

Check if the docker client can connect to the docker daemon.

cmdline(**args*)

Construct a list of arguments to use when starting the container.

Parameters

args (*str*) – Additional arguments to use when starting the container

classmethod configure(*factories_manager*, *daemon_id*, *root_dir=None*, *defaults=None*, *overrides=None*, ***configure_kwargs*)

Configure the salt daemon.

container_start_check(*callback*, **args*, ***kwargs*)

Register a function to run after the container starts to confirm readiness for work.

The callback must accept as the first argument `timeout_at` which is a float. The callback must stop trying to confirm running behavior once `time.time() > timeout_at`. The callback should return `True` to confirm that the daemon is ready for work.

For example:

```
def check_running_state(timeout_at: float) -> bool:
    while time.time() <= timeout_at:
        # run some checks
        ...
        # if all is good
        break
    else:
        return False
    return True
```

Parameters

- **callback** (*Callable*) – The function to call back
- **args** – The arguments to pass to the callback
- **kwargs** – The keyword arguments to pass to the callback

classmethod default_config(*root_dir*, *master_id*, *defaults=None*, *overrides=None*, *order_masters=False*, *master_of_masters=None*, *system_service=False*)

Return the default configuration.

get_base_script_args()

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

get_check_ports()

Return a list of ports to check against to ensure the daemon is running.

get_container_start_check_callbacks()

Return a list of the start check callbacks.

get_host_port_binding(*port*, *protocol='tcp'*, *ipv6=False*)

Return the host binding for a port on the container.

Parameters

- **port** (*int*) – The port.
- **protocol** (*str*) – The port protocol. Defaults to `tcp`.
- **ipv6** (*bool*) – If true, return the ipv6 port binding.

Returns
int: The matched port binding on the host. None: When not port binding was matched.

get_script_args()

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

get_script_path()

Returns the path to the script to run.

Return type
str

get_start_check_callbacks()

Return a list of the start check callbacks.

Return type
List[*Callback*]

is_running()

Returns true if the container is running.

classmethod load_config(*config_file*, *config*)

Return the loaded configuration.

property pid: int | None

The pid of the running process. None if not running.

process_output(*stdout*, *stderr*, *cmdline=None*)

Process the output. When possible JSON is loaded from the output.

Returns:
Returns a tuple in the form of `(stdout, stderr, loaded_json)`

Parameters

- **stdout** (*str*) –
- **stderr** (*str*) –
- **cmdline** (*List[str] | None*) –

Return type
Tuple[str, str, *Dict*[*Any*, *Any*] | None]

run(**cmd*, ***kwargs*)

Run a command inside the container.

run_container_start_checks(*started_at*, *timeout_at*)

Run startup checks.

run_start_checks(*started_at*, *timeout_at*)

Run checks to confirm that the daemon has started.

Parameters

- **started_at** (*float*) –
- **timeout_at** (*float*) –

Return type
bool

salt_api_daemon(***kwargs*)

Please see the documentation in *salt_api_daemon*.

salt_cli(*factory_class=<class 'saltfactories.cli.salt.Salt'>*, ***factory_class_kwargs*)

Return a *salt* CLI process for this master instance.

salt_client(*functions_known_to_return_none=None*, *factory_class=<class 'saltfactories.client.LocalClient'>*)

Return a local salt client object.

salt_cloud_cli(*defaults=None*, *overrides=None*, *factory_class=<class 'saltfactories.cli.cloud.SaltCloud'>*, ***factory_class_kwargs*)

Return a salt-cloud CLI instance.

Args:

defaults(dict):
A dictionary of default configuration to use when configuring the minion

overrides(dict):
A dictionary of configuration overrides to use when configuring the minion

Returns:

SaltCloud:
The salt-cloud CLI script process class instance

salt_cp_cli(*factory_class=<class 'saltfactories.cli.cp.SaltCp'>*, ***factory_class_kwargs*)

Return a *salt-cp* CLI process for this master instance.

salt_key_cli(*factory_class=<class 'saltfactories.cli.key.SaltKey'>*, ***factory_class_kwargs*)

Return a *salt-key* CLI process for this master instance.

salt_master_daemon(*master_id*, ***kwargs*)

This method will configure a master under a master-of-masters.

Please see the documentation in *salt_master_daemon*

salt_minion_daemon(*minion_id*, ***kwargs*)

Please see the documentation in `configure_salt_minion`.

salt_proxy_minion_daemon(*minion_id*, ***kwargs*)

Please see the documentation in *salt_proxy_minion_daemon*.

salt_run_cli(*factory_class=<class 'saltfactories.cli.run.SaltRun'>*, ***factory_class_kwargs*)

Return a *salt-run* CLI process for this master instance.

salt_spm_cli(*defaults=None*, *overrides=None*, *factory_class=<class 'saltfactories.cli.spm.Spm'>*, ***factory_class_kwargs*)

Return a *spm* CLI process for this master instance.

salt_ssh_cli(*factory_class=<class 'saltfactories.cli.ssh.SaltSsh'>*, *roster_file=None*, *target_host=None*, *client_key=None*, *ssh_user=None*, ***factory_class_kwargs*)

Return a *salt-ssh* CLI process for this master instance.

Args:

roster_file(str):
The roster file to use

target_host(str):
The target host address to connect to

client_key(str):
The path to the private ssh key to use to connect

ssh_user(str):
The remote username to connect as

`salt_syndic_daemon`(*syndic_id*, ***kwargs*)

Please see the documentation in `salt_syndic_daemon`.

`start`(**extra_cli_arguments*, *max_start_attempts=None*, *start_timeout=None*)

Start the daemon.

`start_check`(*callback*, **args*, ***kwargs*)

Register a function to run after the daemon starts to confirm readiness for work.

The callback must accept as the first argument `timeout_at` which is a float. The callback must stop trying to confirm running behavior once `time.time() > timeout_at`. The callback should return `True` to confirm that the daemon is ready for work.

Arguments:

callback:
The function to call back

Keyword Arguments:

args:
The arguments to pass to the callback

kwargs:
The keyword arguments to pass to the callback

Returns:
Nothing.

Example:

```
def check_running_state(timeout_at: float) -> bool:
    while time.time() <= timeout_at:
        # run some checks
        ...
        # if all is good
        break
    else:
        return False
    return True
```

Parameters

- **callback** (*Callable[[...], bool]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

started(**extra_cli_arguments*, *max_start_attempts=None*, *start_timeout=None*)

Start the daemon and return it's instance so it can be used as a context manager.

stopped(*before_stop_callback=None*, *after_stop_callback=None*, *before_start_callback=None*, *after_start_callback=None*)

Stop the daemon and return it's instance so it can be used as a context manager.

Keyword Arguments:

before_stop_callback:
A callable to run before stopping the daemon. The callback must accept one argument, the daemon instance.

after_stop_callback:
A callable to run after stopping the daemon. The callback must accept one argument, the daemon instance.

before_start_callback:
A callable to run before starting the daemon. The callback must accept one argument, the daemon instance.

after_start_callback:
A callable to run after starting the daemon. The callback must accept one argument, the daemon instance.

This context manager will stop the factory while the context is in place, it re-starts it once out of context.

Example:

```
assert factory.is_running() is True

with factory.stopped():
    assert factory.is_running() is False

assert factory.is_running() is True
```

Parameters

- **before_stop_callback** (*Callable[[Daemon], None] | None*) –
- **after_stop_callback** (*Callable[[Daemon], None] | None*) –
- **before_start_callback** (*Callable[[Daemon], None] | None*) –
- **after_start_callback** (*Callable[[Daemon], None] | None*) –

Return type
Generator[*Daemon*, None, None]

terminate()

Terminate the container.

classmethod verify_config(*config*)

Verify the configuration dictionary.

classmethod write_config(*config*)

Write the configuration to file.

5.5.4 CLI

salt-key

salt-key CLI factory.

class saltfactories.cli.key.**SaltKey**(*, *config*, *config_dir=_Nothing.NOTHING*, *python_executable=None*, *system_service=False*, *cwd=_Nothing.NOTHING*, *environ=_Nothing.NOTHING*, *slow_stop=True*, *system_encoding=_Nothing.NOTHING*, *timeout=_Nothing.NOTHING*, *script_name*, *base_script_args=_Nothing.NOTHING*, *hard_crash=False*, *merge_json_output=True*)

Bases: *SaltCli*

salt-key CLI factory.

Parameters

- **cwd** (*Path*) –
- **environ** (*EnvironDict*) –
- **slow_stop** (*bool*) –
- **system_encoding** (*str*) –
- **timeout** (*int | float*) –
- **script_name** (*str*) –
- **base_script_args** (*List[str]*) –

get_minion_tgt(*minion_tgt=None*)

Overridden method because salt-key does not target minions.

process_output(*stdout*, *stderr*, *cmdline=None*)

Process the returned output.

cmdline(**args*, *minion_tgt=None*, *merge_json_output=None*, ***kwargs*)

Construct a list of arguments to use when starting the subprocess.

Parameters

- **args** (*str*) – Additional arguments to use when starting the subprocess
- **minion_tgt** (*str*) – The minion ID to target
- **merge_json_output** (*bool*) – The default behavior of salt outputters is to print one line per minion return, which makes parsing the whole output as JSON impossible when targeting multiple minions. If this value is `True`, an attempt is made to merge each JSON line into a single dictionary.
- **kwargs** – Additional keyword arguments will be converted into `key=value` pairs to be consumed by the salt CLI's

get_base_script_args()

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

`get_display_name()`

Returns a human readable name for the factory.

`get_script_args()`

Returns any additional arguments to pass to the CLI script.

`get_script_path()`

Returns the path to the script to run.

Return type
str

`is_running()`

Returns true if the sub-process is alive.

Return type
bool

`property pid: int | None`

The pid of the running process. None if not running.

`run(*args, env=None, _timeout=None, **kwargs)`

Run the given command synchronously.

Keyword Arguments:

args:
The list of arguments to pass to `cmdline()` to construct the command to run

env:
Pass a dictionary of environment key, value pairs to inject into the subprocess.

_timeout:
The timeout value for this particular `run()` call. If this value is not `None`, it will be used instead of `timeout`, the default timeout.

Parameters

- **args** (*str*) –
- **env** (*EnvironDict | None*) –
- **_timeout** (*int | float | None*) –
- **kwargs** (*Any*) –

Return type
ProcessResult

`terminate()`

Terminate the started subprocess.

Return type
ProcessResult

salt

`salt` CLI factory.

class saltfactories.cli.salt.**Salt**(*, *config*, *config_dir=_Nothing.NOTHING*, *python_executable=None*, *system_service=False*, *cwd=_Nothing.NOTHING*, *environ=_Nothing.NOTHING*, *slow_stop=True*, *system_encoding=_Nothing.NOTHING*, *timeout=_Nothing.NOTHING*, *script_name*, *base_script_args=_Nothing.NOTHING*, *hard_crash=False*, *merge_json_output=True*)

Bases: *SaltCli*

salt CLI factory.

Parameters

- **cwd** (*Path*) –
- **environ** (*EnvironDict*) –
- **slow_stop** (*bool*) –
- **system_encoding** (*str*) –
- **timeout** (*int | float*) –
- **script_name** (*str*) –
- **base_script_args** (*List[str]*) –

cmdline(**args*, *minion_tgt=None*, ***kwargs*)

Build the command line for the CLI in question.

process_output(*stdout*, *stderr*, *cmdline=None*)

Process the returned output.

get_base_script_args()

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

get_display_name()

Returns a human readable name for the factory.

get_minion_tgt(*minion_tgt=None*)

Return the minion target ID.

get_script_args()

Returns any additional arguments to pass to the CLI script.

get_script_path()

Returns the path to the script to run.

Return type
str

is_running()

Returns true if the sub-process is alive.

Return type
bool

property pid: int | None

The pid of the running process. None if not running.

run(*args, env=None, _timeout=None, **kwargs)

Run the given command synchronously.

Keyword Arguments:

args:
The list of arguments to pass to `cmdline()` to construct the command to run

env:
Pass a dictionary of environment key, value pairs to inject into the subprocess.

_timeout:
The timeout value for this particular `run()` call. If this value is not `None`, it will be used instead of `timeout`, the default timeout.

Parameters

- **args** (*str*) –
- **env** (*EnvironDict | None*) –
- **_timeout** (*int | float | None*) –
- **kwargs** (*Any*) –

Return type
ProcessResult

terminate()

Terminate the started subprocess.

Return type
ProcessResult

salt-call

`salt-call` CLI factory.

class saltfactories.cli.call.SaltCall(*, *config*, *config_dir=_Nothing.NOTHING*, *python_executable=None*, *system_service=False*, *cwd=_Nothing.NOTHING*, *environ=_Nothing.NOTHING*, *slow_stop=True*, *system_encoding=_Nothing.NOTHING*, *timeout=_Nothing.NOTHING*, *script_name*, *base_script_args=_Nothing.NOTHING*, *hard_crash=False*, *merge_json_output=True*)

Bases: *SaltCli*

`salt-call` CLI factory.

Parameters

- **cwd** (*Path*) –
- **environ** (*EnvironDict*) –
- **slow_stop** (*bool*) –
- **system_encoding** (*str*) –

- **timeout** (*int | float*) –
- **script_name** (*str*) –
- **base_script_args** (*List[str]*) –

get_minion_tgt(*minion_tgt=None*)

Overridden method because salt-run does not target minions, it runs locally.

process_output(*stdout*, *stderr*, *cmdline=None*)

Process the returned output.

cmdline(**args*, *minion_tgt=None*, *merge_json_output=None*, ***kwargs*)

Construct a list of arguments to use when starting the subprocess.

Parameters

- **args** (*str*) – Additional arguments to use when starting the subprocess
- **minion_tgt** (*str*) – The minion ID to target
- **merge_json_output** (*bool*) – The default behavior of salt outputters is to print one line per minion return, which makes parsing the whole output as JSON impossible when targeting multiple minions. If this value is `True`, an attempt is made to merge each JSON line into a single dictionary.
- **kwargs** – Additional keyword arguments will be converted into `key=value` pairs to be consumed by the salt CLI's

get_base_script_args()

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

get_display_name()

Returns a human readable name for the factory.

get_script_args()

Returns any additional arguments to pass to the CLI script.

get_script_path()

Returns the path to the script to run.

Return type
str

is_running()

Returns true if the sub-process is alive.

Return type
bool

property pid: int | None

The pid of the running process. None if not running.

run(**args*, *env=None*, *_timeout=None*, ***kwargs*)

Run the given command synchronously.

Keyword Arguments:

args:
The list of arguments to pass to `cmdline()` to construct the command to run

env:
Pass a dictionary of environment key, value pairs to inject into the subprocess.

_timeout:
The timeout value for this particular `run()` call. If this value is not `None`, it will be used instead of `timeout`, the default timeout.

Parameters

- **args** (*str*) –
- **env** (*EnvironDict | None*) –
- **_timeout** (*int | float | None*) –
- **kwargs** (*Any*) –

Return type
ProcessResult

`terminate()`
Terminate the started subprocess.

Return type
ProcessResult

salt-run

`salt-run` CLI factory.

`class saltfactories.cli.run.SaltRun(*, config, config_dir=_Nothing.NOTHING, python_executable=None, system_service=False, cwd=_Nothing.NOTHING, environ=_Nothing.NOTHING, slow_stop=True, system_encoding=_Nothing.NOTHING, timeout=_Nothing.NOTHING, script_name, base_script_args=_Nothing.NOTHING, hard_crash=False, merge_json_output=True)`

Bases: *SaltCli*

salt-run CLI factory.

Parameters

- **cwd** (*Path*) –
- **environ** (*EnvironDict*) –
- **slow_stop** (*bool*) –
- **system_encoding** (*str*) –
- **timeout** (*int | float*) –
- **script_name** (*str*) –
- **base_script_args** (*List[str]*) –

`get_minion_tgt(minion_tgt=None)`
Overridden method because salt-run does not target minions.

process_output(*stdout*, *stderr*, *cmdline=None*)

Process the returned output.

cmdline(**args*, *minion_tgt=None*, *merge_json_output=None*, ***kwargs*)

Construct a list of arguments to use when starting the subprocess.

Parameters

- **args** (*str*) – Additional arguments to use when starting the subprocess
- **minion_tgt** (*str*) – The minion ID to target
- **merge_json_output** (*bool*) – The default behavior of salt outputters is to print one line per minion return, which makes parsing the whole output as JSON impossible when targeting multiple minions. If this value is `True`, an attempt is made to merge each JSON line into a single dictionary.
- **kwargs** – Additional keyword arguments will be converted into `key=value` pairs to be consumed by the salt CLI's

get_base_script_args()

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

get_display_name()

Returns a human readable name for the factory.

get_script_args()

Returns any additional arguments to pass to the CLI script.

get_script_path()

Returns the path to the script to run.

Return type
str

is_running()

Returns true if the sub-process is alive.

Return type
bool

property pid: int | None

The pid of the running process. None if not running.

run(**args*, *env=None*, *_timeout=None*, ***kwargs*)

Run the given command synchronously.

Keyword Arguments:

args:
The list of arguments to pass to `cmdline()` to construct the command to run

env:
Pass a dictionary of environment key, value pairs to inject into the subprocess.

_timeout:
The timeout value for this particular `run()` call. If this value is not `None`, it will be used instead of `timeout`, the default timeout.

Parameters

- **args** (*str*) –
- **env** (*EnvironDict | None*) –
- **_timeout** (*int | float | None*) –
- **kwargs** (*Any*) –

Return type
ProcessResult

terminate()

Terminate the started subprocess.

Return type
ProcessResult

salt-cp

salt-cp CLI factory.

class saltfactories.cli.cp.**SaltCp**(*, *config*, *config_dir=_Nothing.NOTHING*, *python_executable=None*, *system_service=False*, *cwd=_Nothing.NOTHING*, *environ=_Nothing.NOTHING*, *slow_stop=True*, *system_encoding=_Nothing.NOTHING*, *timeout=_Nothing.NOTHING*, *script_name*, *base_script_args=_Nothing.NOTHING*, *hard_crash=False*, *merge_json_output=True*)

Bases: *SaltCli*

salt-cp CLI factory.

Parameters

- **cwd** (*Path*) –
- **environ** (*EnvironDict*) –
- **slow_stop** (*bool*) –
- **system_encoding** (*str*) –
- **timeout** (*int | float*) –
- **script_name** (*str*) –
- **base_script_args** (*List[str]*) –

process_output(*stdout*, *stderr*, *cmdline=None*)

Process the returned output.

cmdline(**args*, *minion_tgt=None*, *merge_json_output=None*, ***kwargs*)

Construct a list of arguments to use when starting the subprocess.

Parameters

- **args** (*str*) – Additional arguments to use when starting the subprocess
- **minion_tgt** (*str*) – The minion ID to target

- **merge_json_output** (*bool*) – The default behavior of salt outputters is to print one line per minion return, which makes parsing the whole output as JSON impossible when targeting multiple minions. If this value is `True`, an attempt is made to merge each JSON line into a single dictionary.
- **kwargs** – Additional keyword arguments will be converted into `key=value` pairs to be consumed by the salt CLI's

`get_base_script_args()`

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

`get_display_name()`

Returns a human readable name for the factory.

`get_minion_tgt(minion_tgt=None)`

Return the minion target ID.

`get_script_args()`

Returns any additional arguments to pass to the CLI script.

`get_script_path()`

Returns the path to the script to run.

Return type
str

`is_running()`

Returns true if the sub-process is alive.

Return type
bool

`property pid: int | None`

The pid of the running process. None if not running.

`run(*args, env=None, _timeout=None, **kwargs)`

Run the given command synchronously.

Keyword Arguments:

args:
The list of arguments to pass to `cmdline()` to construct the command to run

env:
Pass a dictionary of environment key, value pairs to inject into the subprocess.

_timeout:
The timeout value for this particular `run()` call. If this value is not `None`, it will be used instead of `timeout`, the default timeout.

Parameters

- **args** (*str*) –
- **env** (*EnvironDict | None*) –
- **_timeout** (*int | float | None*) –
- **kwargs** (*Any*) –

Return type
ProcessResult

`terminate()`

Terminate the started subprocess.

Return type
ProcessResult

salt-cloud

`salt-cloud` CLI factory.

`class saltfactories.cli.cloud.SaltCloud(*, config, config_dir=_Nothing.NOTHING, python_executable=None, system_service=False, cwd=_Nothing.NOTHING, environ=_Nothing.NOTHING, slow_stop=True, system_encoding=_Nothing.NOTHING, timeout=_Nothing.NOTHING, script_name, base_script_args=_Nothing.NOTHING, hard_crash=False, merge_json_output=True)`

Bases: *SaltCli*

salt-cloud CLI factory.

Parameters

- **cwd** (*Path*) –
- **environ** (*EnvironDict*) –
- **slow_stop** (*bool*) –
- **system_encoding** (*str*) –
- **timeout** (*int | float*) –
- **script_name** (*str*) –
- **base_script_args** (*List[str]*) –

`static default_config(root_dir, master_id, defaults=None, overrides=None)`

Return the default configuration for the daemon.

`classmethod configure(factories_manager, daemon_id, root_dir=None, defaults=None, overrides=None, **configure_kwargs)`

Configure the CLI.

`classmethod verify_config(config)`

Verify the configuration dictionary.

`classmethod write_config(config)`

Verify the loaded configuration.

`cmdline(*args, minion_tgt=None, merge_json_output=None, **kwargs)`

Construct a list of arguments to use when starting the subprocess.

Parameters

- **args** (*str*) – Additional arguments to use when starting the subprocess
- **minion_tgt** (*str*) – The minion ID to target

- **merge_json_output** (*bool*) – The default behavior of salt outputters is to print one line per minion return, which makes parsing the whole output as JSON impossible when targeting multiple minions. If this value is `True`, an attempt is made to merge each JSON line into a single dictionary.
- **kwargs** – Additional keyword arguments will be converted into `key=value` pairs to be consumed by the salt CLI's

get_base_script_args()

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

get_display_name()

Returns a human readable name for the factory.

get_minion_tgt(*minion_tgt=None*)

Return the minion target ID.

get_script_args()

Returns any additional arguments to pass to the CLI script.

get_script_path()

Returns the path to the script to run.

Return type
str

is_running()

Returns true if the sub-process is alive.

Return type
bool

property pid: int | None

The pid of the running process. None if not running.

process_output(*stdout*, *stderr*, *cmdline=None*)

Process the output. When possible JSON is loaded from the output.

Returns
Returns a tuple in the form of `(stdout, stderr, loaded_json)`

Return type
tuple

run(**args*, *env=None*, *_timeout=None*, ***kwargs*)

Run the given command synchronously.

Keyword Arguments:

args:
The list of arguments to pass to `cmdline()` to construct the command to run

env:
Pass a dictionary of environment key, value pairs to inject into the subprocess.

_timeout:
The timeout value for this particular `run()` call. If this value is not `None`, it will be used instead of `timeout`, the default timeout.

Parameters

- **args** (*str*) –
- **env** (*EnvironDict | None*) –
- **_timeout** (*int | float | None*) –
- **kwargs** (*Any*) –

Return type
ProcessResult

terminate()

Terminate the started subprocess.

Return type
ProcessResult

spm

spm CLI factory.

class saltfactories.cli.spm.**Spm**(**, config, config_dir=_Nothing.NOTHING, python_executable=None, system_service=False, cwd=_Nothing.NOTHING, environ=_Nothing.NOTHING, slow_stop=True, system_encoding=_Nothing.NOTHING, timeout=_Nothing.NOTHING, script_name, base_script_args=_Nothing.NOTHING, hard_crash=False, merge_json_output=True*)

Bases: *SaltCli*

spm CLI factory.

Parameters

- **cwd** (*Path*) –
- **environ** (*EnvironDict*) –
- **slow_stop** (*bool*) –
- **system_encoding** (*str*) –
- **timeout** (*int | float*) –
- **script_name** (*str*) –
- **base_script_args** (*List[str]*) –

get_minion_tgt(*minion_tgt=None*)

Overridden method because spm does not target minions.

static default_config(*root_dir, master_factory, defaults=None, overrides=None*)

Return the default configuration for the daemon.

classmethod configure(*master_factory, root_dir=None, defaults=None, overrides=None*)

Configure the CLI.

classmethod verify_config(*config*)

Verify the configuration dictionary.

classmethod write_config(*config*)

Verify the loaded configuration.

cmdline(**args*, *minion_tgt=None*, *merge_json_output=None*, ***kwargs*)

Construct a list of arguments to use when starting the subprocess.

Parameters

- **args** (*str*) – Additional arguments to use when starting the subprocess
- **minion_tgt** (*str*) – The minion ID to target
- **merge_json_output** (*bool*) – The default behavior of salt outputters is to print one line per minion return, which makes parsing the whole output as JSON impossible when targeting multiple minions. If this value is `True`, an attempt is made to merge each JSON line into a single dictionary.
- **kwargs** – Additional keyword arguments will be converted into `key=value` pairs to be consumed by the salt CLI's

get_base_script_args()

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

get_display_name()

Returns a human readable name for the factory.

get_script_args()

Returns any additional arguments to pass to the CLI script.

get_script_path()

Returns the path to the script to run.

Return type
str

is_running()

Returns true if the sub-process is alive.

Return type
bool

property pid: int | None

The pid of the running process. None if not running.

process_output(*stdout*, *stderr*, *cmdline=None*)

Process the output. When possible JSON is loaded from the output.

Returns
Returns a tuple in the form of `(stdout, stderr, loaded_json)`

Return type
tuple

run(**args*, *env=None*, *_timeout=None*, ***kwargs*)

Run the given command synchronously.

Keyword Arguments:

args:
The list of arguments to pass to `cmdline()` to construct the command to run

env:
Pass a dictionary of environment key, value pairs to inject into the subprocess.

_timeout:
The timeout value for this particular `run()` call. If this value is not `None`, it will be used instead of `timeout`, the default timeout.

Parameters

- **args** (*str*) –
- **env** (*EnvironDict | None*) –
- **_timeout** (*int | float | None*) –
- **kwargs** (*Any*) –

Return type
ProcessResult

`terminate()`

Terminate the started subprocess.

Return type
ProcessResult

salt-ssh

`salt-ssh` CLI factory.

`class saltfactories.cli.ssh.SaltSsh(*, config, config_dir=_Nothing.NOTHING, python_executable=None, system_service=False, cwd=_Nothing.NOTHING, environ=_Nothing.NOTHING, slow_stop=True, system_encoding=_Nothing.NOTHING, timeout=_Nothing.NOTHING, script_name, base_script_args=_Nothing.NOTHING, hard_crash=False, merge_json_output=True, roster_file=None, client_key=None, target_host=None, ssh_user=None)`

Bases: *SaltCli*

salt CLI factory.

Parameters

- **cwd** (*Path*) –
- **environ** (*EnvironDict*) –
- **slow_stop** (*bool*) –
- **system_encoding** (*str*) –
- **timeout** (*int | float*) –
- **script_name** (*str*) –
- **base_script_args** (*List[str]*) –

get_script_args()

Return the CLI arguments for the script.

get_minion_tgt(*minion_tgt=None*)

Return the minion ID to target.

cmdline(**args*, *minion_tgt=None*, *merge_json_output=None*, ***kwargs*)

Construct a list of arguments to use when starting the subprocess.

Parameters

- **args** (*str*) – Additional arguments to use when starting the subprocess
- **minion_tgt** (*str*) – The minion ID to target
- **merge_json_output** (*bool*) – The default behavior of salt outputters is to print one line per minion return, which makes parsing the whole output as JSON impossible when targeting multiple minions. If this value is `True`, an attempt is made to merge each JSON line into a single dictionary.
- **kwargs** – Additional keyword arguments will be converted into `key=value` pairs to be consumed by the salt CLI's

get_base_script_args()

Returns any additional arguments to pass to the CLI script.

Return type
List[str]

get_display_name()

Returns a human readable name for the factory.

get_script_path()

Returns the path to the script to run.

Return type
str

is_running()

Returns true if the sub-process is alive.

Return type
bool

property pid: int | None

The pid of the running process. None if not running.

process_output(*stdout*, *stderr*, *cmdline=None*)

Process the output. When possible JSON is loaded from the output.

Returns
Returns a tuple in the form of `(stdout, stderr, loaded_json)`

Return type
tuple

run(**args*, *env=None*, *_timeout=None*, ***kwargs*)

Run the given command synchronously.

Keyword Arguments:

args:
The list of arguments to pass to `cmdline()` to construct the command to run

env:
Pass a dictionary of environment key, value pairs to inject into the subprocess.

_timeout:
The timeout value for this particular `run()` call. If this value is not `None`, it will be used instead of `timeout`, the default timeout.

Parameters

- **args** (*str*) –
- **env** (*EnvironDict | None*) –
- **_timeout** (*int | float | None*) –
- **kwargs** (*Any*) –

Return type
ProcessResult

`terminate()`

Terminate the started subprocess.

Return type
ProcessResult

Local Client

Salt Client in-process implementation.

`class saltfactories.client.LocalClient(*, master_config, functions_known_to_return_none=_Nothing.NOTHING)`

Bases: `object`

Wrapper class around Salt's local client.

`run(function, *args, minion_tgt='minion', timeout=300, **kwargs)`

Run a single salt function.

Additional condition the return down to match the behavior of the raw function call.

CHAPTER

SIX

CHANGELOG

Versions follow Semantic Versioning (<*major*>.<*minor*>.<*patch*>).

Backward incompatible (breaking) changes will only be introduced in major versions with advance notice in the **Deprecations** section of releases.

Previous Changelog Entries

Before the 1.0.0 release, due to the fast evolving pace and breakage introduced while developing the library, no changelog was kept. Please refer to the git history for details.

6.1 [UNRELEASED DRAFT] (2024-03-21)

No significant changes.

6.2 1.0.0 (2024-03-21)

6.2.1 Improvements

- The *SSHD* daemon implementation now has a *get_host_keys* method which returns the host keys that can then be written to a `known_hosts` file. (#176)

6.2.2 Trivial/Internal Changes

- CI pipeline changes:
  - Stop testing against Pytest 8.0.0rc2 and instead test against 8.0.x
  - Stop testing against Salt 3005.x
  - Add Salt 3007.x to the versions to test
  - Stop testing against Pytest 7.3.x and add 8.1.x to the list of versions to test (#177)

6.3 1.0.0rc29 (2024-01-23)

6.3.1 Improvements

- Add `--sys-info-and-exit` which basically prints the system information and exit. Doesn't run any tests. (#173)

6.3.2 Trivial/Internal Changes

- Switch pipelines to use Python 3.11 and start testing Pytest 8.0.0rc2 (#173)

6.4 1.0.0rc28 (2023-11-25)

6.4.1 Features

- Added a containarized salt master class implementation, `SaltMaster` (#169)

6.4.2 Improvements

- Switch to testing against Salt 3006.x instead of 3005.x (#169)

6.4.3 Bug Fixes

- The `Container._pull_container` callback now properly registers on the `SaltMinion` and the `SaltMaster` classes when `pull_before_start` is True (#168)

6.4.4 Improved Documentation

- Fix the readthedocs builds due to https://blog.readthedocs.com/migrate-configuration-v2/ (#169)

6.5 1.0.0rc27 (2023-09-27)

6.5.1 Bug Fixes

- Allow the Salt engine to run on Python 3.6 (#167)

6.6 1.0.0rc26 (2023-09-20)

6.6.1 Bug Fixes

- Check if path exists before running additional checks on the *temp_directory* context manager. (#160)
- The `container` implementation is now sensible to the `exited` state when starting containers. (#165)

6.6.2 Improved Documentation

- Updated documentation for SaltEnv temp_file and added an example usage (#163)

6.7 1.0.0rc25 (2023-07-31)

6.7.1 Improvements

- Remove *pytest-tempdir* package dependency (#154)
- Stop using deprecated *@pytest.mark.trylast* (#155)
- Simplify and reduce the amount of patching required to unit test loader modules (#156)

6.7.2 Trivial/Internal Changes

- Some internal processes improvements:
  - Publish packages to PyPi with trusted publishers
  - Enable dependabot to update the GH Actions versions on a weekly basis (#151)
- Start using actionlint and shellcheck to validate GH Actions workflows (#153)
- Improve code coverage by either removing code not getting used anymore or marking sections of the code which are not expected to be covered (#157)

6.8 1.0.0rc24 (2023-07-27)

6.8.1 Improvements

- Several improvements to reduce failure points:
  - Log the exception instead of raising it.
  - Always populate the **_dirs* config settings, regardless of how salt-factories is being used
  - Improved the connect/disconnect behavior of the event listener client
  - The minimum supported Salt version is now 3005.0
  - The minimum supported Pytest version is now 7.0.0 (#149)

6.8.2 Bug Fixes

- Do not blindly overwrite the *retuner_address* configuration key (#146)

6.8.3 Trivial/Internal Changes

- Start checking the code base with ruff (#149)

6.9 1.0.0rc23 (2022-12-15)

6.9.1 Bug Fixes

- Fixed Salt's deferred imports to allow onedir builds while not breaking non-onedir builds:
  - Additionally, stopped relying on *salt.utils.files* and *salt.utils.yaml*
  - Stopped using *zmq* to forward events(this was where the breakage was showing) for a plain TCP implementation.
  - The *event_listener* fixture is now started/stopped like a regular pytest fixture
  - The *event_listener* server now restarts in case something goes wrong to the point where it crashes. (#146)

6.10 1.0.0rc22 (2022-12-02)

6.10.1 Breaking Changes

- Drop support for Python 3.5 and 3.6 (#123)

6.10.2 Improvements

- Defer all *salt* imports so that we can use pytest-salt-factories to test onedir builds (#144)
- A few improvements to functional testing support:
  - Allow *StateReturn* to be accessed by key instead of just attribute
  - Add warning for when more than a state function is used under the same state key
  - Return an instance of *MatchString* for *StateResult.comment* (#145)

6.10.3 Trivial/Internal Changes

- Update the github actions versions to avoid deprecation errors (#145)

6.11 1.0.0rc21 (2022-11-04)

6.11.1 Improvements

- Several improvements to the state module wrappers:
  - Allow getting the state chunk by *__id__* on MultiStateResult
  - Wrap a few more functions from *salt.modules.state* (#140)

6.11.2 Trivial/Internal Changes

- Pipeline and requirements fixes:
  - Test against 3005.* and not 3005rc2 since it's now released.
  - Install *importlib-metadata<5.0.0* since only Salt>=3006 will be able to handle it (#140)

6.12 1.0.0rc20 (2022-08-25)

6.12.1 Bug Fixes

- The *spm* CLI now properly lays down the configuration files required (#137)

6.13 1.0.0rc19 (2022-08-22)

6.13.1 Breaking Changes

- In *saltfactories.utils.cli_scipts.generate_script()*:
  - For coverage tracking, both *coverate_db_path* and *coverage_rc_path* must be passed. They will not be infered by *root_dir*.
  - *inject_coverage* was removed. (#135)
- The minimum Salt version in now *3004* (#136)

6.13.2 Trivial/Internal Changes

- CI and internal changes:
  - Start testing Salt 3005.x (rc2 for now)
  - Skip testing 3005rc2 on windows and macOS for now.
  - Lock system tests to a version of nox that still works
  - Bump python version to 3.9 for lint workflow
  - Bumped pylint requirement to *2.14.5* and cleaned up issues
  - Don't build the salt minion container during test runs, pull an existing container. (#136)

6.14 1.0.0rc18 (2022-07-14)

6.14.1 Breaking Changes

- Renamed the `system_install` configuration flag, markers and behaviours when set to `system_service` to better reflect what it's actually used for. (#96)

6.14.2 Features

- Allow passing `--python-executable` to teak which python get's used to prefix CLI commands, when needed. (#129)
- Allow passing `--scripts-dir` to tell salt-factories where to look for the Salt daemon and CLI scripts. The several scripts to the Salt daemons and CLI's **must** exist. Also, passing this option will additionally make salt-factories **NOT** generate said scripts and set `python_executable` to `None` (#130)
- Added CLI support(`--system-service`) to change salt-factories to use Salt previously installed from the platform's package manager. (#131)
- Inject `engines_dirs` and `log_handlers_dirs` when `system_service=True` or `scripts_path` is not `None` These flags suggest that the salt being imported and used by salt-factories might not be the same as the one being tested. So, in this case, make sure events and logging from started daemons still get forwarded to salt-factories. (#133)

6.15 1.0.0rc17 (2022-06-17)

6.15.1 Bug Fixes

- Bump deprecations targeted for 2.0.0 to 3.0.0 (#122)
- Try to pass `loaded_base_name` to each of Salt's loaders used in our `Loaders` class, if not supported, patch it at runtime. (#126)
- `saltfactories.utils.warn_until()` is now aware of Pytest's rewrite calls and properly reports the offending code. (#127)

6.16 1.0.0rc16 (2022-05-28)

6.16.1 Improvements

- Switch to internal start check `callables`.

  Additionally, significant container improvements, like:

  - Get host ports to check from the container port bindings.
  - Always terminate the containers.
  - Support randomly assigned host port bindings

  `skip_on_salt_system_install` is now also a marker provided by `pytest-salt-factories`. (#120)

6.17 1.0.0rc15 (2022-05-09)

6.17.1 Improvements

- Now that the new logging changes are merged into Salt's master branch, adjust detection of those changes on `SaltKey`. (#118)

6.17.2 Bug Fixes

- `--timeout` is now correctly passed for CLI factories when either `timeout` is defined on the configuration or when `timeout` is passed to the CLI factory constructor. (#117)

6.17.3 Trivial/Internal Changes

- Test PyTest 7.0.x and 7.1.x & Fix tests requirements
  - Don't allow `pytest-subtests` to upgrade pytest
  - Test under PyTest 7.0.x and 7.1.x
  - Force Jinja2 to be < 3.1 on Salt 3003.x
  - Fix the requirements of the example `echo-extension`
  - Explicitly pass a timeout to Salt CLI's on spawning platforms.
  - Windows builds were not getting passed the `PYTEST_VERSION_REQUIREMENT` env var. (#116)

6.18 1.0.0rc14 (2022-04-06)

6.18.1 Bug Fixes

- Fixed container tests not passing on macOS (#114)

6.18.2 Trivial/Internal Changes

- Pin click on the black pre-commit hooks (#115)

6.19 1.0.0rc13 (2022-03-28)

6.19.1 Bug Fixes

- Handle docker client initialization error on macOS. (#113)

6.20 1.0.0rc12 (2022-03-27)

6.20.1 Bug Fixes

- Catch `APIError` when removing containers (#112)

6.21 1.0.0rc11 (2022-03-22)

6.21.1 Improvements

- Provide a `SECURITY.md` file for the project (#67)
- It's no longer necessary to pass a docker client instance as `docker_client` when using containers. (#111)

6.22 1.0.0rc10 (2022-03-21)

6.22.1 Improvements

- The docker container daemon now pulls the image by default prior to starting it. (#109)

6.22.2 Bug Fixes

- Provide backwards compatibility imports for the old factory exceptions, now in pytest-shell-utilities (#108)
- Base classes for the `SaltDaemon` containers order is now fixed. (#110)

6.23 1.0.0rc9 (2022-03-20)

6.23.1 Improvements

- Use old-style Salt entrypoints for improved backwards compatibility. (#98)

6.24 1.0.0rc8 (2022-03-12)

6.24.1 Bug Fixes

- Instead of just removing *saltfactories.utils.ports* and *saltfactories.utils.processes*, redirect the imports to the right library and show a deprecation warning. (#106)

6.25 1.0.0rc7 (2022-02-19)

6.25.1 Bug Fixes

- The containers factory does not accept the `stats_processes` keyword. (#105)

6.26 1.0.0rc6 (2022-02-17)

6.26.1 Bug Fixes

- Include the started daemons in the `stats_processes` dictionary (#104)

6.27 1.0.0rc5 (2022-02-17)

6.27.1 Improvements

- Wipe the `cachedir` for on each `saltfactories.utils.functional.Loaders` reset (#103)

6.28 1.0.0rc4 (2022-02-17)

6.28.1 Bug Fixes

- Properly handle missing keys in the configuration for the pytest salt logging handler. (#101)
- Fix passing `--timeout` to Salt's CLI's (#102)

6.29 1.0.0rc3 (2022-02-16)

6.29.1 Bug Fixes

- Fix `pathlib.path` typo (#99)
- Fixed issue with `sdist` recompression for reproducible packages not iterating though subdirectories contents. (#100)

6.30 1.0.0rc2 (2022-02-14)

6.30.1 Improvements

- Improve documentation (#92)

6.30.2 Bug Fixes

- Fix issue where, on system installations, the minion ID on the configuration, if not explicitly passed on `overrides` or `defaults`, would default to the master ID used to create the salt minion factory. (#93)
- Allow configuring `root_dir` in `setup_salt_factories` fixture (#95)

6.31 0.912.2 (2022-02-14)

6.31.1 Bug Fixes

- Use salt's entry-points instead of relying on loader `*_dirs` configs (#98)

6.32 0.912.1 (2022-02-05)

6.32.1 Improvements

- Set lower required python version to 3.5.2 (#97)

6.33 1.0.0rc1 (2022-01-27)

6.33.1 Breaking Changes

- Switch to the extracted pytest plugins
  - Switch to pytest-system-statistics
  - Switch to pytest-shell-utilities (#90)

6.34 0.912.0 (2022-01-25)

6.34.1 Breaking Changes

- Name things once. (#50)
- `get_unused_localhost_port` no longer cached returned port by default (#51)
- Rename the `SaltMaster.get_salt_cli` to `SaltMaster.salt_cli`, forgotten on PR #50 (#70)

6.34.2 Features

- Temporary state tree management
  - Add `temp_file` and `temp_directory` support as pytest helpers
  - Add `SaltStateTree` and `SaltPillarTree` for easier temp files support (#38)
- Added skip markers for AArch64 platform, `skip_on_aarch64` and `skip_unless_on_aarch64` (#40)
- Added a `VirtualEnv` helper class to create and interact with a virtual environment (#43)
- Add `skip_on_spawning_platform` and `skip_unless_on_spawning_platform` markers (#81)

6.34.3 Improvements

- Switch project to an `src/` based layout (#41)
- Start using towncrier to maintain the changelog (#42)
- Forwarding logs, file and pillar roots fixes
  - Salt allows minions and proxy minions to also have file and pillar roots configured
  - All factories will now send logs of level `debug` or higher to the log server (#49)
- Log the test outcome (#52)
- Take into account that `SystemExit.code` might not be an integer on the generated CLI scripts (#62)
- Catch unhandled exceptions and write their traceback to `sys.stderr` in the generated CLI scripts (#63)
- Several fixes/improvements to the `ZMQHandler` log forwarding handler (#64)
- ZMQ needs to reconnect on forked processes or else Salt's own multiprocessing log forwarding log records won't be logged by the `ZMQHandler` (#69)
- Some more additional changes to the ZMQHandler to make sure it's resources are cleaned when terminating (#74)
- The `sshd` server no longer generates `dsa` keys if the system has FIPS enabled (#80)
- Add `to_salt_config` method to `SaltEnv` and `SaltEnvs`. This will simplify augmenting the salt configuration dictionary. (#82)
- Rename `SaltEnv.to_salt_config()` to `SaltEnv.as_dict()` (#83)
- Switch to pytest-skip-markers. (#84)

6.34.4 Bug Fixes

- Adjust to the upcoming salt loader changes (#77)

6.34.5 Trivial/Internal Changes

- CI pileline adjustements
  - Bump salt testing requirement to 3002.6
  - Drop testing of FreeBSD since it's too unreliable on Github Actions
  - Full clone when testing so that codecov does not complain (#39)
- Upgrade to black 21.4b2 (#56)
- Drop Pytest requirement to 6.0.0 (#57)
- Increase and match CI system tests *timeout-minutes* to Linux tests *timeout-minutes* (#64)
- Switch to the new codecov uploader (#72)
- Fix codecov flags, report name, and coverage (#73)
- Update to latest versions on some pre-commit hooks
  - `pyupgrade`: 2.23.3
  - `reorder_python_imports`: 2.6.0
  - `black`: 21.b7
  - `blacken-docs`: 1.10.0 (#79)
- Remove `transport` keyword argument from the call to `salt.utils.event.get_event` (#87)
- Add `build` and `release` nox targets (#89)

PYTHON MODULE INDEX

INDEX

A

B

C

D

E

F

G

H

I

L

M

N

P

R

S

T

U

V

W

Z