orphan:

Unit Testing Ansible Modules

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]testing_units_modules.rst`, **line 9)**
>
> Unknown directive type "highlight".
>
> ```
> .. highlight:: python
> ```

Topics

Introduction

This document explains why, how and when you should use unit tests for Ansible modules. The document doesn't apply to other parts of Ansible for which the recommendations are normally closer to the Python standard. There is basic documentation for Ansible unit tests in the developer guide :ref:`testing_units`. This document should be readable for a new Ansible module author. If you find it incomplete or confusing, please open a bug or ask for help on the #ansible-devel chat channel (using Matrix at ansible.im or using IRC at irc.libera.chat).

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]testing_units_modules.rst`, **line 16)**; ***backlink***
>
> Unknown interpreted text role "ref".

What Are Unit Tests?

Ansible includes a set of unit tests in the :file:`test/units` directory. These tests primarily cover the internals but can also cover Ansible modules. The structure of the unit tests matches the structure of the code base, so the tests that reside in the :file:`test/units/modules/` directory are organized by module groups.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]testing_units_modules.rst`, **line 26)**; ***backlink***
>
> Unknown interpreted text role "file".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]testing_units_modules.rst`, **line 26)**; ***backlink***
>
> Unknown interpreted text role "file".

Integration tests can be used for most modules, but there are situations where cases cannot be verified using integration tests. This means that Ansible unit test cases may extend beyond testing only minimal units and in some cases will include some level of functional testing.

Why Use Unit Tests?

Ansible unit tests have advantages and disadvantages. It is important to understand these. Advantages include:

- Most unit tests are much faster than most Ansible integration tests. The complete suite of unit tests can be run regularly by a developer on their local system.
- Unit tests can be run by developers who don't have access to the system which the module is designed to work on, allowing a level of verification that changes to core functions haven't broken module expectations.
- Unit tests can easily substitute system functions allowing testing of software that would be impractical. For example, the `sleep()` function can be replaced and we check that a ten minute sleep was called without actually waiting ten minutes.
- Unit tests are run on different Python versions. This allows us to ensure that the code behaves in the same way on different Python versions.

There are also some potential disadvantages of unit tests. Unit tests don't normally directly test actual useful valuable features of software, instead just internal implementation

- Unit tests that test the internal, non-visible features of software may make refactoring difficult if those internal features have to change (see also naming in How below)
- Even if the internal feature is working correctly it is possible that there will be a problem between the internal code tested and the actual result delivered to the user

Normally the Ansible integration tests (which are written in Ansible YAML) provide better testing for most module functionality. If those tests already test a feature and perform well there may be little point in providing a unit test covering the same area as well.

When To Use Unit Tests

There are a number of situations where unit tests are a better choice than integration tests. For example, testing things which are impossible, slow or very difficult to test with integration tests, such as:

- Forcing rare / strange / random situations that can't be forced, such as specific network failures and exceptions
- Extensive testing of slow configuration APIs
- Situations where the integration tests cannot be run as part of the main Ansible continuous integration running in Azure Pipelines.

Providing quick feedback

Example:

> A single step of the rds_instance test cases can take up to 20 minutes (the time to create an RDS instance in Amazon). The entire test run can last for well over an hour. All 16 of the unit tests complete execution in less than 2 seconds.

The time saving provided by being able to run the code in a unit test makes it worth creating a unit test when bug fixing a module, even if those tests do not often identify problems later. As a basic goal, every module should have at least one unit test which will give quick feedback in easy cases without having to wait for the integration tests to complete.

Ensuring correct use of external interfaces

Unit tests can check the way in which external services are run to ensure that they match specifications or are as efficient as possible *even when the final output will not be changed.*

Example:

> Package managers are often far more efficient when installing multiple packages at once rather than each package separately. The final result is the same: the packages are all installed, so the efficiency is difficult to verify through integration tests. By providing a mock package manager and verifying that it is called once, we can build a valuable test for module efficiency.

Another related use is in the situation where an API has versions which behave differently. A programmer working on a new version may change the module to work with the new API version and unintentionally break the old version. A test case which checks that the call happens properly for the old version can help avoid the problem. In this situation it is very important to include version numbering in the test case name (see Naming unit tests below).

Providing specific design tests

By building a requirement for a particular part of the code and then coding to that requirement, unit tests _can_ sometimes improve the code and help future developers understand that code.

Unit tests that test internal implementation details of code, on the other hand, almost always do more harm than good. Testing that your packages to install are stored in a list would slow down and confuse a future developer who might need to change that list into a dictionary for efficiency. This problem can be reduced somewhat with clear test naming so that the future developer immediately knows to delete the test case, but it is often better to simply leave out the test case altogether and test for a real valuable feature of the code, such as installing all of the packages supplied as arguments to the module.

How to unit test Ansible modules

There are a number of techniques for unit testing modules. Beware that most modules without unit tests are structured in a way that makes testing quite difficult and can lead to very complicated tests which need more work than the code. Effectively using unit tests may lead you to restructure your code. This is often a good thing and leads to better code overall. Good restructuring can make your code clearer and easier to understand.

Naming unit tests

Unit tests should have logical names. If a developer working on the module being tested breaks the test case, it should be easy to figure what the unit test covers from the name. If a unit test is designed to verify compatibility with a specific software or API version then include the version in the name of the unit test.

As an example, `test_v2_state_present_should_call_create_server_with_name()` would be a good name, `test_create_server()` would not be.

Use of Mocks

Mock objects (from https://docs.python.org/3/library/unittest.mock.html) can be very useful in building unit tests for special / difficult cases, but they can also lead to complex and confusing coding situations. One good use for mocks would be in simulating an API. As for 'six', the 'mock' python package is bundled with Ansible (use `import units.compat.mock`).

Ensuring failure cases are visible with mock objects

Functions like :meth:`module.fail_json` are normally expected to terminate execution. When you run with a mock module object this doesn't happen since the mock always returns another mock from a function call. You can set up the mock to raise an exception as shown above, or you can assert that these functions have not been called in each test. For example:

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]testing_units_modules.rst`, line 168); *backlink*
>
> Unknown interpreted text role "meth".

```
module = MagicMock()
function_to_test(module, argument)
module.fail_json.assert_not_called()
```

This applies not only to calling the main module but almost any other function in a module which gets the module object.

Mocking of the actual module

The setup of an actual module is quite complex (see Passing Arguments below) and often isn't needed for most functions which use a

module. Instead you can use a mock object as the module and create any module attributes needed by the function you are testing. If you do this, beware that the module exit functions need special handling as mentioned above, either by throwing an exception or ensuring that they haven't been called. For example:

```
class AnsibleExitJson(Exception):
    """Exception class to be raised by module.exit_json and caught by the test case"""
    pass

# you may also do the same to fail json
module = MagicMock()
module.exit_json.side_effect = AnsibleExitJson(Exception)
with self.assertRaises(AnsibleExitJson) as result:
    results = my_module.test_this_function(module, argument)
module.fail_json.assert_not_called()
assert results["changed"] == True
```

API definition with unit test cases

API interaction is usually best tested with the function tests defined in Ansible's integration testing section, which run against the actual API. There are several cases where the unit tests are likely to work better.

Defining a module against an API specification

This case is especially important for modules interacting with web services, which provide an API that Ansible uses but which are beyond the control of the user.

By writing a custom emulation of the calls that return data from the API, we can ensure that only the features which are clearly defined in the specification of the API are present in the message. This means that we can check that we use the correct parameters and nothing else.

Example: in rds_instance unit tests a simple instance state is defined:

```
def simple_instance_list(status, pending):
    return {u'DBInstances': [{u'DBInstanceArn': 'arn:aws:rds:us-east-1:1234567890:db:fakedb',
                              u'DBInstanceStatus': status,
                              u'PendingModifiedValues': pending,
                              u'DBInstanceIdentifier': 'fakedb'}]}
```

This is then used to create a list of states:

```
rds_client_double = MagicMock()
rds_client_double.describe_db_instances.side_effect = [
    simple_instance_list('rebooting', {"a": "b", "c": "d"}),
    simple_instance_list('available', {"c": "d", "e": "f"}),
    simple_instance_list('rebooting', {"a": "b"}),
    simple_instance_list('rebooting', {"e": "f", "g": "h"}),
    simple_instance_list('rebooting', {}),
    simple_instance_list('available', {"g": "h", "i": "j"}),
    simple_instance_list('rebooting', {"i": "j", "k": "l"}),
    simple_instance_list('available', {}),
    simple_instance_list('available', {}),
]
```

These states are then used as returns from a mock object to ensure that the `await` function waits through all of the states that would mean the RDS instance has not yet completed configuration:

```
rds_i.await_resource(rds_client_double, "some-instance", "available", mod_mock,
                     await_pending=1)
assert(len(sleeper_double.mock_calls) > 5), "await_pending didn't wait enough"
```

By doing this we check that the `await` function will keep waiting through potentially unusual that it would be impossible to reliably trigger through the integration tests but which happen unpredictably in reality.

Defining a module to work against multiple API versions

This case is especially important for modules interacting with many different versions of software; for example, package installation modules that might be expected to work with many different operating system versions.

By using previously stored data from various versions of an API we can ensure that the code is tested against the actual data which will be sent from that version of the system even when the version is very obscure and unlikely to be available during testing.

Ansible special cases for unit testing

There are a number of special cases for unit testing the environment of an Ansible module. The most common are documented below, and suggestions for others can be found by looking at the source code of the existing unit tests or asking on the Ansible chat channel or mailing lists. For more information on joining chat channels and subscribing to mailing lists, see :ref:`communication`.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]testing_units_modules.rst, line 280); *backlink*
>
> Unknown interpreted text role "ref".

Module argument processing

There are two problems with running the main function of a module:

- Since the module is supposed to accept arguments on `STDIN` it is a bit difficult to set up the arguments correctly so that the module will get them as parameters.
- All modules should finish by calling either the :meth:`module.fail_json` or :meth:`module.exit_json`, but these won't work correctly in a testing environment.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]testing_units_modules.rst, line 292); *backlink*
>
> Unknown interpreted text role "meth".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]testing_units_modules.rst, line 292); *backlink*

Unknown interpreted text role "meth".

Passing Arguments

To pass arguments to a module correctly, use the `set_module_args` method which accepts a dictionary as its parameter. Module creation and argument processing is handled through the :class:`AnsibleModule` object in the basic section of the utilities. Normally this accepts input on `STDIN`, which is not convenient for unit testing. When the special variable is set it will be treated as if the input came on `STDIN` to the module. Simply call that function before setting up your module:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]testing_units_modules.rst, line 301); *backlink*

Unknown interpreted text role "class".

```
import json
from units.modules.utils import set_module_args
from ansible.module_utils.common.text.converters import to_bytes

def test_already_registered(self):
    set_module_args({
        'activationkey': 'key',
        'username': 'user',
        'password': 'pass',
    })
```

Handling exit correctly

The :meth:`module.exit_json` function won't work properly in a testing environment since it writes error information to `STDOUT` upon exit, where it is difficult to examine. This can be mitigated by replacing it (and :meth:`module.fail_json`) with a function that raises an exception:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]testing_units_modules.rst, line 326); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]testing_units_modules.rst, line 326); *backlink*

Unknown interpreted text role "meth".

```
def exit_json(*args, **kwargs):
    if 'changed' not in kwargs:
        kwargs['changed'] = False
    raise AnsibleExitJson(kwargs)
```

Now you can ensure that the first function called is the one you expected simply by testing for the correct exception:

```
def test_returned_value(self):
    set_module_args({
        'activationkey': 'key',
        'username': 'user',
        'password': 'pass',
    })

    with self.assertRaises(AnsibleExitJson) as result:
        my_module.main()
```

The same technique can be used to replace :meth:`module.fail_json` (which is used for failure returns from modules) and for the `aws_module.fail_json_aws()` (used in modules for Amazon Web Services).

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]testing_units_modules.rst, line 353); *backlink*

Unknown interpreted text role "meth".

Running the main function

If you do want to run the actual main function of a module you must import the module, set the arguments as above, set up the appropriate exit exception and then run the module:

```
# This test is based around pytest's features for individual test functions
import pytest
import ansible.modules.module.group.my_module as my_module

def test_main_function(monkeypatch):
    monkeypatch.setattr(my_module.AnsibleModule, "exit_json", fake_exit_json)
    set_module_args({
        'activationkey': 'key',
        'username': 'user',
        'password': 'pass',
    })
    my_module.main()
```

Handling calls to external executables

Module must use :meth:`AnsibleModule.run_command` in order to execute an external command. This method needs to be mocked:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]testing_units_modules.rst, line 382); *backlink*

Unknown interpreted text role "meth".

Here is a simple mock of :meth:`AnsibleModule.run_command` (taken from :file:`test/units/modules/packaging/os/test_rhn_register.py`):

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]testing_units_modules.rst, line 385); *backlink*
>
> Unknown interpreted text role "meth".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]testing_units_modules.rst, line 385); *backlink*
>
> Unknown interpreted text role "file".

```
with patch.object(basic.AnsibleModule, 'run_command') as run_command:
    run_command.return_value = 0, '', ''  # successful execution, no output
        with self.assertRaises(AnsibleExitJson) as result:
            my_module.main()
        self.assertFalse(result.exception.args[0]['changed'])
# Check that run_command has been called
run_command.assert_called_once_with('/usr/bin/command args')
self.assertEqual(run_command.call_count, 1)
self.assertFalse(run_command.called)
```

A Complete Example

The following example is a complete skeleton that reuses the mocks explained above and adds a new mock for :meth:`Ansible.get_bin_path`:

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]testing_units_modules.rst, line 403); *backlink*
>
> Unknown interpreted text role "meth".

```
import json

from units.compat import unittest
from units.compat.mock import patch
from ansible.module_utils import basic
from ansible.module_utils.common.text.converters import to_bytes
from ansible.modules.namespace import my_module


def set_module_args(args):
    """prepare arguments so that they will be picked up during module creation"""
    args = json.dumps({'ANSIBLE_MODULE_ARGS': args})
    basic._ANSIBLE_ARGS = to_bytes(args)


class AnsibleExitJson(Exception):
    """Exception class to be raised by module.exit_json and caught by the test case"""
    pass


class AnsibleFailJson(Exception):
    """Exception class to be raised by module.fail_json and caught by the test case"""
    pass


def exit_json(*args, **kwargs):
    """function to patch over exit_json; package return data into an exception"""
    if 'changed' not in kwargs:
        kwargs['changed'] = False
    raise AnsibleExitJson(kwargs)


def fail_json(*args, **kwargs):
    """function to patch over fail_json; package return data into an exception"""
    kwargs['failed'] = True
    raise AnsibleFailJson(kwargs)


def get_bin_path(self, arg, required=False):
    """Mock AnsibleModule.get_bin_path"""
    if arg.endswith('my_command'):
        return '/usr/bin/my_command'
    else:
        if required:
            fail_json(msg='%r not found !' % arg)


class TestMyModule(unittest.TestCase):

    def setUp(self):
        self.mock_module_helper = patch.multiple(basic.AnsibleModule,
                                                 exit_json=exit_json,
                                                 fail_json=fail_json,
                                                 get_bin_path=get_bin_path)
        self.mock_module_helper.start()
        self.addCleanup(self.mock_module_helper.stop)

    def test_module_fail_when_required_args_missing(self):
        with self.assertRaises(AnsibleFailJson):
            set_module_args({})
            my_module.main()


    def test_ensure_command_called(self):
        set_module_args({
            'param1': 10,
            'param2': 'test',
        })
```

```
        with patch.object(basic.AnsibleModule, 'run_command') as mock_run_command:
            stdout = 'configuration updated'
            stderr = ''
            rc = 0
            mock_run_command.return_value = rc, stdout, stderr  # successful execution

            with self.assertRaises(AnsibleExitJson) as result:
                my_module.main()
            self.assertFalse(result.exception.args[0]['changed']) # ensure result is changed

        mock_run_command.assert_called_once_with('/usr/bin/my_command --value 10 --name test')
```

Restructuring modules to enable testing module set up and other processes

Often modules have a `main()` function which sets up the module and then performs other actions. This can make it difficult to check argument processing. This can be made easier by moving module configuration and initialization into a separate function. For example:

```
argument_spec = dict(
    # module function variables
    state=dict(choices=['absent', 'present', 'rebooted', 'restarted'], default='present'),
    apply_immediately=dict(type='bool', default=False),
    wait=dict(type='bool', default=False),
    wait_timeout=dict(type='int', default=600),
    allocated_storage=dict(type='int', aliases=['size']),
    db_instance_identifier=dict(aliases=["id"], required=True),
)

def setup_module_object():
    module = AnsibleAWSModule(
        argument_spec=argument_spec,
        required_if=required_if,
        mutually_exclusive=[['old_instance_id', 'source_db_instance_identifier',
                             'db_snapshot_identifier']],
    )
    return module

def main():
    module = setup_module_object()
    validate_parameters(module)
    conn = setup_client(module)
    return_dict = run_task(module, conn)
    module.exit_json(**return_dict)
```

This now makes it possible to run tests against the module initiation function:

```
def test_rds_module_setup_fails_if_db_instance_identifier_parameter_missing():
    # db_instance_identifier parameter is missing
    set_module_args({
        'state': 'absent',
        'apply_immediately': 'True',
     })

    with self.assertRaises(AnsibleFailJson) as result:
        my_module.setup_json
```

See also `test/units/module_utils/aws/test_rds.py`

Note that the `argument_spec` dictionary is visible in a module variable. This has advantages, both in allowing explicit testing of the arguments and in allowing the easy creation of module objects for testing.

The same restructuring technique can be valuable for testing other functionality, such as the part of the module which queries the object that the module configures.

Traps for maintaining Python 2 compatibility

If you use the `mock` library from the Python 2.6 standard library, a number of the assert functions are missing but will return as if successful. This means that test cases should take great care *not* use functions marked as _new_ in the Python 3 documentation, since the tests will likely always succeed even if the code is broken when run on older versions of Python.

A helpful development approach to this should be to ensure that all of the tests have been run under Python 2.6 and that each assertion in the test cases has been checked to work by breaking the code in Ansible to trigger that failure.

> **Warning**
>
> Maintain Python 2.6 compatibility
>
> Please remember that modules need to maintain compatibility with Python 2.6 so the unittests for modules should also be compatible with Python 2.6.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]testing_units_modules.rst, line 565)**
>
> Unknown directive type "seealso".
>
> ```
> .. seealso::
>
>    :ref:`testing_units`
>        Ansible unit tests documentation
>    :ref:`testing_running_locally`
>        Running tests locally including gathering and reporting coverage data
>    :ref:`developing_modules_general`
>        Get started developing a module
>    `Python 3 documentation - 26.4. unittest â€” Unit testing framework <https://docs.python.org/3/library/unittest.ht
>        The documentation of the unittest framework in python 3
>    `Python 2 documentation - 25.3. unittest â€” Unit testing framework <https://docs.python.org/3/library/unittest.ht
>        The documentation of the earliest supported unittest framework - from Python 2.6
>    `pytest: helps you write better programs <https://docs.pytest.org/en/latest/>`_
>        The documentation of pytest - the framework actually used to run Ansible unit tests
>    `Development Mailing List <https://groups.google.com/group/ansible-devel>`_
>        Mailing list for development topics
>    `Testing Your Code (from The Hitchhiker's Guide to Python!) <https://docs.python-guide.org/writing/tests/>`_
>        General advice on testing Python code
>    `Uncle Bob's many videos on YouTube <https://www.youtube.com/watch?v=QedpQjxBPMA&list=PLlu0CT-JnSasQzGrGzddSczJQQU
>        Unit testing is a part of the of various philosophies of software development, including
>        Extreme Programming (XP), Clean Coding.  Uncle Bob talks through how to benefit from this
> ```

```
`"Why Most Unit Testing is Waste" <https://rbcs-us.com/documents/Why-Most-Unit-Testing-is-Waste.pdf>`_
    An article warning against the costs of unit testing
`'A Response to "Why Most Unit Testing is Waste"' <https://henrikwarne.com/2014/09/04/a-response-to-why-most-unit-
    An response pointing to how to maintain the value of unit tests
```