relaxdiego

Writing Ansible Modules Part 2 - Write Your First Module

28 Sep 2016 : 12 minute read

Co-Written by Andreas Hubert

This is part 2 of a series of articles. For other parts, see the introductory article.

It’s Just Somebody’s Computer

Let’s write a module for a fictitious cloud provider named Somebody’s Computer. First, let’s create our module’s subdir:

```
$ mkdir cloud/somebodyscomputer
$ touch cloud/somebodyscomputer/__init__.py
```

From now on, I’ll refer to this as your **module dir**.

Let’s Kick The Tires for a Bit

First, let’s create a topic branch from the HEAD of devel and work there. Working on a topic branch has its advantages in that, should new changes be added to upstream/devel, all you have to do is fetch those and then rebase your topic branch on top of it.

Let’s create our topic branch:

```
$ git checkout -b test_branch devel
```

Let’s use the simple example from the Module Development Page, which we just slightly modify in order to work with the current Ansible develop branch, just to get familiar with the terrain a bit. In your **module dir**, create a file called `timetest.py` with the following content:

```
#!/usr/bin/python

import datetime
import json

def show_time():
    date = str(datetime.datetime.now())
    print json.dumps({
        "time" : date
    })

show_time()
```

You just created your first module! At this point, we can create a playbook that uses your timetest module and then execute it with ansible-playbook. But why when the Ansible repo provides a convenient script that allows you to bypass all that! So from your **ansible repo**, run:

```
$ hacking/test-module -m <path to module dir>/timetest.py
```

This should give you an output similar to the following:

```
* including generated source, if any, saving to: ~/.ansible_module_generated
***********************************
RAW OUTPUT
{"time": "2016-06-09 11:00:01.445100"}


***********************************
PARSED OUTPUT
{
    "time": "2016-06-09 11:00:01.445100"
}
```

What just happened is that the test-module script executed your module without loading all of ansible. This is a nice way to quickly do a sort-of-end-to-end test of your module after you've written your unit tests. I **would not** recommend using it exclusively as your testing strategy. It's best used alongside unit tests and `validate-modules` which we'll use next.

Run:

```
$ test/sanity/validate-modules/validate-modules <path to your first module dir>
```

This should get you the following errors:

```
============================================================================
<path to first module dir>/timetest.py
============================================================================
<path to first module dir>/timestamp.py:0:0: E301 No DOCUMENTATION provided
<path to first module dir>/timestamp.py:0:0: E310 No EXAMPLES provided
<path to first module dir>/timestamp.py:0:0: E314 No ANSIBLE_METADATA provided
<path to first module dir>/timestamp.py:0:0: E103 Did not find a call to main
<path to first module dir>/timestamp.py:0:0: E201 Did not find a module_utils import
<path to first module dir>/timestamp.py:0:0: E105 GPLv3 license header not found
```

Ignore those errors for now while we're still kicking the tires.

Let's Write A Real(-ish) Module!

Let's start with a clean slate, run:

```
$ git reset --hard
```

Next, let's install some Python packages needed by our tests. From your **ansible repo**, run one of the following:

If you're working on Ansible code before 2.4:

```
$ pip install -r test/utils/tox/requirements.txt

NOTE: If you're developing on Python 3.0+, use requirements-py3.txt instead
```

If you're working on Ansible code version 2.4 or later

```
$ pip install -r test/runner/requirements/units.txt \
    -r test/runner/requirements/coverage.txt
```

Next, let's write a module that fetches a resource pointed to by a URL and then writes it to disk. So in our **ansible repo**, create a file at `cloud/somebodyscomputer/firstmod.py` with the following content:

```
1    #!/usr/bin/python
2    # Make coding more python3-ish
3    from __future__ import (absolute_import, division)
4    __metaclass__ = type
5
6    from ansible.module_utils.basic import AnsibleModule
7
8
9    def save_data(mod):
10       raise NotImplementedError
11
12
13   def main():
14       mod = AnsibleModule(
15           argument_spec=dict(
16               url=dict(required=True),
17               dest=dict(required=False, default="/tmp/firstmod")
18           )
19       )
20
21       save_data(mod)
22
23
24   if __name__ == '__main__':
25       main()
```

Let's discuss line by line:

- **Lines 1 to 4** sets up some things to make our code more or less behave well in Python 3
- **Lines 24 to 25** executes the `main` function
- **Lines 13 to 19** instantiates `AnsibleModule`, defining the arguments accepted by the module.

- **Line 21** calls the function that actually does the work. Passing the AnsibleModule instance to it.
- **Lines 9 to 10** defines our worker function which just raises an exception for now.

This is the generally accepted structure of a module. Specifically, the `main()` function should just instantiate `AnsibleModule` and then pass that to another function that will do the actual work. Now because `main()` is very thin, unit testing it is pointless since the test, should we write it, will only end up looking almost like `main()` and that's not very useful. What we really want to test is `save_data()`.

WARNING: Here Be (Testing) Dragons!

I expect that you already know how to write good tests and mocks because I don't have time to teach you that. If you don't, you might still be able to follow along and make out a few things but testing know-how will go a long way in these parts.

If you're confident with your mad testing skillz but your mocking-fu is a bit rusty, I will have to ask you to read Mocking Objects in Python. It's a quick 5~6-minute read.

Let's Write the Test First

From your **ansible repo**, create a unit test directory for your module:

```
$ mkdir -p test/units/modules/cloud/somebodyscomputer
```

IMPORTANT: Make sure you run the above command from the root of your **ansible repo**.

On With the Tests

Let's make `save_data()` actually do some work. We'll design it to fetch the resource and then write it to disk. First, since we're going to be using nose as our test framework, we have to ensure that every subdirectory in the following path has an `__init__.py`, otherwise nose will not load our tests. Go ahead and make sure there's that file in every directory in this path in your **ansible repo**:

```
find test/units/modules/ -type d -exec touch {}/__init__.py  \;
```

Next create `test/units/modules/cloud/somebodyscomputer/test_firstmod.py` as follows:

```
# Make coding more python3-ish
from __future__ import (absolute_import, division)
__metaclass__ = type

```

```
5    from ansible.compat.tests import unittest
6    from ansible.compat.tests.mock import call, create_autospec, patch, mock_open
7    from ansible.module_utils.basic import AnsibleModule
8
9    from ansible.modules.cloud.somebodyscomputer import firstmod
10
11
12   class TestFirstMod(unittest.TestCase):
13
14       @patch('ansible.modules.cloud.somebodyscomputer.firstmod.write')
15       @patch('ansible.modules.cloud.somebodyscomputer.firstmod.fetch')
16       def test__save_data__happy_path(self, fetch, write):
17           # Setup
18           mod_cls = create_autospec(AnsibleModule)
19           mod = mod_cls.return_value
20           mod.params = dict(
21               url="https://www.google.com",
22               dest="/tmp/firstmod.txt"
23           )
24
25           # Exercise
26           firstmod.save_data(mod)
27
28           # Verify
29           self.assertEqual(1, fetch.call_count)
30           expected = call(mod.params["url"])
31           self.assertEqual(expected, fetch.call_args)
32
33           self.assertEqual(1, write.call_count)
34           expected = call(fetch.return_value, mod.params["dest"])
35           self.assertEqual(expected, write.call_args)
36
37           self.assertEqual(1, mod.exit_json.call_count)
38           expected = call(msg="Data saved", changed=True)
39           self.assertEqual(expected, mod.exit_json.call_args)
```

- **Lines 5 to 9** - Import Python modules that we're going to need for our test
- **Lines 14 and 15** - Patch two new methods in our module, `write` and `fetch`
- **Lines 18 to 23** - Set up a mock of AnsibleModule that we will pass on to `save_data()`. We expect the function to get the arguments from the AnsibleModule's `param` attribute, so we stubbed that in line 20.
- **Line 26** - Exercise the code
- **Lines 29 to 31** - Verify that it called `fetch` properly
- **Lines 33 to 35** - Verify that it called `write` properly
- **Lines 37 to 39** - Verify that it called `AnsibleModule.exit_json` properly

Let's execute this test. From the **ansible repo**, run:

```
$ nosetests --doctest-tests -v test/units/modules/cloud/somebodyscomputer/test_firstmod.py
```

This should get you an error because we haven't written any code that satisfies the test yet!

SIDEBAR: That's a Lot of Typing Just to Run One Test!

Well, if you set up your editor properly, you can run it with as few as two keystrokes! Don't know how to do it, check out what I did.

Let's Write Code to Pass the Test

```python
#!/usr/bin/python
# Make coding more python3-ish
from __future__ import (absolute_import, division)
__metaclass__ = type

from ansible.module_utils.basic import AnsibleModule


def fetch(url):
    raise NotImplementedError


def write(data, dest):
    raise NotImplementedError


def save_data(mod):
    data = fetch(mod.params["url"])
    write(data, mod.params["dest"])
    mod.exit_json(msg="Data saved", changed=True)


def main():
    mod = AnsibleModule(
        argument_spec=dict(
            url=dict(required=True),
            dest=dict(required=False, default="/tmp/firstmod")
        )
    )

    save_data(mod)


if __name__ == '__main__':
    main()
```

Run the test again to see it pass:

```
ansible.modules.core.test.unit.cloud.somebodyscomputer.test_firstmod.TestFirstMod.test__save_data__
happy_path ... ok

----------------------------------------------------------------------
Ran 1 test in 0.024s

OK
```

Testing for Failures

The happy path is always the first path that I test but I don't stop there. In this context, I also test for when `fetch()` or `write()` fail. The steps are fairly similar to the happy path so I'll leave it to you to see how I did it by looking at the final test and source code.

You Rock!

You made it this far and that deserves a pat on the back. Good job again! Take another breather, then head on over to part 3 where we'll continue implementing our first module. Alternatively, you can go back to the the introduction if you want to jump ahead to other parts.