PRACTICAL PORTS

Developing Custom Ansible Modules

BY BENEDICT REUSCHLING

Ansible offers a lot of different modules and a typical user makes use of them without the need to ever write their own due to the sheer size of available modules. Even if the necessary functionality is not available in the `ansible.builtin` modules, the Ansible Galaxy offers plenty of third party modules from enthusiasts that extend the module count even more.

When the desired functionality is not covered by a single module or a combination of them, then you have to develop your own. Developers can chose to keep custom models local without having to publish them on the Internet or without Ansible Galaxy using them. Modules are commonly developed in Python, but other programming languages are possible when the module is not planned for submission into the official Ansible ecosystem.

Developers can chose to keep custom models local without having to publish them on the Internet or without Ansible Galaxy using them.

To test the module, install the `ansible-core` package, which helps by providing common code that Ansible uses internally. The custom module can then piggy-back onto much of the core Ansible functionality that existing modules use and is both reliable and stable.

Example Module Using Shell Programming

We'll start with a simple example to understand the basics. Later, we will extend it to use Python for more functionality.

Description of our custom module: Our custom module called `touch` checks for a file in `/tmp` called `BSD.txt`. If it exists, the module returns `true` (state unchanged). If it does not exist, it creates that (empty) file and returns `state: changed`.

Custom modules are in a library directory next to the playbook that uses the module. Create that directory using mkdir:

```
mkdir library
```

Create a shell script in library that holds the module code:

```
touch library/touch
```

Enter the following code in library/touch as the module logic:

```
1  FILENAME=/tmp/BSD.txt
2  changed=false
3  msg=''
4  if [ ! -f ${FILENAME} ]; then
5      touch ${FILENAME}
6      msg="${FILENAME} created"
7      changed=true
8  fi
9  printf ‘{"changed": "%s", "msg": "%s"}’ "$changed" "$msg"
```

First, we define some variables and set some default values. Line 4 checks if the file does not exist. If that is the case, we let the module create the file and update the `msg` variable. We need to notify Ansible about the changed state, so we return a variable called `changed` along with the updated message in line.

Create a playbook called `touch.yml` at the same location as the `library` directory. It looks like this:

```
---
- hosts: localhost
  gather_facts: false
  tasks:
    - name: Run our custom touch module
      touch:
      register: result

    - debug: var=result
```

Note: We could execute the custom module against any remote nodes, not `localhost` alone. It's easier to test against `localhost` first during development.

Run the playbook like any other we've written before:

```
ansible-playbook touch.yml
```

Running the Example Module

When the file `/tmp/BSD.txt` does not exist, the playbook output is:

```
PLAY [localhost] *****************************************

TASK [Run our custom touch module] ***********************
changed: [localhost]

TASK [debug] *********************************************
ok: [localhost] => {
    "changed": true,
    "result": {
        "failed": false,
        "msg": "/tmp/BSD.txt created"
    }
}
```

When the file **/tmp/BSD.txt** exists (from a previous run), the output is:

```
PLAY [localhost] *****************************************

TASK [Run our custom touch module] ***********************
ok: [localhost]

TASK [debug] *********************************************
ok: [localhost] => {
    "result": {
        "changed": false,
        "failed": false,
        "msg": ""
    }
}
```

Custom Modules in Python

What are the benefits of writing a module in Python, like the rest of the **ansible.builtin** modules? One benefit is that we can use the existing parsing library for the module parameters without having to reinvent our own. It's difficult in shell to define the name of each parameter in our own module. In Python, we can teach the module to accept some parameters as optional and require others as mandatory. Data types define what kind of inputs the module user must provide for each parameter. For example, a **dest**: parameter should be a path data type rather than an integer. Ansible provides some handy functionality to include in our script so that we can focus on our module's core functionality.

If I use a feature only twice per year, I may need to look it up again.

The Ansiballz Framework

Modern Ansible modules use the Ansiballz framework. Unlike the Module Replacer, which were used by Ansible versions before 2.1, it uses real Python imports from **ansible/module_utils** instead of preprocessing the module.

Module Functionality: Ansiballz constructs a zip file. Contents:

- the module file
- **ansible/module_utils** files imported by the module
- boilerplate for the module parameters

The zip file is Base64 encoded and wrapped into a small Python script for decoding it. Next, Ansible copies it into the temp directory of the target node. When executing, the Ansible module script extracts the zip file and places itself in the temp dir, too. It then sets the PTHONPATH to find Python modules inside the zip and imports the Ansible module under the special **name**. Python then thinks it executes a regular script rather than importing a module. This allows Ansible to run both the wrapper script and the module's code in a single Python copy on the target host.

Creating the Python Module

To create a module, use a `venv` or `virtualenv` for the development part. We start like before with a `library` directory where we create a new `hello.py` module with this content:

```
#!/usr/bin/env python3

from ansible.module_utils.basic import *

def main():
    module = AnsibleModule(argument_spec={})
    response = {"hello": "world!"}
    module.exit_json(changed=False, meta=response)

if name == "__main__":
    main()
```

`import` imports the Ansiballz framework to construct modules. It includes code constructs like argument parsing, file operations, and formatting return values as JSON.

Executing the Python Module from a Playbook

```
---
- hosts: localhost
  gather_facts: false
  tasks:
    - name: Testing the Python module
      hello:
      register: result

    - debug: var=result
```

Again, we run the playbook like this: `ansible-playbook hello.yml`

```
PLAY [localhost] ******************************************

TASK [Testing the Python module] *************************
ok: [localhost]

TASK [debug] *********************************************
ok: [localhost] => {
    "result": {
        "changed": false,
        "failed": false,
        "meta": {
            "hello": "world!"
        }
    }
}
```

Defining Module Parameters

The modules we used had taken parameters like **`path:`**, **`src:`**, or **`dest:`** to control the behavior of the module. Some of these parameters are essential for the module to function properly, while others were optional. In our own module, we want to control what parameters we take overall and which are required. Defining the data type makes our module robust against incorrect inputs.

The **`argument_spec`** provided to **`AnsibleModule`** defines the supported module arguments, as well as their type, defaults, and more.

Example parameter definition:

```
parameters = {
    'name': {"required": True, "type": 'str'},
    'age': {"required": False, "type": 'int', "default": 0},
    'homedir': {"required": False, "type": 'path'}
}
```

The required parameter **`name`** is of type string. Both **`age`** (an integer) and **`homedir`** (a path) are optional and if not defined, sets **`age`** to 0 by default. A new module that uses these parameter definitions calculates the result from passing two numbers and an optional math operator. When not provided, we assume an addition as default parameter. Create a new python file in **`library`** called **`calc.py`**:

```
#!/usr/bin/env python3
from ansible.module_utils.basic import AnsibleModule

def main():
    parameters = {
        "number1": {"required": True, "type": "int"},
        "number2": {"required": True, "type": "int"},
        "math_op": {"required": False, "type": "str", "default": "+"},
    }

    module = AnsibleModule(argument_spec=parameters)

    number1 = module.params["number1"]
    number2 = module.params["number2"]
    math_op = module.params["math_op"]

    if math_op == "+":
        result = number1 + number2

    output = {
        "result": result,
    }

    module.exit_json(changed=False, **output)

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

The Playbook for the Module

```
---
- hosts: localhost
  gather_facts: false
  tasks:
    - name: Testing the calc module
      calc:
        number1: 4
        number2: 3
      register: result

    - debug: var=result
```

The `calc` module optionally takes a parameter `math_op`, but since we defined a default action (+) for it, the user can omit it in the playbook or on the commandline. The task that runs the module must specify the required parameters or the playbook will fail to execute.

Running the calc Module

The relevant output of the playbook execution is below:

```
ok: [localhost] => {
    "result": {
        "changed": false,
        "failed": false,
        "result": 7
    }
}
```

We extend the example to properly handle +, -, *, /. The module returns `false` when it gets a `math_op` that is is different from the ones defined. Also, handling division by zero by returning "Invalid Operation" is a classic assignment for students since the dawn of time. I need to properly learn Python one day, but until then, my solution looks like this:

```
#!/usr/bin/env python3
    from ansible.module_utils.basic import AnsibleModule

def main():
    parameters = {
        "number1": {"required": True, "type": "int"},
        "number2": {"required": True, "type": "int"},
        "operation": {"required": False, "type": "str", "default": "+"},
}

    module = AnsibleModule(argument_spec=parameters)

    number1 = module.params["number1"]
    number2 = module.params["number2"]
    operation = module.params["operation"]
    result = ""
    if operation == "+":
        result = number1 + number2
    elif operation == "-":
        result = number1 - number2
    elif operation == "*":
        result = number1 * number2
    elif operation == "/":
        if number2 == 0:
            module.fail_json(msg="Invalid Operation")
        else:
            result = number1 / number2
    else:
        result = False

    output = {
        "result": result,
    }

    module.exit_json(changed=False, **output)

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

Testing our extended module is straightforward. Here is the test for division by zero:

```
---
- hosts: localhost
  gather_facts: false
  tasks:
    - name: Testing the calc module
      calc:
        number1: 4
        number2: 0
        map_op: '/'
      register: result

    - debug: var=result
```

Which results in the following expected output:

```
TASK [Testing the calc module] **********************************************
fatal: [localhost]: FAILED! => {"changed": false, "msg": "Invalid Operation"}
```

Conclusion

With these basics, its easy to get started on a custom module. Bear in mind that these modules need to run on different operating systems. Add extra checks to find out the availability of certain commands or let your module outright refuse to run in certain environments. Be as compatible as possible to increase the module's popularity and usefulness. There are not a lot of BSD-specific modules available. How about adding a bhyve module, or one that manages boot environments, the pf firewall or `rc.conf` entries? Plenty of options await the intrepid developer with a background in both Ansible and Python.

References:

- Ansible module architecture

BENEDICT REUSCHLING is a documentation committer in the FreeBSD project and member of the documentation engineering team. In the past, he served on the FreeBSD core team for two terms. He administers a big data cluster at the University of Applied Sciences, Darmstadt, Germany. He's also teaching a course "Unix for Developers" for undergraduates. Benedict is one of the hosts of the weekly bsdnow.tv podcast.