Writing Ansible Modules in Bash

License

Introduction

One of the strengths of Ansible is that you can write modules for it in any language. Whilst there are advantages to writing modules in Python, such as a bunch of helper routines, the only thing you miss out on by using another language is the ability for your module to support dry run mode.

In this guide I'm going to look at writing modules in bash, although the same principles can be used in any scripting language.

Why Bash?

Although bash lacks a lot of features compared to languages such as Python, Ruby or Perl, a large number of people have some experience with it. Turning a bash script into an Ansible module doesn't take much additional knowledge.

Input

Ansible will run your module and pass it one argument: the name of a file containing the module arguments you specified in your playbook. For example, if you had

```
bashmod: dest=/tmp/hello state=present
```

you would get passed the name of a file with the contents

```
dest=/tmp/hello state=preset
```

You can get the name of this file from the `$1` variable and, handily, this file is valid bash syntax so you can `source` it to convert these arguments to bash variables, e.g.

```
source $1
```

This will create the variables `$dest` and `$state`.

Output

The output from your module must be in JSON format. If you return any other output, Ansible will treat it as a failure. This means you need to capture stdout and stderr from any commands you run.

Ansible looks for the following variables in the output:

- `changed`: Return this if your module was successful. Set it to `true` if it made any changes or `false` if everything was already in the correct state.
- `failed`: set this to `true` if your modules failed. You can set it to `false` if you module worked, or you can leave it out and Ansible will assume it worked.
- `msg`: Return an error message if your module failed. You can also set this to an information message on success.

Any other variables can be returned and will be displayed by Ansible in the output for the task.

For example:

```
echo '{"changed": true, "msg": "Updated stuff"}'
```

The JSON variable names and values must be enclosed in double quotes, with the exception for the values `true` and `false` which don't need quotes.

Because of the double quotes, I've surrounded the whole string in single quotes. This is ok if you're returning fixed strings, but doesn't work if you want to use variables as bash won't do variable expansion in a single quoted string.

One way around this is to escape the double quotes, e.g.

```
echo "{\"changed\": true, \"msg\": \"$msg\"}"
```

I find this hard to read and error prone, so I prefer to use `printf` instead:

```
printf '{"changed": true, "msg": "%s"}' "$msg"
```

Some characters have special meaning to JSON and need to be escaped. If you return any output from a command, you're better off escaping it just in case. There are probably numerous ways to do this, but I like to pipe it through a Python one-liner; if you're using Ansible, you're already going to have Python installed. I don't know who came up with this Python command, so I can't give them attribution; thanks, though!

```
$msg = $(echo "$msg" | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
```

This might be a good thing to put in a function so you can easily use it in multiple places. One thing to be careful of if you're using this method is that the Python `json.dumps` command puts starting and ending double quotes around the string. This means you need to modify the above `printf` command to remove the quotes from around the `%s`, as follows:

```
printf '{"changed": true, "msg": %s}' "$msg"
```

Example module

This is a simple example module which writes some text to a file and can convert it to upper or lower case.

```
#!/bin/bash

function create_file
    {
    if [ -f "$dest" ]; then
        changed="false"
        msg="file already exists"
    else
        echo 'Hello, "world!"' >> $dest
        changed="true"
        msg="file created"
    fi
    contents=$(cat "$dest" 2>&1 | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
    }

function delete_file
    {
    if [ -f "$dest" ]; then
        changed="true"
        msg="file deleted"
        contents=$(cat "$dest" 2>&1 | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
        output=$(rm -f $dest 2>&1 | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
        if [ $? -ne 0 ]; then
            printf '{"failed": "true", "msg": "error deleting file", "output": %s}' "$output"
            exit 1
        fi
    else
        changed="false"
        msg="file not present"
        contents='""'
    fi
    }

function convert_to_upper
    {
    if [ ! -f "$dest" ]; then
        create_file
        msg="$msg, "
    fi
    current=$(cat $dest)
    new=$(echo "$current" | tr '[:lower:]' '[:upper:]')
    if [ "$current" = "$new" ]; then
        changed="false"
        msg="${msg}file not changed"
        contents=$(printf "$current" | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
    else
        echo "$new" > $dest
        changed="true"
        msg="${msg}file converted to upper case"
        contents=$(printf "$new" | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
    fi
    }

function convert_to_lower
    {
    if [ ! -f "$dest" ]; then
        create_file
        msg="$msg, "
    fi
    contents=$(ls -l "$dest" 2>&1 | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
    current=$(cat $dest)
    new=$(echo "$current" | tr '[:upper:]' '[:lower:]')
    if [ "$current" = "$new" ]; then
        changed="false"
        msg="${msg}file not changed"
        contents=$(printf "$current" | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
    else
        echo "$new" > $dest
        changed="true"
        msg="${msg}file converted to lower case"
        contents=$(printf "$new" | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
    fi
    }

source $1

if [ -z "$dest" ]; then
    printf '{"failed": "true", "msg": "missing required arguments: dest"}'
    exit 1
fi
if [ -z "$state" ]; then
    printf '{"failed": "true", "msg": "missing required arguments: state"}'
    exit 1
fi

changed="false"
msg=""
contents=""

case $state in
    present)
        create_file
        ;;
    absent)
        delete_file
        ;;
    upper)
        convert_to_upper
        ;;
    lower)
        convert_to_lower
        ;;
    *)
        printf '{"failed": true, "msg": "invalid state: %s"}' "$state"
        exit 1
        ;;
esac

printf '{"changed": "%s", "msg": "%s", "contents": %s}' "$changed" "$msg" "$contents"

exit 0
```

Testing Your Module

Testing Using `test-module`

If you've installed Ansible from the GitHub repository, you'll have a directory called "hacking". In there is a program called `test-module`.

`test-module` will properly handle your module outputting lines other than JSON and will display them in the `RAW OUTPUT` section. Your JSON output will be displayed under `PARSED OUTPUT`.

If you've install Ansible via a package, you may not have this directory; you'll either need to do a clone from GitHub or use one of the other methods below to test your module. However, `test-module` is the best option.

Setting up your environment to use it:

- Change directory into the `ansible/hacking` directory.
- Run `source env-setup`. This will update your environment variable to enable Anduble to run out of this directory.
- Add the current directory to your path so you can run `test-module` without having to specify its path all the time:

  ```
  export PATH=$PATH:`pwd`
  ```

Example Run: create the file

This example creates the file `test.txt` with the default text of `Hello, "world!"`

```
$ test-module -m bashmod -a 'dest=test.txt state=present'
* including generated source, if any, saving to: /Users/paul/.ansible_module_generated
* this may offset any line numbers in tracebacks/debuggers!
***********************************
RAW OUTPUT
{"changed": "true", "msg": "file created", "contents": "Hello, \"world!\"\n"}
***********************************
PARSED OUTPUT
{
    "changed": "true",
    "contents": "Hello, \"world!\"\n",
    "msg": "file created"
}
```

Example Run: file already exists

Here's what happens if you rerun the previous example. As the file already exists, it returns `changed: false`, which indicates that it's already in the correct state.

```
$ test-module -m bashmod -a 'dest=test.txt state=present'
* including generated source, if any, saving to: /Users/paul/.ansible_module_generated
* this may offset any line numbers in tracebacks/debuggers!
***********************************
RAW OUTPUT
{"changed": "false", "msg": "file already exists", "contents": "Hello, \"world!\"\n"}
***********************************
PARSED OUTPUT
{
    "changed": "false",
    "contents": "Hello, \"world!\"\n",
    "msg": "file already exists"
}
```

Example Run: convert to upper case

Convert the contents of the file to upper case.

```
$ test-module -m bashmod -a 'dest=test.txt state=upper'
* including generated source, if any, saving to: /Users/paul/.ansible_module_generated
* this may offset any line numbers in tracebacks/debuggers!
***********************************
RAW OUTPUT
{"changed": "true", "msg": "file converted to uppercase", "contents": "HELLO, \"WORLD!\""}
***********************************
PARSED OUTPUT
{
    "changed": "true",
    "contents": "HELLO, \"WORLD!\"",
    "msg": "file converted to upper case"
}
```

Example Run: convert to lower case

Convert the contents of the file to lower case.

```
$ test-module -m bashmod -a 'dest=test.txt state=lower'
* including generated source, if any, saving to: /Users/paul/.ansible_module_generated
* this may offset any line numbers in tracebacks/debuggers!
***********************************
RAW OUTPUT
{"changed": "true", "msg": "file converted to uppercase", "contents": "hello, \"world!\""}
***********************************
PARSED OUTPUT
{
    "changed": "true",
    "contents": "hello, \"world!\"",
    "msg": "file converted to lower case"
}
```

Example Run: delete the file

```
$ test-module -m bashmod -a 'dest=test.txt state=absent'
* including generated source, if any, saving to: /Users/paul/.ansible_module_generated
* this may offset any line numbers in tracebacks/debuggers!
***********************************
RAW OUTPUT
{"changed": "true", "msg": "file deleted", "contents": "hello, \"world!\"\n"}
***********************************
PARSED OUTPUT
{
    "changed": "true",
    "contents": "hello, \"world!\"\n",
    "msg": "file deleted"
}
```

Example Run: create file and convert to upper case

This example specifies the state as `upper` but the file hadn't been created yet. The module creates the file and converts the contents to upper case. Both actions are reflected in the `msg` variable that's returned.

```
$ test-module -m bashmod -a 'dest=test.txt state=upper'
* including generated source, if any, saving to: /Users/paul/.ansible_module_generated
* this may offset any line numbers in tracebacks/debuggers!
***********************************
RAW OUTPUT
{"changed": "true", "msg": "file created, file converted to uppercase", "contents": "HELLO, \"WORLD!\""}
***********************************
PARSED OUTPUT
{
    "changed": "true",
    "contents": "HELLO, \"WORLD!\"",
    "msg": "file created, file converted to uppercase"
}
```

Testing Using Ansible Command Line

You can use the `ansible` command to run your module. If your module outputs anything other than JSON, it will be treated as a failure.

```
$ ansible -c local -i 'localhost,'  -M . -m bashmod -a 'dest=test.txt state=present' all
localhost | success >> {
    "changed": "true",
    "contents": "Hello, \"world!\"\n",
    "msg": "file created"
}
```

Testing Using bash

You can execute your module directly by writing the arguments to a file in key=value pairs on a single line. Run the module and pass the name of this file.

This will not check that your output is valid JSON format, but can be handy if you want to add debugging statements to the module.

```
$ echo 'dest=test.txt state=present' > args
$ bash bashmod args
{"changed": "false", "msg": "file already exists", "contents": "Hello, \"world!\"\n"}
```

You can run your module with the `-x` option to trace its execution.

```
$ echo 'dest=test.txt state=present' > args
$ bash -x bashmod args
+ source args
++ dest=test.txt
++ state=present
+ '[' -z test.txt ']'
+ '[' -z present ']'
+ changed=false
+ msg=
+ contents=
+ case $state in
+ create_file
+ '[' -f test.txt ']'
+ echo 'Hello, "world!"'
+ changed=true
+ msg='file created'
++ cat test.txt
++ python -c 'import json,sys; print json.dumps(sys.stdin.read())'
+ contents='"Hello, \"world!\"\n"'
+ printf '{"changed": "%s", "msg": "%s", "contents": %s}' true 'file created' '"Hello, \"world!\"\n"'
{"changed": "true", "msg": "file created", "contents": "Hello, \"world!\"\n"}+ exit 0
```