Lab 10. Troubleshooting and Testing Strategies

There are a bunch of different ways to prevent or mitigate a bug in Ansible playbooks. In this lab, we will cover the following topics:

- Digging into playbook execution problems
- Using host facts to diagnose failures
- Testing with a playbook
- Using check mode
- Solving host connection issues
- Passing working variables via the CLI
- Limiting the host's execution
- Flushing the code cache
- Checking for bad syntax

Digging into playbook execution problems

There are cases where an Ansible execution will interrupt. Many things can cause these situations.

The single most frequent cause of problems I've found while executing Ansible playbooks is the network. Since the machine that is issuing the commands and the one that is performing them are usually linked through the network, a problem in the network will immediately show itself as an Ansible execution problem.

Sometimes, and this is particularly true for some modules, such as [shell] or [command], the return code is non-zero, even though the execution was successful. In those cases, you can ignore the error by using the following line in your module:

```
ignore_errors: yes
```

For instance, if you run the [/bin/false] command, it will always return [1]. To execute this in a playbook so that you can avoid it blocking there, you can write something like the following:

```
- name: Run a command that will return 1
  command: /bin/false
  ignore_errors: yes
```

As we have seen, [/bin/false] will always return [1] as return code, but we still managed to go forward in the execution. Be aware that this is a particular case, and often, the best approach is to fix your application so that you're following UNIX standards and return [0] if the application runs appropriately, instead of putting a workaround in your Playbooks.

Next, we will talk more about the methods we can use to diagnose Ansible execution problems.

Using host facts to diagnose failures

Some execution failures derive from the state of the target machine. The most common problem of this kind is the case where Ansible expects a file or variable to be present, but it's not.

Sometimes, it can be enough to print the machine facts to find the problem.

To do so, we need to create a simple playbook, called [print_facts.yaml], which contains the following content:

```
---
- hosts: target_host
  tasks:
    - name: Display all variables/facts known for a host
      debug:
        var: hostvars[inventory_hostname]
```

This technique will give you a lot of information about the state of the target machine during Ansible execution.

Testing with a playbook

One of the most complex things in the IT field is not creating software and systems, but debugging them when they have problems. Ansible is not an exception. No matter how good you are at creating Ansible playbooks, sooner or later, you'll find yourself debugging a playbook that is not behaving as you thought it would.

The simplest way of performing basic tests is to print out the values of variables during execution. Let's learn how to do this with Ansible, as follows:

1. First of all, we need a playbook called [debug.yaml] with the following content:

```
---
- hosts: localhost
  tasks:
    - shell: /usr/bin/uptime
      register: result
    - debug:
        var: result
```

2. Run it with the following command:

```
$ ansible-playbook debug.yaml
```

You will receive an output similar to the following:

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

TASK [Gathering Facts]
*********************************************************************
ok: [localhost]

TASK [shell]
*************************************************************************************
changed: [localhost]

TASK [debug]
*************************************************************************************
ok: [localhost] => {
    "result": {
        "changed": true,
```

```
        "cmd": "/usr/bin/uptime",
        "delta": "0:00:00.003461",
        "end": "2019-06-16 11:30:51.087322",
        "failed": false,
        "rc": 0,
        "start": "2019-06-16 11:30:51.083861",
        "stderr": "",
        "stderr_lines": [],
        "stdout": " 11:30:51 up 40 min, 1 user, load average: 1.11, 0.73, 0.53",
        "stdout_lines": [
            " 11:30:51 up 40 min, 1 user, load average: 1.11, 0.73, 0.53"
        ]
    }
}

PLAY RECAP
*********************************************************************************

localhost : ok=3 changed=1 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

In the first task, we used the [command] module to execute the [uptime] command and saved its output in the [result] variable. Then, in the second task, we used the [debug] module to print the content of the [result] variable.

The [debug] module is the module that allows you to print the value of a variable (by using the [var] option) or a fixed string (by using the [msg] option) during Ansible's execution.

The [debug] module also provides the [verbosity] option. Let's say you change the playbook in the following way:

```
---
- hosts: localhost
  tasks:
    - shell: /usr/bin/uptime
      register: result
    - debug:
       var: result
       verbosity: 2
```

Now, if you try to execute it in the same way you did previously, you will notice that the debug step won't be executed and that the following line will appear in the output instead:

```
TASK [debug]
*********************************************************************************
skipping: [localhost]
```

This is because we set the minimum required [verbosity] to [2], and by default, Ansible runs with a [verbosity] of [0].

To see the result of using the debug module with this new playbook, we will need to run a slightly different command:

```
$ ansible-playbook debug2.yaml -vv
```

By putting two [-v] options in the command line, we will be running Ansible with [verbosity] of [2]. This will not only affect this specific module but all the modules (or Ansible itself) that are set to behave differently at different debug levels.

Now that you have learned how to test with a playbook, let's learn how to use check mode.

Using check mode

Although you might be confident in the code you have written, it still pays to test it before running it for real in a production environment. In such cases, it is a good idea to be able to run your code, but with a safety net in place. This is what check mode is for. Follow these steps:

1. First of all, we need to create an easy playbook to test this feature. Let's create a playbook called [check-mode.yaml] that contains the following content:

```
---
- hosts: localhost
  tasks:
    - name: Touch a file
      file:
        path: /tmp/myfile
        state: touch
```

2. Now, we can run the playbook in the check mode by specifying the [--check] option in the invocation:

```
$ ansible-playbook check-mode.yaml --check
```

This will output everything as if it was really performing the operation, as follows:

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

TASK [Gathering Facts]
*********************************************************************
ok: [localhost]

TASK [Touch a file]
*********************************************************************
ok: [localhost]

PLAY RECAP
*********************************************************************************

localhost : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

However, if you look in [/tmp], you won't find [myfile].

Ansible check mode is usually called a dry run. The idea is that the run won't change the state of the machine and will only highlight the differences between the current status and the status declared in the playbook.

Not all modules support check mode, but all major modules do, and more and more modules are being added at every release. In particular, note that the [command] and [shell] modules do not support it because it is impossible for the module to tell what commands will result in a change, and what won't. Therefore, these modules will always return changed when they're run outside of check mode because they assume a change has been made.

A similar feature to check mode is the [--diff] flag. What this flag allows us to do is track what exactly changed during an Ansible execution. So, let's say we run the same playbook with the following command:

```
$ ansible-playbook check-mode.yaml --diff
```

This will return something like the following:

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

TASK [Gathering Facts]
*********************************************************************
ok: [localhost]

TASK [Touch a file]
*********************************************************************************
--- before
+++ after
@@ -1,6 +1,6 @@
 {
- "atime": 1560693571.3594637,
- "mtime": 1560693571.3594637,
+ "atime": 1560693571.3620908,
+ "mtime": 1560693571.3620908,
     "path": "/tmp/myfile",
- "state": "absent"
+ "state": "touch"
 }

changed: [localhost]

PLAY RECAP
*********************************************************************************

localhost : ok=2 changed=1 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

As you can see, the output says [changed], which means that something was changed (more specifically, the file was created), and in the output, we can see a diff-like output that tells us that the state moved from [absent] to [touch], which means the file was created. [mtime] and [atime] also changed, but this is probably due to how files are created and checked.

Now that you have learned how to use check mode, let's learn how to solve host connection issues.

Solving host connection issues

Ansible is often used to manage remote hosts or systems. To do this, Ansible will need to be able to connect to the remote host, and only after that will it be able to issue commands. Sometimes, the problem is that Ansible is unable to connect to the remote host. A typical example of this is when you try to manage a machine that hasn't booted yet. Being able to quickly recognize these kinds of problems and fix them promptly will help you save a lot of time.

Follow these steps to get started:

1. Let's create a playbook called [remote.yaml] with the following content:

```
---
- hosts: all
  tasks:
    - name: Touch a file
      file:
        path: /tmp/myfile
        state: touch
```

2. We can try to run the [remote.yaml] playbook against a non-existent FQDN, as follows:

```
$ ansible-playbook -i host.example.com, remote.yaml
```

In this case, the output will clearly inform us that the SSH service did not reply in time:

```
PLAY [all]
*********************************************************************************



TASK [Gathering Facts]
*************************************************************************
fatal: [host.example.com]: UNREACHABLE! => {"changed": false, "msg": "Failed to
connect to the host via ssh: ssh: Could not resolve hostname host.example.com: Name or
service not known", "unreachable": true}

PLAY RECAP
*********************************************************************************

host.example.com : ok=0 changed=0 unreachable=1 failed=0 skipped=0 rescued=0 ignored=0
```

There is also the possibility that we'll receive a different error:

```
PLAY [all]
*********************************************************************************



TASK [Gathering Facts]
*************************************************************************
fatal: [host.example.com]: UNREACHABLE! => {"changed": false, "msg": "Failed to
connect to the host via ssh: fale@host.example.com: Permission denied
(publickey,gssapi-keyex,gssapi-with-mic).", "unreachable": true}

PLAY RECAP
*********************************************************************************

host.example.com : ok=0 changed=0 unreachable=1 failed=0 skipped=0 rescued=0 ignored=0
```

In this case, the host did reply, but we don't have enough access to be able to SSH into it.

SSH connections usually fail for one of two reasons:

- The SSH client is unable to establish a connection with the SSH server
- The SSH server refuses the credentials provided by the SSH client

Due to OpenSSH's very high stability and backward compatibility, when the first issue occurs, it's very probable that the IP address or the port is wrong, so the TCP connection isn't feasible. Very rarely, this kind of error occurs in SSH-specific problems. Usually, double-checking the IP and the hostname (if it's a DNS, check that it resolves to the right IP) solves the problem. To investigate this further, you can try performing an SSH connection from the same machine to check if there are problems. For instance, I would do this like so:

```
$ ssh host.example.com -vvv
```

I've taken the hostname from the error itself to ensure that I'm simulating exactly what Ansible is doing. I'm doing this to ensure that I can see all possible logging messages that SSH is able to give me to troubleshoot the problem.

The second problem might be a little bit more complex to debug since it can happen for multiple reasons. One of those is that you are trying to connect to the wrong host and you don't have the credentials for that machine. Another common case is that the username is wrong. To debug it, you can take the [user@host] address that is shown in the error (in my case, [fale@host.example.com]) and use the same command you used previously:

```
$ ssh fale@host.example.com -vvv
```

This should raise the same error that Ansible reported to you, but with much more details.

Now that you have learned how to solve host connection issues, let's learn how to pass working variables via the CLI.

Passing working variables via the CLI

One thing that can help during debugging, and definitely helps for code reusability, is passing variables to playbooks via the command line. Every time your application -- either an Ansible playbook or any kind of application -- receives an input from a third party (a human, in this case), it should ensure that the value is reasonable. An example of this would be to check that the variable has been set and therefore is not an empty string. This is a security golden rule, but should also be applied when the user is trusted since the user might mistype the variable name. The application should identify this and protect the whole system by protecting itself. Follow these steps:

1. The first thing we want to have is a simple playbook that prints the content of a variable. Let's create a playbook called [printvar.yaml] that contains the following content:

```
---
- hosts: localhost
  tasks:
    - debug:
       var: variable
```

2. Now that we have an Ansible playbook that allows us to see if a variable has been set to what we were expecting, let's run it with [variable] declared in the execution statement:

```
$ ansible-playbook printvar.yaml --extra-vars='{"variable": "Hello, World!"}'
```

By running this, we will receive an output similar to the following:

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

TASK [Gathering Facts]
*********************************************************************************
ok: [localhost]
```

```
TASK [debug]
*********************************************************************************
ok: [localhost] => {
 "variable": "Hello, World!"
}

PLAY RECAP
*********************************************************************************

localhost : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

Ansible allows variables to be set in various modes and with different priorities. More specifically, you can set them with the following:

- Command-line values (lowest priority)
- Role defaults
- Inventory files or script group [vars]
- Inventory [group_vars/all]
- Playbook [group_vars/all]
- Inventory [group_vars/*]
- Playbook [group_vars/*]
- Inventory files or script host vars
- Inventory [host_vars/*]
- Playbook [host_vars/*]
- Host facts/cached [set_facts]
- Play [vars]
- Play [vars_prompt]
- Play [vars_files]
- Role [vars] (defined in [role/vars/main.yml])
- Block [vars] (only for tasks in block)
- Task [vars] (only for the task)
- [include_vars]
- [set_facts]/registered vars
- Role (and [include_role]) params
- [include] params
- Extra vars (highest priority)

As you can see, the last option (and the highest priority of them all) is using [--extra-vars] in the execution command.

Now that you have learned how to pass working variables via CLI, let's learn how to limit the host's execution.

Limiting the host's execution

While testing a playbook, it might make sense to test on a restricted number of machines; for instance, just one. Let's get started:

1. To use the limitation of target hosts on Ansible, we will need a playbook. Create a playbook called [helloworld.yaml] that contains the following content:

```
---
- hosts: all
  tasks:
```

```
    - debug:
        msg: "Hello, World!"
```

2. We also need to create an inventory with at least two hosts. In my case, I created a file called [inventory] that contains the following content:

```
[hosts]
host1.example.com
host2.example.com
host3.example.com
```

Let's run the playbook in the usual way with the following command:

```
$ ansible-playbook -i inventory helloworld.yaml
```

By doing this, we will receive the following output:

```
PLAY [all]
*********************************************************************************



TASK [Gathering Facts]
*********************************************************************
ok: [host1.example.com]
ok: [host3.example.com]
ok: [host2.example.com]

TASK [debug]
*********************************************************************************
ok: [host1.example.com] => {
    "msg": "Hello, World!"
}
ok: [host2.example.com] => {
    "msg": "Hello, World!"
}
ok: [host3.example.com] => {
    "msg": "Hello, World!"
}

PLAY RECAP
*********************************************************************************

host1.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
host2.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
host3.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

This means that the playbook was executed on all the machines in the inventory. If we just want to run it against [host3.example.com], we will need to specify this on the command line, as follows:

```
$ ansible-playbook -i inventory helloworld.yaml --limit=host3.example.com
```

To prove that this works as expected, we can run it. By doing this, we will receive the following output:

```
PLAY [all]
*********************************************************************************



TASK [Gathering Facts]
*************************************************************************
ok: [host3.example.com]

TASK [debug]
*********************************************************************************
ok: [host3.example.com] => {
    "msg": "Hello, World!"
}

PLAY RECAP
*********************************************************************************

host3.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

Before Ansible executes the playbook we mentioned in the command line, it analyzes the inventory to detect which targets are in scope and which are not. By using the [--limit] keyword, we can force Ansible to ignore all the hosts that are outside what is specified in the limit parameter.

It's possible to specify multiple hosts as a list or with patterns, so both of the following commands will execute the playbook against [host2.example.com] and [host3.example.com]:

```
$ ansible-playbook -i inventory helloworld.yaml --
limit=host2.example.com,host3.example.com

$ ansible-playbook -i inventory helloworld.yaml --limit=host[2-3].example.com
```

The limit will not override the inventory but will add restrictions to it. So, let's say we limit to a host that is not part of the inventory, as follows:

```
$ ansible-playbook -i inventory helloworld.yaml --limit=host4.example.com
```

Here, we will receive the following error, and nothing will be done:

```
 [WARNING]: Could not match supplied host pattern, ignoring: host4.example.com

ERROR! Specified hosts and/or --limit does not match any hosts
```

Now that you have learned how to limit the host's execution, let's learn how to flush the code cache.

Flushing the code cache

Flushing caches in Ansible is very straightforward, and it's enough to run [ansible-playbook], which we are already running, with the addition of the [--flush-cache] option, as follows:

```
ansible-playbook -i inventory helloworld.yaml --flush-cache
```

Ansible uses Redis to save host variables, as well as execution variables. Sometimes, those variables might be left behind and influence the following executions. When Ansible finds a variable that should be set in the step it just started, Ansible might assume that the step has already been completed, and therefore pick up that old variable as if it has just been created. By using the [--flush-cache] option, we can avoid this since it will ensure that Ansible flushes the Redis cache during its execution.

Now that you have learned how to flush the code cache, let's learn how to check for bad syntax.

Checking for bad syntax

Defining whether a file has the right syntax or not is fairly easy for a machine, but might be more complex for humans. This does not mean that machines are able to fix the code for you, but they can quickly identify whether a problem is present or not. To use Ansible's built-in syntax checker, we need a playbook with a syntax error. Let's get started:

1. Let's create a [syntaxcheck.yaml] file with the following content:

```
---
- hosts: all
  tasks:
    - debug:
      msg: "Hello, World!"
```

2. Now, we can use the [--syntax-check] command:

```
$ ansible-playbook syntaxcheck.yaml --syntax-check
```

By doing this, we will receive the following output:

```
ERROR! 'msg' is not a valid attribute for a Task

The error appears to be in
'/home/fale/ansible/Ansible2Cookbook/Ch11/syntaxcheck.yaml': line 4, column 7, but may
be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

  tasks:
    - debug:
      ^ here

This error can be suppressed as a warning using the "invalid_task_attribute_failed"
configuration
```

3. We can now proceed to fix the indentation problem on line 4:

```
---
- hosts: all
  tasks:
    - debug:
        msg: "Hello, World!"
```

When we recheck the syntax, we will see that it now returns no errors:

```
$  ansible-playbook syntaxcheck-fixed.yaml --syntax-check

playbook: syntaxcheck.yaml
```

When the syntax check doesn't find any errors, the output will resemble the previous one, where it listed the files that were analyzed without listing any errors.

Since Ansible knows all the supported options in all the supported modules, it can quickly read your code and validate whether the YAML you provided contains all the required fields and that it does not contain any unsupported fields.

Summary

In this lab, you learned about the various options that Ansible provides so that you can look for problems in your Ansible code. More specifically, you learned how to use host facts to diagnose failures, how to include testing within a playbook, how to use check mode, how to solve host connection issues, how to pass variables from the CLI, how to limit the execution to a subset of hosts, how to flush the code cache, and how to check for bad syntax.

In the next lab, you will learn how to get started with Ansible Tower.

Questions

1. True or False: The debug module allows you to print the value of a variable or a fixed string during Ansible's execution.

A) True

B) False

2. Which keyword allows Ansible to force limit the host's execution?

A) [--limit]

B) [--max]

C) [--restrict]

D) [--force]

E) [--except]