Testing Strategies

Integrating Testing With Ansible Playbooks

Many times, people ask, "how can I best integrate testing with Ansible playbooks?" There are many options. Ansible is actually designed to be a "fail-fast" and ordered system, therefore it makes it easy to embed testing directly in Ansible playbooks. In this chapter, we'll go into some patterns for integrating tests of infrastructure and discuss the right level of testing that may be appropriate.

> **Note**
>
> This is a chapter about testing the application you are deploying, not the chapter on how to test Ansible modules during development. For that content, please hop over to the Development section.

By incorporating a degree of testing into your deployment workflow, there will be fewer surprises when code hits production and, in many cases, tests can be used in production to prevent failed updates from migrating across an entire installation. Since it's push-based, it's also very easy to run the steps on the localhost or testing servers. Ansible lets you insert as many checks and balances into your upgrade workflow as you would like to have.

The Right Level of Testing

Ansible resources are models of desired-state. As such, it should not be necessary to test that services are started, packages are installed, or other such things. Ansible is the system that will ensure these things are declaratively true. Instead, assert these things in your playbooks.

```
tasks:
  - ansible.builtin.service:
      name: foo
      state: started
      enabled: yes
```

If you think the service may not be started, the best thing to do is request it to be started. If the service fails to start, Ansible will yell appropriately. (This should not be confused with whether the service is doing something functional, which we'll show more about how to do later).

Check Mode As A Drift Test

In the above setup, `--check` mode in Ansible can be used as a layer of testing as well. If running a deployment playbook against an existing system, using the `--check` flag to the *ansible* command will report if Ansible thinks it would have had to have made any changes to bring the system into a desired state.

This can let you know up front if there is any need to deploy onto the given system. Ordinarily, scripts and commands don't run in check mode, so if you want certain steps to execute in normal mode even when the `--check` flag is used, such as calls to the script module, disable check mode for those tasks:

```
roles:
  - webserver

tasks:
  - ansible.builtin.script: verify.sh
    check_mode: no
```

Modules That Are Useful for Testing

Certain playbook modules are particularly good for testing. Below is an example that ensures a port is open:

```
tasks:

  - ansible.builtin.wait_for:
      host: "{{ inventory_hostname }}"
      port: 22
    delegate_to: localhost
```

Here's an example of using the URI module to make sure a web service returns:

```
tasks:

  - action: uri url=https://www.example.com return_content=yes
    register: webpage

  - fail:
      msg: 'service is not happy'
```

```
    when: "'AWESOME' not in webpage.content"
```

It's easy to push an arbitrary script (in any language) on a remote host and the script will automatically fail if it has a non-zero return code:

```
tasks:

  - ansible.builtin.script: test_script1
  - ansible.builtin.script: test_script2 --parameter value --parameter2 value
```

If using roles (you should be, roles are great!), scripts pushed by the script module can live in the 'files/' directory of a role.

And the assert module makes it very easy to validate various kinds of truth:

```
tasks:

   - ansible.builtin.shell: /usr/bin/some-command --parameter value
     register: cmd_result

   - ansible.builtin.assert:
       that:
         - "'not ready' not in cmd_result.stderr"
         - "'gizmo enabled' in cmd_result.stdout"
```

Should you feel the need to test for the existence of files that are not declaratively set by your Ansible configuration, the 'stat' module is a great choice:

```
tasks:

   - ansible.builtin.stat:
       path: /path/to/something
     register: p

   - ansible.builtin.assert:
       that:
         - p.stat.exists and p.stat.isdir
```

As mentioned above, there's no need to check things like the return codes of commands. Ansible is checking them automatically. Rather than checking for a user to exist, consider using the user module to make it exist.

Ansible is a fail-fast system, so when there is an error creating that user, it will stop the playbook run. You do not have to check up behind it.

Testing Lifecycle

If writing some degree of basic validation of your application into your playbooks, they will run every time you deploy.

As such, deploying into a local development VM and a staging environment will both validate that things are according to plan ahead of your production deploy.

Your workflow may be something like this:

```
- Use the same playbook all the time with embedded tests in development
- Use the playbook to deploy to a staging environment (with the same playbooks) that simulates productio
- Run an integration test battery written by your QA team against staging
- Deploy to production, with the same integrated tests.
```

Something like an integration test battery should be written by your QA team if you are a production webservice. This would include things like Selenium tests or automated API tests and would usually not be something embedded into your Ansible playbooks.

However, it does make sense to include some basic health checks into your playbooks, and in some cases it may be possible to run a subset of the QA battery against remote nodes. This is what the next section covers.

Integrating Testing With Rolling Updates

If you have read into :ref:`playbooks_delegation` it may quickly become apparent that the rolling update pattern can be extended, and you can use the success or failure of the playbook run to decide whether to add a machine into a load balancer or not.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\reference_appendices\[ansible-devel][docs][docsite][rst][reference_appendices]test_strategies.rst`, line 147); *backlink*
>
> Unknown interpreted text role "ref".

This is the great culmination of embedded tests:

```
---

- hosts: webservers
```

```
    serial: 5

    pre_tasks:

      - name: take out of load balancer pool
        ansible.builtin.command: /usr/bin/take_out_of_pool {{ inventory_hostname }}
        delegate_to: 127.0.0.1

    roles:

       - common
       - webserver
       - apply_testing_checks

    post_tasks:

      - name: add back to load balancer pool
        ansible.builtin.command: /usr/bin/add_back_to_pool {{ inventory_hostname }}
        delegate_to: 127.0.0.1
```

Of course in the above, the "take out of the pool" and "add back" steps would be replaced with a call to an Ansible load balancer module or appropriate shell command. You might also have steps that use a monitoring module to start and end an outage window for the machine.

However, what you can see from the above is that tests are used as a gate -- if the "apply_testing_checks" step is not performed, the machine will not go back into the pool.

Read the delegation chapter about "max_fail_percentage" and you can also control how many failing tests will stop a rolling update from proceeding.

This above approach can also be modified to run a step from a testing machine remotely against a machine:

```
---

- hosts: webservers
  serial: 5

  pre_tasks:

    - name: take out of load balancer pool
      ansible.builtin.command: /usr/bin/take_out_of_pool {{ inventory_hostname }}
      delegate_to: 127.0.0.1

  roles:

     - common
     - webserver

  tasks:
     - ansible.builtin.script: /srv/qa_team/app_testing_script.sh --server {{ inventory_hostname }}
       delegate_to: testing_server

  post_tasks:

    - name: add back to load balancer pool
      ansible.builtin.command: /usr/bin/add_back_to_pool {{ inventory_hostname }}
      delegate_to: 127.0.0.1
```

In the above example, a script is run from the testing server against a remote node prior to bringing it back into the pool.

In the event of a problem, fix the few servers that fail using Ansible's automatically generated retry file to repeat the deploy on just those servers.

Achieving Continuous Deployment

If desired, the above techniques may be extended to enable continuous deployment practices.

The workflow may look like this:

```
- Write and use automation to deploy local development VMs
- Have a CI system like Jenkins deploy to a staging environment on every code change
- The deploy job calls testing scripts to pass/fail a build on every deploy
- If the deploy job succeeds, it runs the same deploy playbook against production inventory
```

Some Ansible users use the above approach to deploy a half-dozen or dozen times an hour without taking all of their infrastructure offline. A culture of automated QA is vital if you wish to get to this level.

If you are still doing a large amount of manual QA, you should still make the decision on whether to deploy manually as well, but it can still help to work in the rolling update patterns of the previous section and incorporate some basic health checks using modules like 'script', 'stat', 'uri', and 'assert'.

Conclusion

Ansible believes you should not need another framework to validate basic things of your infrastructure is true. This is the case because Ansible is an order-based system that will fail immediately on unhandled errors for a host, and prevent further configuration of that host. This forces errors to the top and shows them in a summary at the end of the Ansible run.

However, as Ansible is designed as a multi-tier orchestration system, it makes it very easy to incorporate tests into the end of a playbook run, either using loose tasks or roles. When used with rolling updates, testing steps can decide whether to put a machine back into a load balanced pool or not.

Finally, because Ansible errors propagate all the way up to the return code of the Ansible program itself, and Ansible by default runs in an easy push-based mode, Ansible is a great step to put into a build environment if you wish to use it to roll out systems as part of a Continuous Integration/Continuous Delivery pipeline, as is covered in sections above.

The focus should not be on infrastructure testing, but on application testing, so we strongly encourage getting together with your QA team and ask what sort of tests would make sense to run every time you deploy development VMs, and which sort of tests they would like to run against the staging environment on every deploy. Obviously at the development stage, unit tests are great too. But don't unit test your playbook. Ansible describes states of resources declaratively, so you don't have to. If there are cases where you want to be sure of something though, that's great, and things like stat/assert are great go-to modules for that purpose.

In all, testing is a very organizational and site-specific thing. Everybody should be doing it, but what makes the most sense for your environment will vary with what you are deploying and who is using it -- but everyone benefits from a more robust and reliable deployment system.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\reference_appendices\[ansible-devel][docs][docsite][rst][reference_appendices]test_strategies.rst, line 263)

Unknown directive type "seealso".

```
.. seealso::

   :ref:`list_of_collections`
       Browse existing collections, modules, and plugins
   :ref:`working_with_playbooks`
       An introduction to playbooks
   :ref:`playbooks_delegation`
       Delegation, useful for working with load balancers, clouds, and locally executed steps.
   `User Mailing List <https://groups.google.com/group/ansible-project>`_
       Have a question?  Stop by the google group!
   :ref:`communication_irc`
       How to join Ansible chat channels
```